Spare a Little Change?
Towards a 5-Nines Internet in 250 Lines of Code

Mukesh Agrawal

CMU-CS-11-101
May 2011

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen
Bruce M. Maggs, Duke University
Srinivasan Seshan, Chair
Hui Zhang

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Copyright © 2011 Mukesh Agrawal

This research was sponsored by the National Science Foundation under grant numbers CNS-0435382, ANI-0092678, and ANI-0081396; the Air Force Research Laboratory under grant number F30602-99-1-0518; the Pittsburgh Digital Greenhouse; and AT&T. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAY 2011		**00-00-2011 to 00-00-2011**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Spare a Little Change? Towards a 5-Nines Internet in 250 Lines of Code	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

From its beginnings as a single link between two research institutions in , the Internet has grown in size and scope, to become a global internetwork connecting over  million computers, and . billion users. No longer a niche facility for scientific collaboration the Internet now touches the lives of the world?s population, irrespective of their occupation or geography. It is used by people the world over, to pay bills, read the news, listen to music, watch videos, telephone or video-conference friends and family and much more. The Internet is the premier communications network of our age. Unfortunately, however, there are some respects in which the Internet lags the networks it replaces. In particular, with respect to reliability, the Internet falls far short of the Public Switched Telephone Network which proceeded it. Whereas the PSTN sought, and often delivered the vaunted ?five nines? of reliability, the Internet struggles to compete. As for the cause of this reliability shortfall, available evidence indicates that much of the shortfall is due to the unreliability of IP routers themselves. Given the importance of a reliable Internet to contemporary society, vendors and researchers have proposed a number of solutions to either improve the reliability of individual IP routers, or to make networks more resilient to the unavailability of a single router. While having some promise, these existing solutions face significant obstacles to widespread deployment. Thus, in this dissertation, we endeavor to find or construct a practical, readily deployable, method for mitigating the outages caused by IP routers. To achieve our goal, we take inspiration from previous proposals, which advocated the use of link migration. These proposals improve network resilience, by moving links away from a failed (or failing) router, to an in-service router. To understand the constraints of a practical solution, and resolve the limitations of previous proposals, we conduct extensive experimentation, and study source code and protocol specifications. Using the insights produced by these studies, we construct a practical, readily deployable migration solution with sub-second outage times.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**260**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Keywords: Internet reliability, BGP performance, Quagga

This document includes excerpts of the source code for the Linux operating system kernel, and the Quagga routing software suite. These programs are copyrighted by their authors; excerpts are reproduced here solely for scholarly purposes.

To my parents
मेरे सब उपलब्धि
आप का न्योछावर पर हुआ

and

नानाजी और नानीजी को
दुनिया में कितना दूर
लेकिन दिल में हमेशा पास

Abstract

From its beginnings as a single link between two research institutions in 1969, the Internet has grown in size and scope, to become a global internetwork connecting over 700 million computers, and 1.7 billion users. No longer a niche facility for scientific collaboration, the Internet now touches the lives of the world's population, irrespective of their occupation or geography. It is used by people the world over, to pay bills, read the news, listen to music, watch videos, telephone or video-conference friends and family, and much more. The Internet is the premier communications network of our age.

Unfortunately, however, there are some respects in which the Internet lags the networks it replaces. In particular, with respect to reliability, the Internet falls far short of the Public Switched Telephone Network which proceeded it. Whereas the PSTN sought, and often delivered the vaunted "five nines" of reliability, the Internet struggles to compete. As for the cause of this reliability shortfall, available evidence indicates that much of the shortfall is due to the unreliability of IP routers themselves.

Given the importance of a reliable Internet to contemporary society, vendors and researchers have proposed a number of solutions to either improve the reliability of individual IP routers, or to make networks more resilient to the unavailability of a single router. While having some promise, these existing solutions face significant obstacles to widespread deployment. Thus, in this dissertation, we endeavor to find or construct a practical, readily deployable, method for mitigating the outages caused by IP routers.

To achieve our goal, we take inspiration from previous proposals, which advocated the use of *link migration*. These proposals improve network resilience, by moving links away from a failed (or failing) router, to an in-service router. To understand the constraints of a practical solution, and resolve the limitations of previous proposals, we conduct extensive experimentation, and study source code and protocol specifications. Using the insights produced by these studies, we construct a practical, readily deployable migration solution with sub-second outage times.

Acknowledgments

This dissertation could not, and would not, have come to pass, but for for the abundant help, guidance, and support I have been so fortunate to receive over the past several years. And so, it brings me much joy to finally have the ideal occasion to give thanks, to the many who helped me along the way. I offer my sincere thanks to all those that follow, and my humble apologies to those who I might, quite regrettably, neglect to mention.

Apart from myself, I suspect no one has invested more time in the work leading to this dissertation than my thesis advisor, Srini Seshan. He has shared his time generously for the past nine years. For that alone, I am deeply indebted. In addition, Srini has made the effort, time and again, to find problems that matched my interests and skills. For that, as well, I am truly grateful.

Srini has always set high expectations. Indeed, some goals seemed simply unattainable at the outset. But such goals were set with the understanding that, if they were truly impossible, an explanation of why they were so would suffice. It is an approach that is simultaneously idealistic, and pragmatic. And it is has served me well — both in research, and beyond. Perhaps, then, the best way to understand limits is to push them…

There are many factors that contribute to research success, some more obvious than others. Perhaps one of the most under-appreciated is a solid methodology. Thus, for the successes I have had, I must thank those who helped develop my approach to research. In addition to my research advisors, these include teachers and colleagues, from primary school through grad school.

For contributing to my research methods, I thank specifically: Juanita Mitchell, who emphasized the importance of meticulousness in her science class; Doug Collar, whose insisted on good note-taking in his literature class; Bianca Schroeder, who taught me how to tease insights from intimidatingly large log files; David Nagle, who advised me to work every day, and to work neither too little, nor too much; and Sean Rhea, who taught me that when stuck, it is better to take a walk than to bang one's head against the proverbial wall.

As many before me, I had no small amount of doubt during graduate school. For helping me through those doubting moments, I thank Michael Abesamis, Luis von Ahn, Aditya Akella, Rajesh Balan, Nikhil Bansal, Ashwin Bharambe, Sharon Burks, Shuchi Chawla, Beatrice Chen, Urs Hengartner, John Langford, Kate Larson, Bruce Maggs, Amit Manjhi, Mahim Mishra, David Nagle, Suman Nath, Yamuna Raju, Bianca Schroeder, Srini Seshan, Maverick Woo, and Jeannette Wing. I thank especially Rajesh Balan, Amit Manjhi, Suman Nath, and Srini Seshan for their kind and sustained encouragement to complete my dissertation.

The Computer Science Department at Carnegie Mellon is an amazing place to study, and conduct research. Beyond the caliber of research talent, and the first-rate facilities, lurk two less well-known, but no less important, treasures. These are the administrative staff, and, the not entirely unrelated department culture. Sharon Burks, Deborah Cavlovich, Catherine Copetas, Tracy Farbacher, Barbara Grandillo, Karen Lindenfelser, Angela Miller, and others took care of administrative matters, so that I never had to.

When I returned to Pittsburgh in the Spring of 2010, Deb even had the foresight to find me an office literally two doors down from my advisor, so that I could easily drop in with quick questions. That Deb foresaw what I would need, when I myself had not, is truly remarkable, and quite appreciated. I also thank Catherine and Karen taking the time to welcome me back during that Spring, despite my having been out of touch for so long.

The administrative staff takes the lead in creating a culture that values students not just as students, but as people with interests and needs beyond those of classwork or research. The students themselves build on this foundation, to provide a welcoming, supportive, and fun-loving social environment. In particular, the members of DEC/5 volunteer their time to organize non-work events, to help balance the lives of their fellow students. Thank you to Chris Colohan, Francisco Pereira, Patrick Riley, Ted Wong, Martin Zinkevich, and many others.

While this work owes much to my time, and my mentors, at Carnegie Mellon, I would be remiss not to mention my mentors from before Carnegie Mellon, and from summers away from Carnegie Mellon. From my time as a graduate student at the University of Michigan, I thank Farnam Jahanian, Rob Malan, and Brian Noble. Farnam gave me my start in research, and he, along with Rob, and Brian, encouraged me to continue towards the doctoral degree. From my summers at IBM Research, I thank Dilip Kandlur, Anees Shaikh, and Dinesh Verma. And from my time at AT&T Labs, I thank Bobbi Bailey, Albert Greenberg, Charles Kalmanek, and Jennifer Yates.

My work has, I believe, an unusually large empirical component. While that is my natural tendency, it took others to help me appreciate its value. Farnam and Anees appreciated my experimental skills long before I could do so myself. Max Poletto, my first manager at Meraki, allowed me to pursue a similar approach in my development work there. It was my first opportunity to experiment with, and then apply experimental insights to improve, a production. Successes in those endeavors led me to trully believe in, and fully embrace, the value of experimental work. Thank you, Farnam, Anees, and Max.

The results presented herein come from no fewer than one hundred experiments, of at least ten trials each, conducted between March and Novemeber of 2010. For the laboratory that made these experiments possible, I thank the late Jay Lepreau, and the dedicated, talented, and visionary team at the Utah Network Testbed. Their work, and their vision, has dramatically expanded the limits of what is possible in empirical Computer Science. I hope that my work here has, in its own small way, demonstrated the power and value of their vision.

Good work cannot be appreciated without good presentation. So I thank those who have improved my presentation skills immeasurably over the past many years. I thank Mor Harchol-Balter, my first advisor at Carnegie Mellon, for emphasizing the importance of easy to read, visually appealing graphics. I thank Peter Steenkiste for organizing the Systems Seminar, which gave me many opportunities to practice the art of presenting my work.

I thank also those who gave me good examples from which to learn. For more good presentations than I can count, I thank the speakers at, and organizers of, the Systems Seminar, the Student Seminar Series, and the SDI/LCS Seminar Series. For ideas and examples that improved the the system charts which underpin many explanations of experimental results herein, I thank Edward Tufte and his series of books on data visualization. For many ideas that informed the design of this document, I thank Peter Wilson, the author of the memoir template for LaTeX. And as a veritable font of good presentation ideas, I thank my officemate of many years, Maverick Woo.

For additional resources that enabled my research, and its presentation, I thank the staff of the University of Oregeon Route Views Project, the staff of the Schooner testbed at the University of Wisconsin, and the authors of the software which was vital to the research presented in, and the

production of, this document. This software includes Quagga, matplotlib, git, LaTeX, and oprofile.

Over time, this document developed into something rather larger than expected. I did not set out to produce, and my committee members likely did not expect to receive, a 236 page tome. I thank them for reviewing it nonetheless. I further thank Bruce Maggs and Hui Zhang for cheerfully accepting a dissertation that arrived three years later than expected. I especially thank David Andersen for graciously agreeing to fill an absence resulting from my delay. I thank him, also, for suggesting the five-nines reliability goal. That goal became the central theme of this dissertation.

In addition to those listed above, there are many others who have helped me reach this goal. Though my debt is too large to detail here, I wish thank many other friends, colleagues, and co-conspirators over the past dozen years, including: Kristal Aliyas, Mike Bailey, Marcella Baker, Justin Burke, Moira Burke, Juan Caballero, Meeyoung Cha, Mary Cherng, Sudhakar Cherukuri, Jin Cordaro, Nilesh Dalvi, Brian Davis, Uri Dekel, Bao Do, Joshua Dunfield, Khalid El-Arini, Marvin Eng, Adam Fass, Cliff Frey, Jason Ganetsky, David Greene, Mafan Gong, Jan Harkes, David Helder, Cathy Hong, George Hong, Ningning Hu, Scott Iekel-Johnson, Hetunandan Kamichetty, Min Gyung Kang, Jean Kao, Hyang-Ah Kim, Lea Kissner, Eddie Kohler, Craig Labovitz, Abiruchi Lanjewar, Charles Lefurgy, Carmen Liang, Peggy Liao, James Liang, Ivy Lim, Alice Lin, Chris Lin, Lucian Lita, Julio Lopez, Wai Yong Low, Rob Malan, David Maltz, Pratyusa Manadhata, Pedro Marrón, Prem Melville, Kobus van der Merwe, Troy Nolan, King Ow, Jia-Yu Pan, Shashank Pandit, Jeff Pang, Altin Papa, Jorge Pastor, Dan Pei, Dan Pelleg, Chase Phillips, Shruti Prakash, Steve Raasch, Jade Rarang, Paul Reitsma, John Reumann, Christa Robinson, Monica Rogati, Mugizi Rwebangira, Lina Saleski, Lois Schonberger, Panagiotis Sebos, Vyas Sekar, Aman Shaikh, Rob Shanks, Abhinav Singhal, Shafeeq Sinnamohideen, Danny Sleator, Charlene So, Chris Tam, Ju Kok Tan, Bing Tian, Mitul Tiwari, Brian Tobin, Michael Tschantz, Gaura Veda, Shobha Venkataraman, Patrick Verkaik, Jimmy Wan, Mike Wang, David Watson, Dan Wendlandt, Xiao Yu, Ying Zhang, Jim Zhu, and the Zephyr Crew.

Last, though most certainly not least, I thank my family. I thank Mom and Dad for, amongst so many things, the time that I spent working at home in the Summer of 2010. Just as in years long past, they took care of the everyday concerns, leaving me without worry, and free to focus on my studies. I thank my younger brother, Ritesh, for commiserating about the challenges of graduate school. And I thank my older brother, Rakesh, for his perspectives on life outside of the academic realm.

Mukesh Agrawal

San Francisco, CA
May 2011

Contents

1 Introduction **1**
- 1.1 Existing Solutions . . . 2
- 1.2 Our Thesis . . . 2
- 1.3 Our Approach . . . 3
- 1.4 Scope . . . 3
- 1.5 Our Contributions . . . 3
- 1.6 Dissertation Roadmap . . . 4

2 Baseline **5**
- 2.1 Experiment Environment . . . 6
  - 2.1.1 Network Model . . . 6
  - 2.1.2 Experimental Network Topology . . . 6
  - 2.1.3 Routing Tables . . . 7
  - 2.1.4 Software . . . 10
  - 2.1.5 Measurement Apparatus . . . 11
  - 2.1.6 Experiment Framework . . . 12
- 2.2 Restart Procedure . . . 12
- 2.3 Experimental Results . . . 14
  - 2.3.1 Outage times on Low Spec hardware . . . 14
  - 2.3.2 Benefits of faster hardware . . . 19
- 2.4 Conclusion . . . 24

3 Rehoming **25**
- 3.1 Rehoming Goals . . . 26
- 3.2 Naïve rehoming . . . 26
  - 3.2.1 High-level comparison to router restart . . . 27
  - 3.2.2 In-depth comparison of trials with the shortest overall outage times . . . 27
  - 3.2.3 Comparison of trials with the longest overall outage times . . . 30
  - 3.2.4 BGP timeout behavior . . . 31
  - 3.2.5 BGP session establishment behavior . . . 31
  - 3.2.6 Avenues for improvement . . . 37
- 3.3 Clean shutdown rehoming . . . 37
  - 3.3.1 Rehoming procedure . . . 37
  - 3.3.2 Empirical results . . . 38
  - 3.3.3 Avenues for improvement . . . 38
- 3.4 Conclusion . . . 41

4 Graceful Rehoming **43**
4.1 Introduction to Graceful Restart . . . 44
4.1.1 Overview of Graceful Restart . . . 44
4.1.2 Graceful Restart in Practice . . . 45
4.2 Experiment Setup . . . 46
4.2.1 Code Modifications . . . 46
4.2.2 Forwarding State Parameter . . . 51
4.3 Graceful Restart with clean shutdown . . . 51
4.4 Graceful Restart with naïve rehoming . . . 52
4.5 Graceful Restart with local-preference . . . 53
4.5.1 Configuration Notes . . . 54
4.5.2 Empirical Results . . . 55
4.6 Design discussion . . . 57
4.7 Avenues for improvement . . . 57
4.8 Conclusion . . . 58

5 On Code: Processing Optimizations **59**
5.1 CPU Profiling With OProfile . . . 60
5.2 Optimizing `zebra` . . . 61
5.2.1 Finding hot-spots with OProfile . . . 61
5.2.2 Gleaning behavior from logfile messages . . . 64
5.2.3 Deeper insight from instrumentation and code inspection . . . 66
5.2.4 Resolving the hot-spot, and assessing our improvements . . . 71
5.2.5 Avenues for improvement . . . 72
5.3 Optimizing `bgpd` . . . 74
5.3.1 Finding the hot-spot . . . 74
5.3.2 Resolving the hot-spot, and assessing our improvements . . . 76
5.3.3 Avenues for improvement . . . 78
5.4 Conclusion . . . 80

6 On Timing: Scheduling Optimizations **83**
6.1 Evaluation Framework . . . 84
6.2 Optimizing Session Establishment . . . 85
6.2.1 Finding the problem . . . 85
6.2.2 Our Patch . . . 86
6.2.3 Evaluation and Avenues for Improvement . . . 87
6.3 Optimizing Route Propagation . . . 88
6.3.1 Finding the problem . . . 88
6.3.2 Our Patch . . . 89
6.3.3 Evaluation . . . 90
6.4 Optimizing Route Processing, Part I . . . 93
6.4.1 Finding the problem, and our patch . . . 94
6.4.2 Evaluation . . . 94
6.4.3 Diagnosis . . . 94
6.5 Optimizing Route Processing, Part II . . . 99
6.5.1 Finding the problem, and our patch . . . 100

6.5.2 Evaluation, and Design discussion . . . 103
6.5.3 Avenues for improvement . . . 105
6.6 Conclusion . . . 106

7 ZIRO: Ziro Interruption Rehoming 107
7.1 The Soft Handoff Concept . . . 108
7.1.1 Our route-map . . . 109
7.1.2 Our patch . . . 110
7.1.3 Our rehoming procedure . . . 113
7.2 ZIRO Results . . . 115
7.2.1 Evaluation . . . 115
7.2.2 Diagnosis . . . 118
7.2.3 Revision and Re-evaluation . . . 120
7.2.4 Design Discussion . . . 122
7.3 Simplifying ZIRO . . . 123
7.3.1 Removing changes to scheduling policy . . . 123
7.3.2 Removing CPU optimizations . . . 125
7.4 ZIRO at Scale . . . 126
7.5 ZIRO Interruption . . . 128
7.5.1 Impact of ZIRO on TCP streams . . . 128
7.5.2 Impact of ZIRO on TCP applications . . . 129
7.5.3 Impact on video conferencing . . . 130
7.6 Conclusion . . . 130

8 Dénouement 133
8.1 Design Principles . . . 133
8.2 Related Work . . . 135
8.2.1 Internet Reliability . . . 135
8.2.2 BGP Behavior and Performance . . . 139
8.2.3 Other Related Work . . . 140
8.3 Future Work . . . 141
8.3.1 Experimental Validation . . . 141
8.3.2 Usage Scenarios . . . 142
8.3.3 Configuration Management . . . 143
8.3.4 Generalizing to Other BGP Implementations . . . 144
8.3.5 Improving BGP Implementations . . . 144
8.4 Concluding Remarks . . . 145

A Supplemental System Charts 149

B Source Code 177
B.1 Implementation of router-id Spoofing . . . 178
B.1.1 Core functionality . . . 178
B.1.2 Configuration handling . . . 179
B.2 Improvements to Quagga's Graceful Restart Implementation . . . 184
B.3 Understanding CPU Utilization . . . 189
B.3.1 Capturing scheduler statistics . . . 189

B.3.2 Capturing hash table statistics . . . 192
B.3.3 Miscellany . . . 199
B.4 Reducing CPU Utilization . . . 201
B.4.1 Resolving scheduler bug . . . 201
B.4.2 Improving hash table performance . . . 202
B.5 Scheduling Optimizations . . . 204
B.5.1 Improving session establishment time . . . 204
B.5.2 Improving route propagation delay . . . 209
B.5.3 Improving route processing delay, Part I . . . 211
B.5.4 Improving route processing delay, Part II . . . 213
B.6 Soft Handoff . . . 222

Bibliography **229**

List of Figures

2.1 Typical Tier 1 Internet Service Provider Network . . . 7
2.2 Our experimental network topology . . . 9
2.3 Architecture of the Quagga routing software suite . . . 11
2.4 Illustration of our experiment framework . . . 13
2.5 Partial system chart for behavior during restart of an access router with a single statically routed customer . . . 16
2.6 Partial system charts for behavior during access router restart . . . 20
2.7 Comparison of restart behavior with a single statically routed customer, for different computing hardware . . . 22
2.8 Comparison of restart behavior with a single BGP customer with dynamic routing, for different computing hardware . . . 23

3.1 Naïve rehoming procedure . . . 26
3.2 Partial system chart for router restart, for the trial with the minimum overall outage time . . . 28
3.3 Partial system chart for naïve rehoming, for the trial with the minimum overall outage time . . . 29
3.4 Partial system chart for naïve rehoming, with the initial router configured to have a higher router-id than the target router . . . 30
3.5 Partial system charts comparing router restart and naïve rehoming, for the trials with the longest overall outage times . . . 32
3.6 Partial system charts comparing the trials of naïve rehoming with the shortest and longest outages . . . 33
3.7 Partial system chart for naïve rehoming with router-id spoofing, for the trial with the longest overall outage time . . . 36
3.8 Clean shutdown rehoming procedure . . . 38
3.9 Partial system charts comparing naïve rehoming with router-id spoofing, and clean shutdown rehoming . . . 39
3.10 Partial system chart for clean shutdown rehoming, for the trial with the longest outage time . . . 40

4.1 Example of Graceful Restart in operation . . . 47
4.2 Example of Graceful Restart applied to rehoming . . . 48
4.3 Flowchart for a restarting router during Graceful Restart . . . 49
4.4 Flowchart for a receiving router during Graceful Restart . . . 50

4.5 Partial system chart for clean shutdown rehoming with router-id spoofing and Graceful Restart, for the trial with the minimal overall outage time 52
4.6 Partial system chart for naïve rehoming with router-id spoofing and Graceful Restart 54
4.7 Partial system charts for clean shutdown with router-id spoofing, and naïve rehoming with router-id spoofing, Graceful Restart and LOCAL_PREF, for the trials with the minimal overall outage times 56

5.1 Partial system chart for naïve rehoming with router-id spoofing, Graceful Restart and LOCAL_PREF, for the trial with the minimal overall outage time 60
5.2 Partial system chart for router restart with a single static customer 61
5.3 Partial system charts for rehoming with and without the patch of Listing 5.11, for the trials with the minimal overall outage times 73
5.4 Partial system charts for rehoming with and without hash table resizing patches . . 79

6.1 Partial system chart following the application of our first patch to improve route processing delay 100
6.2 Sequence of packets exchanged by the target router and the customer router, between the start of the outage for traffic to CUSTN, and **+CUSTN**$_{\triangleleft C}$ 105
6.3 Excerpted example of Graceful Rehoming (excerpted from Figure 4.2) 106

7.1 Excerpted example of Graceful Rehoming (excerpted from Figure 4.2) 109
7.2 Soft handoff rehoming procedure 113
7.3 Illustration of race condition that would exist without the "new route" barrier in the soft handoff rehoming procedure 116
7.4 Partial system charts comparing trials of ZIRO, using the soft handoff rehoming procedure 117
7.5 Revised soft handoff rehoming procedure 122
7.6 Partial system charts comparing ZIRO with and without changes to `bgpd` scheduling policies 124
7.7 Partial system charts comparing ZIRO with and without CPU optimizations in `zebra` and `bgpd` 127

A.1 Full system chart for router restart with a single statically routed customer 152
A.2 Full system chart for router restart with a single BGP customer, using default routing for outbound traffic 153
A.3 Full system chart for router restart with a single BGP customer, using dynamic routing for outbound traffic 154
A.4 Full system chart for router restart with a single static customer, on High Spec hardware 155
A.5 Full system chart for router restart with a single BGP customer, using dynamic routing for outbound traffic, on High Spec hardware 156
A.6 Full system chart for naïve rehoming, for the trial with the minimum overall outage time 157
A.7 Full system chart for naïve rehoming, with the initial router configured to have a higher router-id than the target router 158
A.8 Full system chart for router restart, with a single BGP customer, using dynamic routing for outbound traffic, for the trial with the longest overall outage time 159
A.9 Full system chart for naïve rehoming, for the trial with the longest overall outage time 160

A.10 Full system chart for naïve rehoming with router-id spoofing, for the trial with the longest overall outage time . . . 161
A.11 Full system chart for clean shutdown rehoming, for the trial with the longest outage time . . . 162
A.12 Full system chart for rehoming with Graceful Restart and clean shutdown, for the trial with the minimal outage time . . . 163
A.13 Full system chart for naïve rehoming with Graceful Restart . . . 164
A.14 Full system chart for clean shutdown with router-id spoofing . . . 165
A.15 Full system chart for naïve rehoming with router-id spoofing, Graceful Restart and LOCAL_PREF, for the trial with the minimal overall outage time . . . 166
A.16 Full system chart for naïve rehoming with router-id spoofing, Graceful Restart, LOCAL_PREF, and the scheduling patch for `zebra` . . . 167
A.17 Full system chart for rehoming with default hash table sizing in `bgpd` . . . 168
A.18 Full system chart for rehoming with increased hash table sizing in `bgpd` . . . 169
A.19 Full system chart for rehoming following the application of our first patch to improve route processing delay . . . 170
A.20 Full system chart for rehoming with our initial soft handoff rehoming procedure, for the trial with the minimal overall outage time . . . 171
A.21 Full system chart for rehoming with our initial soft handoff rehoming procedure, for the trial with the maximal overall outage time . . . 172
A.22 Full system chart for ZIRO, with our changes to `bgpd` scheduling policies . . . 173
A.23 Full system chart for ZIRO, following the removal of our changes to `bgpd` scheduling policies, but with our CPU optimizations for `zebra` and `bgpd` in place . . . 174
A.24 Full system chart for ZIRO, following the removal of our CPU optimizations for `zebra` and `bgpd` . . . 175
A.25 Full system chart for a trial exhibiting the unsuccessful case of the race condition illustrated in Figure 7.3(b) . . . 176

List of Tables

2.1 Role of each node in the topology . . . 8
2.2 Hardware specifications of experiment nodes . . . 9
2.3 Key parameters of our route traces . . . 10
2.4 Customer routing strategies . . . 15
2.5 Outage times for customers with three different routing strategies . . . 15
2.6 Key to events depicted on system charts . . . 18
2.7 Improvement in outage times achievable through faster hardware . . . 19

3.1 Comparison of outage times for router restart and naïve rehoming . . . 27
3.2 Comparison of outage times for router restart, naïve rehoming and clean shutdown rehoming . . . 38

4.1 Comparison of outage times for clean shutdown, and clean shutdown with router-id spoofing and graceful restart . . . 51
4.2 Comparison of outage times for clean shutdown, naïve, and naïve with router-id spoofing and Graceful Restart . . . 53
4.3 Outage times for naïve rehoming using router-id spoofing, Graceful Restart and local-preference . . . 55

5.1 Top ten functions called by `zebra`, in terms of CPU time . . . 62
5.2 Work-queue statistics for the `route_node processing` workqueue in `zebra` . . . 65
5.3 Improvement due to the patching of `work_queue_run` . . . 71
5.4 Top ten functions called by `zebra`, before and after our patch to `work_queue_run`, in terms of CPU time . . . 72
5.5 Top ten functions called by `bgpd`, in terms of CPU utilization . . . 74
5.6 Hash table statistics for `bgpd`, at the completion of rehoming . . . 76
5.7 Cost of, and improvement due to, the resizing of hash tables . . . 78
5.8 Top ten functions called by `bgpd`, in terms of CPU utilization, with resized hash tables 80

6.1 Breakdown of outage time before any improvements . . . 84
6.2 Comparison of outage times before, and with, our patch for improving session establishment delay . . . 87
6.3 Breakdown of outage time after application of our patch for improving session establishment delay . . . 88
6.4 Comparison of outage times before, and with, our patch for improving route propagation delay . . . 90

6.5 Breakdown of outage time after application of our our patch for improving route propagation delay . . . 92
6.6 Comparison of CPU time used by routing processes before and after application of our patch to improve route propagation delay . . . 92
6.7 Comparison of outage times before, and with, our first patch for improving route processing delay . . . 96
6.8 Breakdown of outage time after application of our first patch for improving route processing delay . . . 97
6.9 Comparison of external and internal breakdowns of outage times, for our first patch for improving route processing delay . . . 97
6.10 Comparison of outage times before, and with, our second patch for improving route processing delay . . . 103
6.11 Breakdown of outage time after application of our second patch for improving route processing delay . . . 104

7.1 Comparison of outage times before, and with soft handoff . . . 116
7.2 Mean delay, over ten trials, between various log file messages and the FIB update for CUST1 . . . 118
7.3 Comparison of outage times before soft handoff, with soft handoff, and with revised soft handoff . . . 122
7.4 Comparison of outage times before and after removal of scheduling policy changes . 125
7.5 Comparison of outage times before and after removal of CPU optimizations . . . 125
7.6 Comparison of outage times for CMU and Google . . . 126
7.7 Top five functions called by bgpd during soft handoff, when rehoming Google . . . 128
7.8 Disruption for constant bitrates stream over TCP . . . 129

8.1 Design principles for rapid recovery of routing sessions . . . 134

A.1 Key to events depicted on system charts (repeats Table 2.6) . . . 151

Listings

3.1 Source code that might generate "Bad BGP Identifier" messages, from the function bgp_open_receive in bgp_packet.c . . . 34
3.2 Source code that generates the "Bad BGP Identifier" messages observed in our experiments . . . 35
3.3 Core source code for patch to enable router-id spoofing . . . 36
5.1 Source code of `acpi_pm_read_verified` and related functions, as annotated by OProfile . . . 63
5.2 Excerpted source code of `thread_fetch`, as annotated by OProfile . . . 64
5.3 Sampling of logfile messages generated by `zebra` during rehoming . . . 65
5.4 Core source code for patch to capture work queue length statistics . . . 66
5.5 Core source code for patch to capture work queue yield counts . . . 66
5.6 Source code of `rib_queue_add` . . . 67
5.7 Source code of `meta_queue_process` . . . 68
5.8 Abstracted source code of `work_queue_run` . . . 69
5.9 Source code of `work_queue_item_requeue` . . . 70
5.10 Source code of `ALL_LIST_ELEMENTS` . . . 70
5.11 Source code for patch to resolve the hot-spot in `zebra` . . . 71
5.12 Line-by-line CPU time for `hash_get`, as called by `bgpd` . . . 75
5.13 Core source code for patch to capture hash table statistics . . . 76
5.14 Core source code, part 1 of 2, for patch to resolve the hot-spot in `bgpd` . . . 77
5.15 Core source code, part 2 of 2, for patch to resolve the hot-spot in `bgpd` . . . 77
6.1 Log file messages from `bgpd`, from the start of the outage for traffic to CUSTN, until `bgpd` has sent its BGP OPEN message to the customer router . . . 86
6.2 Core source code for our patch to improve BGP session establishment time . . . 87
6.3 Log file messages from `bgpd`, relating to the ∼CUSTN and +CUSTN$_{\triangleright R}$ events . . . 89
6.4 Log file messages from `bgpd`, relating to the ∼CUSTN and +CUSTN$_{\triangleright R}$ events . . . 90
6.5 Core source code for our patch to improve route propagation delay . . . 91
6.6 Log file messages from `bgpd`, relating to the ∼CUSTN and +CUSTN$_{\triangleright R}$ events, following application of our patch to reduce route propagation delay . . . 93
6.7 Source code of `bgp_process` . . . 95
6.8 Core source code for our first patch to improve route processing delay . . . 96
6.9 Log file messages from `bgpd`, between the +CUSTN$_{\triangleleft C}$ and ∼CUSTN events . . . 98
6.10 Excerpted source code of `bgp_establish` . . . 101
6.11 Source code of `bgp_announce_table` . . . 102
6.12 Core source code for our second patch to improve route processing delay . . . 103
7.1 The route-map used to identify customer routes, and mark them for soft handoff . . . 110
7.2 Core source code, part 1 of 3, for patch to implement soft handoff . . . 111

7.3 Core source code, part 2 of 3, for patch to implement soft handoff . . . 112
7.4 Core source code, part 3 of 3, for patch to implement soft handoff . . . 113
7.5 Log file messages from `bgpd`, for the five seconds prior to the **enable peering** event . 119
7.6 Source code for the core loop of `bgp_scan`, in `bgp_nexthop.c` . . . 121
B.1 Patch to bgp_open_send, in bgp_packet.c . . . 178
B.2 Patch to struct peer, in bgpd.h . . . 178
B.3 Patch to neighbor_local_id, in bgp_vty.c . . . 179
B.4 Patch to peer_local_id_set, in bgpd.c . . . 180
B.5 Patch to bgp_vty_init, in bgp_vty.c . . . 181
B.6 Patch to bgp_config_write_peer, in bgpd.c . . . 181
B.7 Patch to peer_global_config_reset, in bgpd.c . . . 182
B.8 Patch to peer_group2peer_config_copy, in bgpd.c . . . 182
B.9 Patch to bgp_router_id_set, in bgpd.c . . . 183
B.10 Patch to toplevel of bgpd.h . . . 183
B.11 Patch to bgp_open_capability, in bgp_open.c . . . 184
B.12 Patch to bgp_capability_restart, in bgp_open.c . . . 185
B.13 Patch to bgp_open_receive, in bgp_packet.c . . . 186
B.14 Patch to bgp_clear_route_table, in bgp_route.c . . . 187
B.15 Patch to static const struct FSM, in bgp_fsm.c . . . 188
B.16 Patch to work_queue_run, in workqueue.c . . . 189
B.17 Patch to get_order, in workqueue.c . . . 189
B.18 Patch to struct work_queue, in workqueue.h . . . 190
B.19 Patch to show_work_queues, in workqueue.c . . . 191
B.20 Patch to struct hash, in hash.h . . . 192
B.21 Patch to hash_get, in hash.c . . . 192
B.22 Patch to toplevel of hash.c . . . 193
B.23 Patch to hash_create_size, in hash.c . . . 193
B.24 Patch to hash_create, in hash.c . . . 193
B.25 Patch to hash_free, in hash.c . . . 194
B.26 Patch to show_hash_tables, in hash.c . . . 194
B.27 Patch to bgp_sync_init, in bgp_advertise.c . . . 195
B.28 Patch to aspath_init, in bgp_aspath.c . . . 196
B.29 Patch to cluster_init, in bgp_attr.c . . . 196
B.30 Patch to transit_init, in bgp_attr.c . . . 196
B.31 Patch to attrhash_init, in bgp_attr.c . . . 196
B.32 Patch to community_init, in bgp_community.c . . . 197
B.33 Patch to ecommunity_init, in bgp_ecommunity.c . . . 197
B.34 Patch to distribute_list_init, in distribute.c . . . 197
B.35 Patch to if_rmap_init, in if_rmap.c . . . 197
B.36 Patch to thread_master_create, in thread.c . . . 198
B.37 Patch to cmd_init, in command.c . . . 199
B.38 Patch to toplevel of hash.h . . . 199
B.39 Patch to struct memory_list, in memtypes.c . . . 200
B.40 Patch to toplevel of command.c . . . 200
B.41 Patch to work_queue_run, in workqueue.c . . . 201
B.42 Patch to toplevel of bgp_attr.h . . . 202

B.43 Patch to bgp_sync_init, in bgp_advertise.c . . . 202
B.44 Patch to attrhash_init, in bgp_attr.c . . . 203
B.45 Patch to peer_open, in bgpd.c . . . 204
B.46 Patch to open_ip_bgp_peer, in bgp_vty.c . . . 205
B.47 Patch to bgp_open_vty, in bgp_vty.c . . . 205
B.48 Patch to bgp_open, in bgp_vty.c . . . 206
B.49 Patch to bgp_open_vty_error, in bgp_vty.c . . . 206
B.50 Patch to bgp_vty_init, in bgp_vty.c . . . 207
B.51 Patch to toplevel of bgpd.h . . . 208
B.52 Patch to toplevel of command.h . . . 208
B.53 Patch to bgp_write, in bgp_packet.c . . . 209
B.54 Patch to bgp_update_receive, in bgp_packet.c . . . 210
B.55 Patch to bgp_update_receive, in bgp_packet.c . . . 211
B.56 Patch to work_queue_run, in workqueue.c . . . 212
B.57 Patch to toplevel of bgp_packet.c . . . 212
B.58 Patch to bgp_announce_route, in bgp_route.c . . . 213
B.59 Patch to bgp_update_receive, in bgp_packet.c . . . 214
B.60 Patch to bgp_open_capability, in bgp_open.c . . . 215
B.61 Patch to bgp_capability_restart, in bgp_open.c . . . 216
B.62 Patch to bgp_establish, in bgp_fsm.c . . . 216
B.63 Patch to bgp_process_announce_selected, in bgp_route.c . . . 217
B.64 Patch to bgp_write_packet, in bgp_packet.c . . . 218
B.65 Patch to neighbor_receive_first, in bgp_vty.c . . . 219
B.66 Patch to bgp_vty_init, in bgp_vty.c . . . 219
B.67 Patch to bgp_config_write_peer, in bgpd.c . . . 220
B.68 Patch to struct peer_flag_action, in bgpd.c . . . 221
B.69 Patch to struct peer, in bgpd.h . . . 221
B.70 Patch to bgp_announce_check, in bgp_route.c . . . 222
B.71 Patch to bgp_soft_reconfig_in, in bgp_route.c . . . 223
B.72 Patch to bgp_update_receive, in bgp_packet.c . . . 224
B.73 Patch to route_set_reflect, in bgp_attr.h . . . 225
B.74 Patch to set_reflect, in bgp_attr.h . . . 226
B.75 Patch to no_set_reflect, in bgp_attr.h . . . 226
B.76 Patch to bgp_route_map_init, in bgp_attr.h . . . 226
B.77 Patch to toplevel of bgp_attr.h . . . 227

It's always best to start at the beginning.

Glinda, the Good Witch of the North

1 Introduction

In September 1969, BBN Technologies installed the first node of the ARPANET at the University of California at Los Angeles. One month later, a second node was installed, at Stanford Research Institute [57]. From these modest beginnings, the ARPANET has evolved into the Internet, a global internetwork connecting over 700 million computers [45], and 1.7 billion users [44].

As it has grown in size, the Internet has also grown in scope. No longer a niche facility for the collaboration of US scientists, the Internet now touches the lives of the world's population. It enables the average citizen to pay his bills, read the news, listen to music, watch videos, e-mail his doctor, engage in voice or video chat with his friends and family, and much more. The Internet has grown to be the premier communications network of our age.

Unfortunately, however, there are some respects in which the Internet lags the networks it replaces. In particular, with respect to reliability, the Internet falls far short of the Public Switched Telephone Network which proceeded it. Whereas the PSTN sought, and often delivered the vaunted "five nines" of reliability [51], the Internet struggles to compete. In [68], for example, Reardon reported that IP networks averaged two to six hours of downtime a year, or just "three nines" of reliability.

While detailed data on the cause of failures in the Internet is scarce, available evidence indicates that outages are often due to the unreliability of IP routers. For example, data from a study of a regional ISP, by Labovitz et al. [53], suggest that router faults accounted for up to half of all trouble tickets at that ISP, between November 1997 and November 1998. In a more recent study [43], researchers from Sprint reported that 40% of outages observed between April and August 2002 were likely resolved via a reset of either the control plane, or the forwarding plane, of the router.

Breaking down the sources of unreliability within an IP router, the data from Labovitz et al. [53] indicate that approximately 30% of router failures were due to hardware or software upgrades, while another 20% of failures were due to hardware or software faults. Similarly, Reardon [68] cites a study conducted for Alcatel, which found that 31% of router failures were due to hardware or software upgrades, and an additional 39% of failures were due to hardware or software failure.

1.1 Existing Solutions

Given the importance of the Internet, and the down time caused by unreliable IP routing equipment, vendors and researchers have proposed a number of solutions to either improve the reliability of individual IP routers, or to make networks more resilient to the unavailability of a single router. We highlight representative examples here, deferring a fuller discussion of related work to Chapter 8.

Solutions to improve the reliability of IP routers, offered by many router vendors, typically exploit redundant hardware to mask both software and hardware failures. For example, Cisco, the dominant vendor of IP routers used in ISP networks, offers "Nonstop Forwarding with Stateful Switchover" [18]. With this software feature, and redundant control plane hardware,[1] a router can mask the failure of control plane software or hardware by switching to the backup control plane.

Taking the alternate approach of making networks resilient to single-node failures, researchers have proposed a number of solutions based on the idea of link migration. In these proposals, when a router fails, or requires maintenance, the links incident to it are migrated to an alternate router. The earliest proposal in this vein is that of Sebos et al. [74]; more recent proposals include those of Wang et al.'s [83], and Keller et al. [50].

While having some promise, each approach faces significant obstacles to widespread deployment. For hardware redundancy, the additional capital cost poses a large financial burden. Migration holds the promise of solving this financial problem by reducing the redundancy required. For example, an ISP might share a single backup router amongst all the routers in a point-of-presence, rather than installing redundant hardware in each router.

Unfortunately, however, existing proposals for migration suffer from their own deployment obstacles. Beginning with the earliest proposal, Sebos et al.'s proposal requires changes to neighboring routers, to facilitate rapid migration. Wang et al. eliminate this requirement, by virtualizing routers, and migrating these virtual routers across physical hardware. Unfortunately, however, this requires a significant architectural change to IP routers, and introduces a new compatibility requirement between them. Namely, the routes must support compatible virtualization schemes.

To avoid the challenges of virtualization, Keller et al. proposed a migration scheme which extracts routing state from one router, and reinstantiates that state on another router. This scheme does not require significant architectural changes. Unfortunately, however, it introduces a compatibility requirement of its own: support for the migration of TCP sockets between hosts. Alas, despite much research on TCP socket migration, going back at least to the work of Maltz and Bhagwat in 1998 [60], TCP socket migration has, itself, yet to see broad, interoperable deployment.

1.2 Our Thesis

We believe that a more deployable solution exists, and can be found. Specifically, we contend that:

> It is possible to dramatically improve the reliability of IP routers, and thereby provide order-of-magnitude improvements in Internet reliability, with modest, interoperable, software-only changes to IP routers.

[1] The primary component of the control plane hardware is often referred to as the route processor.

1.3 Our Approach

We view reliability not as a broad architectural challenge, but as a system and performance optimization problem. Namely, informed by the observation of Patterson et al. [66], that system reliability can be improved by minimizing either the Mean Time to Failure, or the Mean Time to Recovery, we seek to understand and improve recovery times following the failure or maintenance of IP routers. To achieve this understanding, and facilitate improvements, we employ extensive experimentation, cross-layer data collection, rich visualization, and the study of source code.

1.4 Scope

To make our work tractable, we limit the scope of this dissertation in two important aspects. In particular:

1. While migration can be used to cope with many causes of router outages, we focus on migration prior to planned maintenance activities, such as hardware or software upgrades.
2. We focus on *access links*, which connect ISP subscribers to the ISP, rather than backbone links (within an ISP), or peering links (between ISPs). This choice is informed by prior work, which indicates that access link failures limit end-to-end reliability to three nines [7].

We discuss the questions of how to apply migration in other failure scenarios, and for other link types, in Chapter 8.

1.5 Our Contributions

The primary technical contributions of this dissertation are:

- a practical, robust, transparent, method for the migration of access links from one IP router to another, while minimizing disruption to network traffic;
- and the demonstration of how this method can be used to facilitate planned maintenance with sub-second outage times.

In addition to this primary contribution, this dissertation makes several substantial, though secondary, contributions. These include:

- a novel, data-rich visualization for understanding the behavior of IP routers when processing BGP messages (see Section 2.3.1);
- two simple, new, pieces of instrumentation for better understanding the performance of the scheduler and BGP message processing code in Quagga, a major open-source IP routing software suite (see Sections 5.2.3 and 5.3.1);
- identifying and resolving several defects in the implementation of high availability support in Quagga (see Section 4.2.1);

- identifying and resolving a defect in Quagga's scheduler, which we found to be the cause of a major performance issue in Quagga's RIB management software (see Section 5.2.4);
- identifying and resolving a performance issue in Quagga's handling of BGP path attributes (see Section 5.3.2); and
- providing insight into the complex scheduling behavior of a real-world BGP router (see Chapter 6).

1.6 Dissertation Roadmap

The remainder of this dissertation is organized as follows:

- In Chapter 2, we introduce our network model, detail our experimental setup, and empirically determine the baseline cost, in terms of seconds of outage time, of a common planned maintenance task: the in-place upgrade of routing software.
- In Chapter 3, we introduce two simple methods for link migration, and then measure and analyze the outage times they achieve.
- In Chapter 4, we explain Graceful Restart, a software component of Cisco's Nonstop Forwarding solution, and how it might be applied to the link migration problem. We then measure and analyze the performance of a link migration scheme which incorporates Graceful Restart.
- In Chapters 5 and 6, we introduce and evaluate further optimizations to our link migration scheme. Chapter 5 focuses on reducing the CPU time used by the routing software, while Chapter 6 introduces scheduling modifications to move non-critical work off the critical path.
- In Chapter 7, we introduce one final optimization, which leverages the cross-node data redundancy inherent in routing protocols. We then consider two significant simplifications to our solution, and demonstrate that they reduce complexity without increasing outage times.
- In Chapter 8, we discuss related work, present our ideas for future work, and offer our concluding remarks.

We suggest that the reader begin with Chapter 2, to establish the appropriate context for interpreting subsequent results, proceed through Chapters 3 and 4 for an understanding of the main concepts behind our migration scheme, and also read Chapter 7 for our end result. Chapters 5 and 6 may be read more selectively, depending on the reader's level of interest in the specific optimizations developed therein. Similarly, Chapter 8 may be read or omitted, concordant with the reader's interest in the broader context of our work.

If we could first know where we are, and whither we are tending, we could better judge what to do, and how to do it.

Abraham Lincoln

2 Baseline

As noted in Chapter 1, one of the most significant causes of downtime in IP networks is the upgrade of routing software itself. Such upgrades can be both, frequent, and highly disruptive. In terms of frequency, Cisco's IOS version 12.4 has, for example, seen twenty-four updates [25] since its initial release in June, 2005.[1] In order to keep current, an ISP would need to upgrade the software on its routers roughly once every 2½ months.

In terms of the degree of disruption caused by an upgrade, Cisco's IOS, the dominant operating system amongst IP routers, requires a full system reboot in order to update routing software. Following the reboot, the router must resynchronize routing state, update forwarding tables, etc. Other systems, such as those running IOS XR, JunOS, or Quagga, do not require a full reboot. They do, however, still require that the router resynchronize routing state, update forwarding tables, and the like.

Despite the frequency of routing software releases, the disruption they cause, and the resulting impact on the reliability of IP networks, there is a paucity of hard data on exactly how long it takes to upgrade routing software, and why the process is so disruptive. In this chapter, we remedy this state of affairs, by examining the behavior of a set of routers as routing software is restarted.

We show that, with the Quagga software router, an 850 MHz Pentium III CPU, a single customer link, and a contemporary-sized routing table (approximately 300,000 entries), the mean outage time experienced by the customer can vary from 110 to 143 seconds, depending on the type of routing used by the customer.

We further show that while faster processors can reduce the outages experienced, substantial down time remains. Upgrading from the 850 MHz system (circa 2000) to a a 3 GHz hyper-threaded Xeon CPU (circa 2005) reduces the mean downtime to between 37 to 64 seconds, again depending on the type of routing used by the customer. However, upgrading to the latest hardware available, a hyper-threaded quad-core Xeon E5530 processor (circa 2010), reduces outages times only marginally, with the outages ranging from 32 to 57 seconds.

[1] These twenty-four updates do not included so-called "rebuild releases", which narrowly target single defects.

Via detailed measurements, and data-rich visualizations, we then explain the difference in outage times between routing strategies, as well as the limitations of relying on CPU improvements to improve outage times. In particular, with regard to CPU improvements, we show that existing routing software is unable to exploit the parallelism offered by modern CPUs.

The remainder of this Chapter is structured as follows:

- In Section 2.1, we introduce and explain the experiment setup that we use throughout the dissertation.
- In Section 2.2, we detail the restart procedure we use to evaluate baseline performance.
- In Section 2.3, we present and analyze empirical measurements of the downtime caused by restarting a router. The measurements cover three different routing strategies, and three hardware configurations, for a total of nine conditions in all.
- In Section 2.4, we summarize our findings, and outline the optimization strategies that we will pursue in Chapters 3 through 6.

2.1 Experiment Environment

To set the stage for our experiments, we now explain our experiment environment. We begin with our high-level network model, and then detail the experimental network topology we use to emulate that model. We next describe the routing table traces used to drive our experiments, as well as our choice of routing software. Finally, we explain our measurement apparatus and automated experiment framework.

2.1.1 Network Model

In order to measure the outage time experienced by a customer when a router is upgraded, we must model and emulate the ISP network. To this end, Figure 2.1 illustrates our model of an ISP network. In this model, the ISP has placed backbone routers at many geographically distinct locations, known as "points-of-presence", or PoPs. The backbone routers are connected via wide-area links. Each PoP also houses one or more access routers. These routers serve as intermediaries between customer routers, and backbone routers.

Note that from the routers' perspective, the links depicted in this diagram are fixed, point-to-point links. Consequently, the opposite end of the link does not change after the link is configured. In practice, however, these links are often reconfigurable at the transport layer. A number of IP routers may be connected to a single transport layer device, such as an add-drop multiplexer (ADM). This provides a level of indirection that enables us to, for example, migrate the customer router's layer-two link to terminate at access router B, instead of access router A.

As noted in Section 1.4, our focus in this dissertation is outages caused by the unavailability of access links. Accordingly, our experiments use a topology focused on the connection between the customer and the access router, as we next describe.

2.1.2 Experimental Network Topology

While the model in Figure 2.1 includes many components, many of them are not vital to understanding, and solving our problem. Accordingly, we use a simplified topology as illustrated in Figure 2.2. Detailed descriptions of the roles of each node are provided in Table 2.1. Note that, in

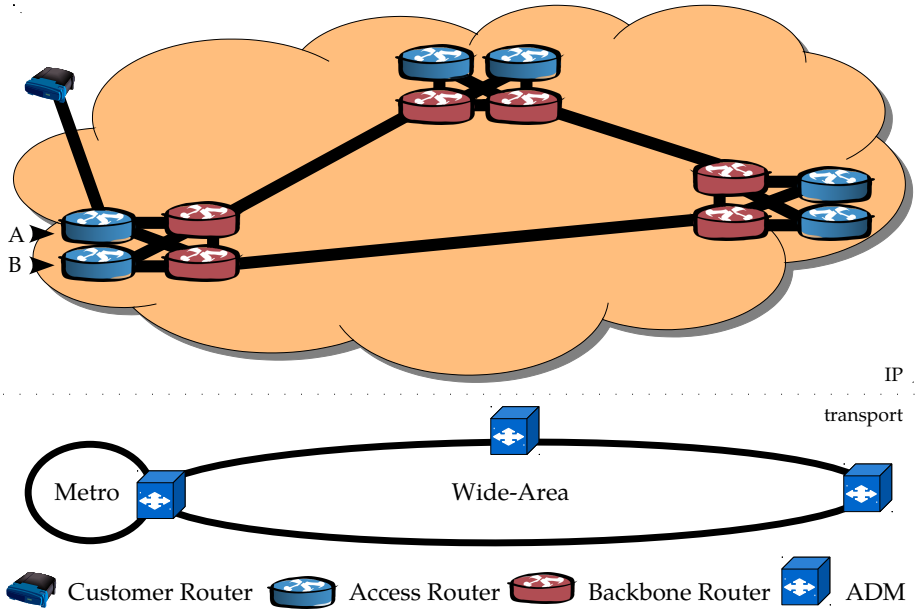


Figure 2.1: Typical Tier 1 Internet Service Provider Network. The upper portion of the diagram illustrates the IP network, while the lower portion illustrates the transport network underlying the IP network.

addition to removing components such as backbone routers, our experimental network topology also adds the "customer sink" and "sink" nodes to facilitate measurement, and the "bridge" node to provide link reconfiguration.

The topology is instantiated on the Emulab [84] at the University of Utah. Depending on the experiment, the nodes are either "Low Spec," "Mid Spec," or "High Spec," as detailed in Table 2.2. In addition to the links depicted in Figure 2.2, each node has a link to a control network, which is used to coordinate the activities of the nodes during the experiment. The control network is not, however, used for the BGP peerings, or the delivery of measurement traffic. The components of our topology are sufficient to emulate critical factors including FIB update time, BGP processing time, and route propagation delays.

2.1.3 Routing Tables

BGP routing performance is significantly influenced by the number of distinct prefixes in the routing system, and the number of distinct path attributes associated across the prefixes. The former impacts the computation required to complete the route decision process, while the latter influences the amount of work required to parse inbound routing messages from BGP peers, and to format outbound routing messages to BGP peers.[2]

[2]When multiple prefixes share the same path attributes, announcements for the group of prefixes can be sent as a single BGP UPDATE message, containing one copy of the shared path attributes, along with an enumeration of the prefixes. For quantitative results on the performance benefits of packing multiple prefixes into a single BGP UPDATE, see[85].

customer sink	1. generates IP datagrams outbound from the customer 2. receives IP datagrams inbound to the customer 3. maintains BGP peering with customer router (as appropriate) 4. injects customer prefixes into the routing system via its BGP peering with the customer router (as appropriate)
customer router	1. maintains BGP peering with customer sink (as appropriate) 2. maintains BGP peering with initial router or target router (as appropriate) 3. forwards IP datagrams between customer sink and initial router or target router (as appropriate)
bridge	1. forwards Ethernet frames between the customer router, and the initial router, or target router (as appropriate)
initial router	1. maintains BGP peering with customer router (as appropriate) 2. maintains BGP peering with target router and remote router 3. forwards IP datagrams to and from customer router
target router	1. maintains BGP peering with customer router (as appropriate) 2. maintains BGP peering with initial router and remote router 3. forwards IP datagrams to and from customer router (as appropriate)
remote router	1. maintains BGP peering with initial router 2. maintains BGP peering with target router 3. maintains BGP peering with sink 4. forwards IP datagrams to and from sink
sink	1. generates IP datagrams outbound from the Internet 2. receives IP datagrams inbound to the Internet 3. maintains BGP peering with remote router 4. injects Internet prefixes into the routing system via its BGP peering with the remote router

Table 2.1: Role of each node in the topology. Some functions may not apply, or may vary, in some experiments. These functions are denoted "(as appropriate)."

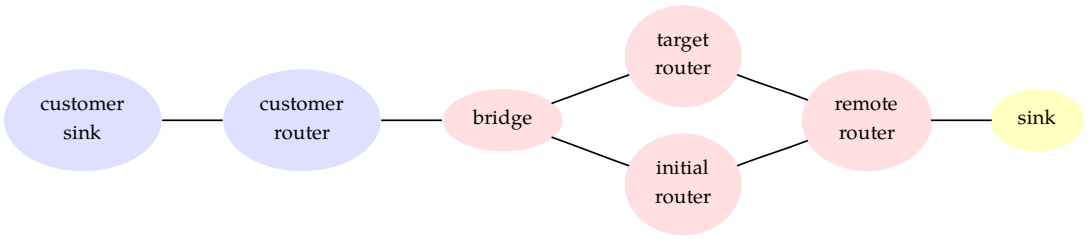


Figure 2.2: Our experimental network topology. Each line indicates a layer-two link. Colors depict autonomous systems.

	CPU Parameters						
	Model	Speed	Cores	Threads	RAM	Network	Vintage
Low Spec	Pentium III	850 MHz	1	1	512 MiB	100 Mbit	2000
Mid Spec	Xeon	3000 MHz	1	2	2048 MiB	100 Mbit	2005
High Spec	Xeon E5530	2400 MHz	4	8	12288 MiB	1000 Mbit	2010

Table 2.2: Hardware specifications of experiment nodes. Note that the Mid Spec machines have Gigabit Ethernet network cards configured in 100 Mbit mode.

To obtain representative results, we conduct our experiments using routing data, obtained from Route Views [81]. Route Views maintains BGP peering sessions with over forty different ISPs, and provides the routing from these peering sessions in two formats. The formats are a) a stream of BGP UPDATE messages, as received by Route Views, and b) a point-in-time dump of the RIB computed by the Route Views router, by processing the update stream without any local policy.

In our experiments, we use a point-in-time RIB dump to seed the RIB on "remote router." To do so, we convert the RIB dump into a sequence of BGP updates, and then use the `sbgp` tool from the Multithreaded Routing Toolkit [54] to replay these updates from "sink" to the BGP routing software on "remote router."[3]

Similarly, we seed the RIB on "customer router" via `sbgp` running on "customer sink." Note that because Route Views only peers with ISPs, we generate the updates for the customer router by processing the RIB dump for a second ISP, and selecting the routes having an ASPATH ending at the customer's AS.

We provide key parameters of the ISP and customer routing data in Table 2.3. We note that while the absolute number of prefixes in the CMU and Google traces are small, this is, in fact, typical of individual customer networks in the Internet. In particular, the CMU and Google traces represent ASes that rank at the 82nd and 99th percentiles in terms of the number of prefixes they originate into the global routing system.

All experiments in this dissertation use the UUNET trace for seeding "remote router." For

[3]It might seem more appropriate to replay a stream of updates as captured by Route Views. However, because Route Views rotates its stream captures every fifteen minutes, and BGP only generates messages when a route changes, we cannot assume that any given stream capture contains complete routing information.

	AS number	# prefixes	# path attributes	first prefix	last prefix
UUNET	701	316592	53799	1.9.0.0	222.255.32.0
CMU	9	7	1	128.2.0.0	209.129.244.0
Google	15169	129	1	8.8.4.0	216.239.60.0

Table 2.3: Key parameters of our route traces. The CMU and Google traces, are generated by filtering the Route Views RIB dump for AS 1239 (Sprint), selecting those prefixes with an AS PATH ending in AS 9 or AS 15169, respectively. The UUNET trace is generated directly from Route Views' RIB dump for AS701. Both RIB dumps were generated by Route Views on July 12, 2010.

seeding "customer router," we use the CMU trace in all experiments except the experiments of Section 7.4.

2.1.4 Software

There are many routing platforms available today, including Cisco's IOS and IOS XR, Juniper's JunOS, GNU Zebra [46], Quagga [48], XORP [39], BIRD [32], and OpenBGPD [16]. We conduct our experiments with Quagga, as i) it runs on readily available hardware and software, ii) its source code is readily available, and iii) it closely models the functionality available in IOS, the dominant routing platform on the Internet today. These properties are important to our work for several reasons:

1. by running Quagga on Fedora Linux and Intel processors, we are able to leverage measurement tools, such as packet capture utilities, and CPU performance counter-based profilers, to gain deep insight into the behavior of the system;
2. without readily available source code, we would not be able to prototype and evaluate our proposed improvements; and
3. the similarity between the features of Quagga and IOS gives some measure of assurance that our changes could be applied to other systems.

We discuss the implications of our work, for other routing platforms, in Section 8.3.4.

Quagga Architecture

Quagga is a multi-process, multi-protocol routing suite, supporting multiple operating systems. It is organized as one process per routing protocol, plus a process which manages interactions with the operating system kernel. These interactions include adding and removing entries from the kernel's routing information base, and listening for changes in network interface status.

Each Quagga process has its own configuration file, as well as a command-line interface, exposed via TCP. Most relevant to our experiments are the `bgpd` and `zebra` processes, which handle the BGP routing protocol, and kernel requests, respectively.[4] We illustrate Quagga's software architecture in Figure 2.3.

[4] For completeness, we note that our experiments also use `ospfd`, to establish routing paths between the ISP-side routers in our topology, using the OSPF routing protocol.

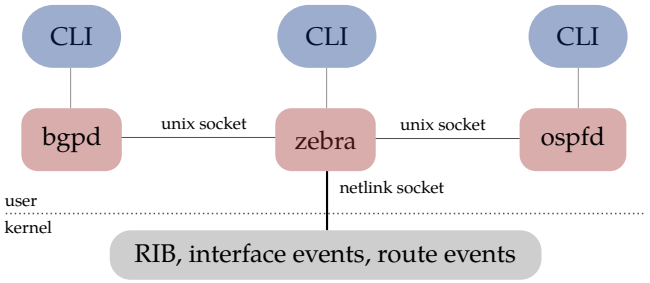


Figure 2.3: Architecture of the Quagga routing software suite.

Configuration Notes

We use Quagga version 0.99.16, and the Fedora Core 10 operating system. Based on past experience, we make two changes to the default Linux socket parameters. First, we increase the size of the socket receive buffers to 10 MB. This provides sufficient buffering so that netlink messages from the kernel to the `zebra` process are not dropped. Second, we increase the size of socket send buffers to 10 MB. This avoids a deadlock we have observed between the `zebra` and `bgpd` processes.

2.1.5 Measurement Apparatus

In order to gain a detailed understanding of system behavior, we instrument the system for measurement across layers, and across nodes in the system. In particular, we capture the following passive measurements:

- all inbound and outbound BGP messages
  (on all nodes running Quagga)
- log messages generated by `bgpd` and `zebra`
  (with microsecond granularity, using default logging levels)
- the time at which each step of restarting or rehoming process is initiated
  (with nanosecond granularity)
- overall system CPU utilization, from the `/proc/stat` pseudo-file
  (captured at one second intervals, with 10 ms granularity)
- CPU utilization of `bgpd`, `ospfd`, and `zebra`, in kernel and user code, from the respective `/proc/<pid>/stat` pseudo-files
  (captured at one second interval intervals, with 10 ms granularity)
- the time at which the FIB entries for the first and last Internet and customer prefixes, hereafter denoted INET1, INETN, CUST1, and CUSTN respectively, are modified
  (polled at one second intervals, reported with nanosecond granularity)

In addition to these passive measurements, we inject ICMP ping packets, to measure the reachability of traffic, both from the customer router to the Internet, and from the Internet to the customer router. This is facilitated and captured by:

- a `ping` process on the customer sink, sending traffic to INET1 (20 ms interval)
- a `ping` process on the customer sink, sending traffic to the INETN (20 ms interval)
- a `ping` process on the sink, sending traffic to CUST1 (20 ms interval)
- a `ping` process on the sink, sending traffic to CUSTN (20 ms interval)
- a capture of all inbound and outbound ICMP packets

Note that it is sufficient to monitor the first and last prefixes because Quagga transmits advertisements in numerical order. One subtlety in this ordering, however, is that prefixes having identical path attributes are "packed" into a single update message.[5] Hence, when we refer to INETN or CUSTN, we mean the last prefixes advertised by the ISP or the customer (respectively), taking in to account both numerical order, and "update packing."

2.1.6 Experiment Framework

BGP performance can exhibit a great deal of variability, due to the effects of timers which govern actions such as the establishment of peering sessions, the transmission of route advertisements, and the processing of periodic tasks such as next-hop reachability checking.[6] To capture the effects of this variation, as well as to provide assurance about the reproducibility of our results more broadly, we built a framework which enables us to readily repeat a single experiment for multiple trials. The framework is illustrated in Figure 2.4, and is realized in under 2200 lines of code. In Section 3.2.4, we highlight one example where timer variation significantly influences outage times.

2.2 Restart Procedure

Having explained our experiment environment, we now turn to the details of the the procedure we use to restart our router. As the Quagga routing software contains three processes (`zebra`, `ospfd`, and `bgpd`), there are numerous procedures we might use for restarting the software. These procedures vary in the order that the processes are restarted, and whether the processes are restarted concurrently or serially.

While the simplest procedure would be to restart all of the processes concurrently, we found this method to be unreliable. Specifically, in some trials with the Low Spec machines, the system would converge to a state where the first customer prefix was installed in the FIB, but the last customer prefix was not. As we were unable to reproduce this behavior with the Mid Spec machines, we

[5] For example, if prefixes 1.0.0.0/8 and 3.0.0.0/8 share path attributes, while 2.0.0.0/8 does not, the message containing 3.0.0.0/8 will arrive before the message containing 2.0.0.0/8.

[6] The timers for establishing peering sessions and transmitting route advertisements include the ConnectRetryTimer, DelayOpenTimer, and IdleHoldTimer. Timers governing route advertisements include the MinASOriginationIntervalTimer, and MinRouteAdvertisementIntervalTimer. More details about these timers are available in the BGP specification [69]. Timers for periodic tasks, such as next-hop reachability checking, are implementation-dependent.

customer sink	customer router	bridge	initial router	target router	remote router	sink
			start monitors			
start pings						start pings
			monitors ready			
INET1 reachable?						CUST1 reachable?
INETN reachable?						CUSTN reachable?
sleep 20						sleep 20
			start			
			<execute experiment code>		new CUSTN in bgpd?	CUSTN reachable?
new route					new route	
INETN reachable?						
rehome reach						rehome reach
	INET1 in FIB?					
	INETN in FIB?					
	>300k routes in bgpd?					
	>300k routes in FIB?					
			done			done
			CPU idle? sleep 20			
			trial complete			
			stop monitors			
			<revert changes>		old CUSTN in bgpd?	
old route					old route	
INET1 reachable?						CUST1 reachable?
INETN reachable?						CUSTN reachable?
stop pings						stop pings
revert reach						
	sleep 3-4 min					
			revert compete			
			CPU idle?			CPU idle?

Figure 2.4: Illustration of our experiment framework. The columns list the steps taken by each node. Light gray bars indicate actions repeated across several nodes. Medium gray bars indicate barriers used to synchronize actions across nodes. Note that steps between barriers are unordered with respect to actions of other nodes. Note further that the steps after "stop monitors" are omitted for restart experiments, except for the 3-4 minute sleep.

believe the behavior is a symptom of a race condition between the deletion of old routes, and the insertion of new ones.[7]

In order to reliably restart the routing software, we use the following procedure:

1. signal `zebra` to terminate
2. wait for `zebra` process to exit
3. start `zebra`
4. wait for the processor to become idle
5. restart `ospfd`
6. restart `bgpd`

The key to the success of this procedure is step 4, which allows `zebra` to complete its startup tasks, including the removal of routes installed by the previous process, before permitting route modifications by `ospfd` or `bgpd`.[8]

2.3 Experimental Results

Having detailed our experiment environment, and the restart procedure, we are now ready to describe our specific experiments, and present our findings. Our experimental results for restarting a router are organized as follows:

- First, in Section 2.3.1, we consider the outage times on Low Spec hardware, for customers with three typical routing strategies. We show that the mean outage time ranges from 110 to 143 seconds, and that much of this time is due to computation delays.
- Second, in Section 2.3.2, we repeat our experiments on Mid Spec and High Spec hardware, to determine the extent to which outage times can be reduced through the use of faster hardware. We find, however, that neither can achieve "five-nines" reliability for customers using BGP with dynamic routing, in the face of a routing software upgrade once every 2½ months.

Note that all experiments are repeated for ten trials. When presenting our results, we generally begin with summary statistics over these ten trials. We then proceed to provide deeper insight into the factors behind the observed behavior by highlighting particular features of representative trials.

2.3.1 Outage times on Low Spec hardware

Herein, we study the outage times experienced when an access router is restarted. We study the outage times for three different kinds of customers, grouped by their routing strategies. These three routing strategies are: static, BGP with default routing, and BGP with dynamic routing. Each routing specifies how routes for traffic inbound to, and outbound from, the customer, are established. The strategies are detailed in Table 2.4. Note that, for BGP with default routing, the initial router is configured not to transmit any route advertisements on its peering session with the customer router.

[7] We have not attempted this restart procedure with the High Spec machines.

[8] In principle, the shell code for restarting `zebra` should accomplish the same, as it calls `ip route flush proto zebra` before starting the new `zebra` process. However, we found that `ip` command would terminate before removing all of the routes installed by `zebra`. This may be a defect in the version of the `ip` command in Fedora Core 10.

	static	bgp + default	bgp + dynamic
inbound	customer prefixes configured on ISP router	customer prefixes advertised to ISP via BGP	customer prefixes advertised to ISP via BGP
outbound	default route configured on customer router	default route configured on customer router	Internet routes acquired through BGP

Table 2.4: Customer routing strategies. Columns list the strategy names, and rows describe how they establish routes for traffic inbound to, and outbound from, the customer.

	static	bgp + default	bgp + dynamic
internet to CUST1	29.56	35.14	58.19
internet to CUSTN	21.81	30.63	54.15
customer to INET1	40.36	41.03	73.69
customer to INETN	96.77	97.97	129.67
any	109.92	111.82	143.17

Table 2.5: Mean outage times, in seconds, and over ten trials, for customers with three different routing strategies. The row labeled "any" gives the total time during which any of the four prefixes is unreachable. If the outages for these prefixes overlapped completely, the "any" outage time would be the maximum of the rows above it. If the outages did not overlap at all, the "any" outage time would be sum of the rows above it.

Conducting experiments with these different routing strategies is valuable for two reasons. First, these strategies are representative of different real-world customers. Second, by comparing the system's behavior for these different routing strategies, we can better understand and isolate the causes of load on the system.

High-level results

We begin our study of outage times with Table 2.5, which gives the mean outage times for these three routing strategies. The table lists the outage time for traffic to each of INET1, INETN, CUST1, CUSTN, as well as the total outage duration. From the data in the table, we can make several important observations:

1. The outage times are significant barriers to achieving "five-nines" reliability. Even the static routing strategy sees a mean outage time of approximately 110 seconds. Thus, a single router restart consumes approximately 35% of the annual outage budget.

2. Static routing and BGP with default routing see similar overall outage times. However, BGP with dynamic routing sees significantly longer overall outage times.

3. Outages for traffic to INET1 are considerably shorter than outages for traffic to INETN.

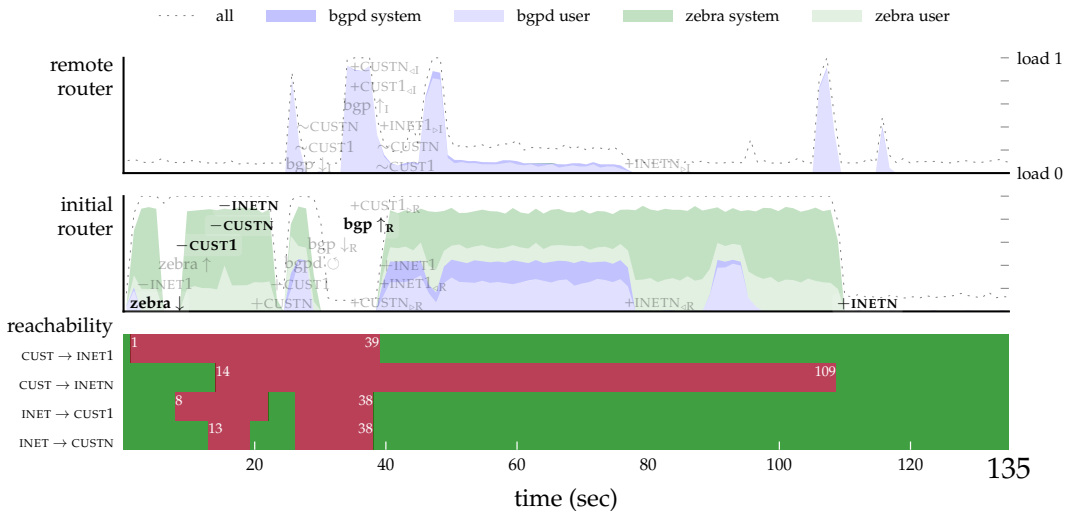


Figure 2.5: Partial system chart for behavior during restart of an access router with a single statically routed customer. This chart illustrates the trial with the shortest total ("any") outage time. The target router, bridge, and customer router are omitted, as they do not significantly contribute to observed behavior. A larger rendering of this chart, is provided as Figure A.1.

4. Surprisingly, for CUST1 and CUSTN, the story is reversed. That is to say that outages to CUST1 are longer than those for CUSTN. The magnitude of the difference is, however, significantly smaller.

Visualizing system behavior

For a deeper understanding of the mechanisms behind these outages, we need more details about the system behavior, and the ability to correlate events across the nodes in the system. To that end, we introduce our *system chart*. The system chart leverages our battery of measurements to provide a single visualization that captures the salient aspects of the system's behavior.

Figure 2.5 presents the system chart for the restart of an access router which hosts a single statically routed customer. The top two graphs present the CPU load on the remote router, and the initial router, respectively. The shaded areas indicate CPU load for the `bgpd` and `zebra` processes, broken down by user mode and system mode time, as denoted in the legend. The light dotted line indicates total CPU load. In cases where CPU load is due to some process other than `bgpd` or `zebra`, the dotted line exceeds the sum of the blue and green shaded areas.

In addition to the illustrating the CPU load, each of the top two graphs also illustrates the timing of important events during the rehoming process. For each such event, we place an annotation on the graph for the node where the event occurred, and at the x-axis position for the time at which the event occurred. For example, **zebra** ↓, the leftmost label and marker on the graph labeled "initial router", indicates that the experiment framework signalled the zebra process to terminate one second after the start of the experiment.[9]

[9]The ordering of closely spaced events can be determined by their vertical positioning. The later event will appear above

We describe the complete set of events illustrated throughout this dissertation, including how they are captured, and the precision of their timestamps, in Table 2.6. Note that, for graphs in the body of the dissertation, some annotations are shown in gray type, while other are shown in black. This formatting emphasizes events referenced from the text, while still presenting the full set of events observed.

Finally, the bottommost graph indicates the reachability of different IP address prefixes, as measured from two vantage points. Green areas denote times during which packets are delivered successfully, and red areas denote times during which packets are lost. For example, the large red region in the series labeled **cust** → **inetn**, indicates that from time 14 seconds to time 107 seconds, packets sent from the customer sink towards inetn were not received at "sink".

In-depth analysis

Having introduced our system charts, we are now equipped to present a deeper analysis of the system behavior. We now use our system charts to illuminate the causes of each of the behaviors highlighted in our high-level results above.

1. Regarding the overall outage time, we see in Figure 2.5 that most of the outage can be attributed to CPU processing delays on the initial router. Because there is no BGP session between the customer router and the initial router, the CPU load on the initial router must be due to the processing of updates from the remote router.[10]

   One obvious way to avoid the outage caused by these delays to move migrate the customer to another router before upgrading the initial router. Accordingly, we introduce and evaluate techniques for link migration in Chapter 3.

2. We next consider the similarities and differences between the overall outages times for the three different types of customers. In order to understand why the outage times for static customers are so similar to those for BGP customers using default routing, and why the outage times for BGP customers with dynamic routing are much higher, we present Figure 2.6.

   This figure compares the system charts for the trials with the minimum outage times, for each routing strategy. Examining Figures 2.6(a) and 2.6(b), we see that, as for static routing, the dominant factor in outage times for BGP customers with default routing is CPU processing time on the initial router. In contrast, Figure 2.6(c) shows that, for BGP customers with dynamic routing, the dominant factor is the processing time on the customer router.

   We further note that the length of the computation on the initial router is appreciably larger for BGP customers with dynamic routing, than the other two routing strategies. Quantifying this difference, we find that, in the mean, BGP customers using dynamic routing require an additional 8 seconds of computation by the BGP process on the initial router, as compared to the other two routing strategies. This likely due to the work required to generate and transmit routing advertisements to the customer router.

the earlier event, except in the case where the earlier event occupies the topmost position in the graph. In such cases, the later event occupies the bottommost position.

[10] Updates from the peering between the initial router's peering with the target router are not an issue, because i) the target router does not originate any routes of its own, and ii) in the restart experiments, the target router does not maintain any EBGP peerings, and iii) the routes that the target router has learned from the remote router can not be advertised to the initial router, because both the remote router and the initial router are IBGP peers of the target router.

	description	capture method	precision
bgpd ↻	the framework restarted the `bgpd` process	experiment logfile	nanosecond
bgp $\downarrow_{c}$	the BGP peering with the customer went down (also initial router, remote router, and ISP router.)	`bgpd` log file	microsecond
bgp $\uparrow_{c}$	the BGP peering with the customer came up (also initial router, remote router, and ISP router.)	`bgpd` log file	microsecond
∼CUST1	the nexthop IP address of the kernel FIB entry for CUST1 has changed (also CUSTN, INET1, and INETN)	polling kernel FIB (once per second)	nanosecond
−CUST1	the kernel FIB entry for CUST1 has been removed (also CUSTN, INET1, and INETN)	polling kernel FIB (once per second)	nanosecond
+CUST1	a kernel FIB entry for CUST1 has been added (also CUSTN, INET1, and INETN)	polling kernel FIB (once per second)	nanosecond
+CUST1$_{\triangleleft C}$	a BGP advertisement for CUST1 was received from the customer router (with variations for different prefixes and BGP peers)	packet capture	microsecond
+CUST1$_{\triangleright ISP}$	a BGP advertisement for CUST1 was sent to the ISP router (with variations for different prefixes and BGP peers)	packet capture	microsecond
-CUST1$_{\triangleleft I}$	a BGP withdrawal for CUST1 was received from the initial router (also CUSTN)	packet capture	microsecond
-CUST1$_{\triangleright R}$	a BGP withdrawal for CUST1 was sent to the remote router (also CUSTN)	packet capture	microsecond
disable nic	the framework disabled the customer-facing network card	experiment logfile	nanosecond
enable nic	the framework enabled the customer-facing network card	experiment logfile	nanosecond
enable peering	the framework enabled a BGP peering between the ISP and the customer	experiment logfile	nanosecond
move link	the framework reconfigured the layer-2 link between the ISP and the customer	experiment logfile	nanosecond
open$_{\triangleright C}$	a BGP OPEN message was received from the customer router	packet capture	microsecond
reject peer	the customer router rejected a peering request from the ISP	`bgpd` log file	microsecond
zebra ↓	the framework signalled the `zebra` process to terminate	experiment logfile	nanosecond
zebra ↑	the framework started a new `zebra` process	experiment logfile	nanosecond

Table 2.6: Key to events depicted on system charts. Note that not all events occur on all charts.

	static	bgp + default	bgp + dynamic
Low Spec	109.92	111.82	143.17
Mid Spec	36.83	37.45	64.52
High Spec	32.02	30.97	56.70

Table 2.7: Improvement in outage times achievable through faster hardware. For each hardware configuration, we show the mean outage time, over ten trials. Note that the "Low Spec" row of this table repeats the "any" row of Table 2.5.

We tackle the problem of customer router computation in Chapter 4, and we optimize BGP processing on the ISP router in Chapter 5.

3. For all routing strategies, the difference in outage times for INET1 and INETN can be explained by CPU processing time. For static routing, or BGP routing with a default route, this processing occurs on the initial router; for BGP with dynamic routing, the processing occurs on the customer router. Surprisingly, the `zebra` uses more CPU time than the `bgpd` process. This suggests that FIB processing is more expensive than BGP processing.

4. To understand the difference in outage times between CUST1 and CUSTN, we focus on Figure 2.6(a), as the other trials depicted in Figure 2.6 do not show any difference between these two prefixes. Specifically, we consider the annotations −**CUST1**, −**CUSTN**, −**INETN**, and the outage graphs for **INET** → **CUST1** and **INET** → **CUSTN**.

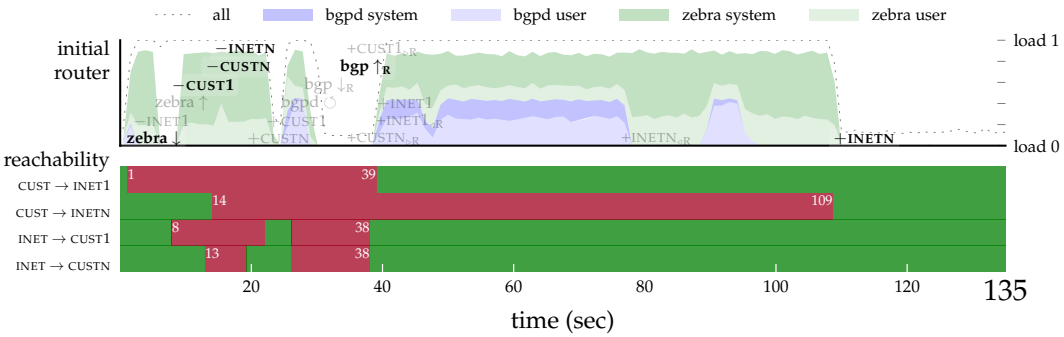
   We first observe that the outage start times correlate well with the annotations for the FIB changes. We next note that the CPU on the initial router is busy in the `zebra` process during this time, and then conclude that the difference in outage times for these prefixes is due to the cost of FIB updates.

   The difference in outage times might seem out of proportion to the number of prefixes originated by this customer (just seven, as listed in Table 2.3). Note, however, that these seven prefixes span a large portion of the IP address space: from 128.2.0.0 to 209.129.244.0. Thus, there can be a considerable delay between the processing of these two prefixes, as the router performs FIB processing for other prefixes in the Internet.

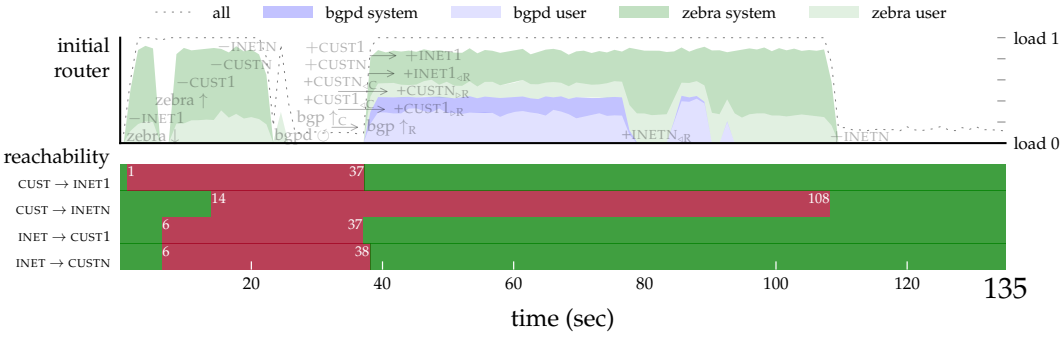
2.3.2 Benefits of faster hardware

Having observed that much of the outage time can be attributed to CPU processing time, it is valuable to consider the extent to which outage times can be improved through the use of faster processors. To that end, we present Table 2.5. This table shows the overall outages time observed with Low Spec, Mid Spec, and High Spec hardware.

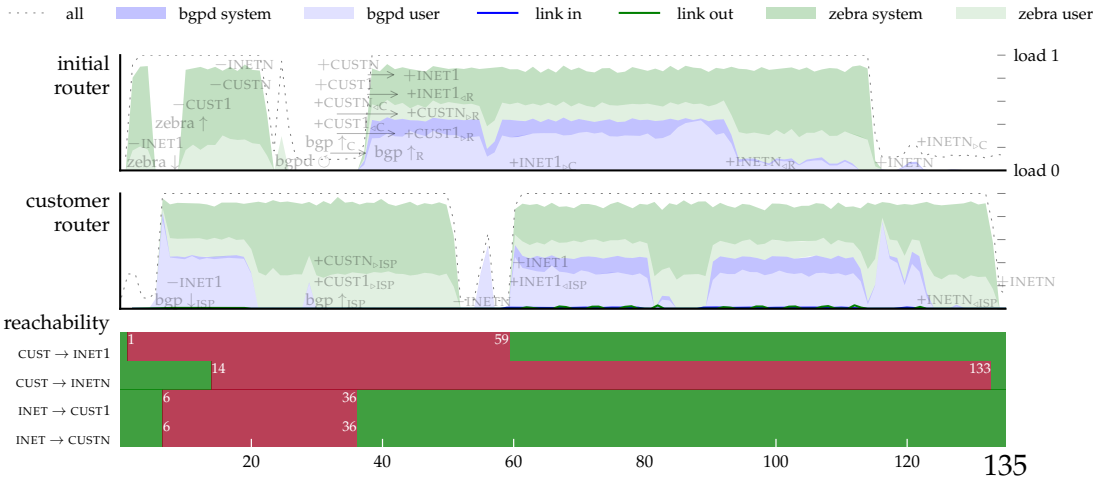
At a high level, we observe that the Mid Spec hardware reduces outage times for static, and BGP with default routing, by approximately 65%. Thus, a single router restart would consume slightly over 10% of the annual outage budget. Given the rate of software releases, however, keeping routing software current could easily consume half the annual outage budget. For BGP customers with dynamic routing, the disruption caused by software updates alone would make it impossible to achieve "five-nines" reliability.



(a) statically routed customer



(b) BGP customer with default route to Internet



(c) BGP customer with dynamic routing to Internet

Figure 2.6: Partial system charts for behavior during access router restart. These charts illustrate trials with the shortest "any" outage times. For ease of comparison a reduced version of Figure 2.5 is repeated here as subfigure (a). The full versions of these three charts are provided as Figures A.1 through A.3.

Intriguingly, the move from Mid Spec hardware to High Spec hardware provides much more modest improvements, of 12-17%. Why is it that, despite a historical trend of computing power doubling every 18 months, a full decade of improvements in computing hardware has proved insufficient to solve our problem? To answer this question, we again turn to our system charts. We first consider statically routed customers, then those using BGP with dynamic routing.

Improvement for a statically routed customer

Figure 2.7 presents system charts for restarting of a router with a single statically routed customer, comparing the behavior of Low Spec and High Spec hardware. From this figure, we make three observations:

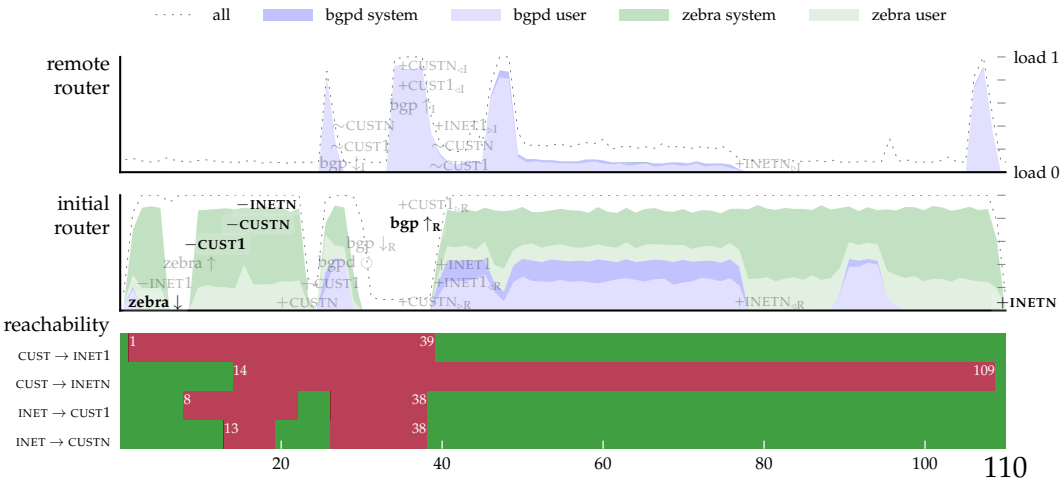
1. A significant fraction of the outage time is no longer due to CPU processing delays. Specifically, we note that from time 3 seconds to 15 seconds in Figure 2.7(b), processors on both the remote router and initial router are idle. During much of this time, `bgpd` on the initial router has restarted (as denoted by **bgpd** ↻), but not yet re-established the peering session with the remote router (as denoted by **bgp** $\uparrow_R$). We investigate delays in peering session establishment, and how to resolve them, in Chapter 6.
2. The newer hardware has reduced the wall-clock time required by the post-restart `zebra` and `bgpd` processes from approximately 70 seconds to approximately 13 seconds. This roughly factor-of-five improvement is far short of historical norms, which would have predicted a 64-fold improvement.
3. The CPU utilization on the initial router never exceeds 2, despite the fact that the hardware supports 8 threads. Thus, much of the shortfall between the factor-of-five improvement in CPU processing time, and the 64-fold improvement we might expect, is due to the mismatch between the evolution of software and hardware. Most software is simply not prepared to take advantage of hardware support for multithreaded execution.[11]

Improvement for a BGP customer with dynamic routing

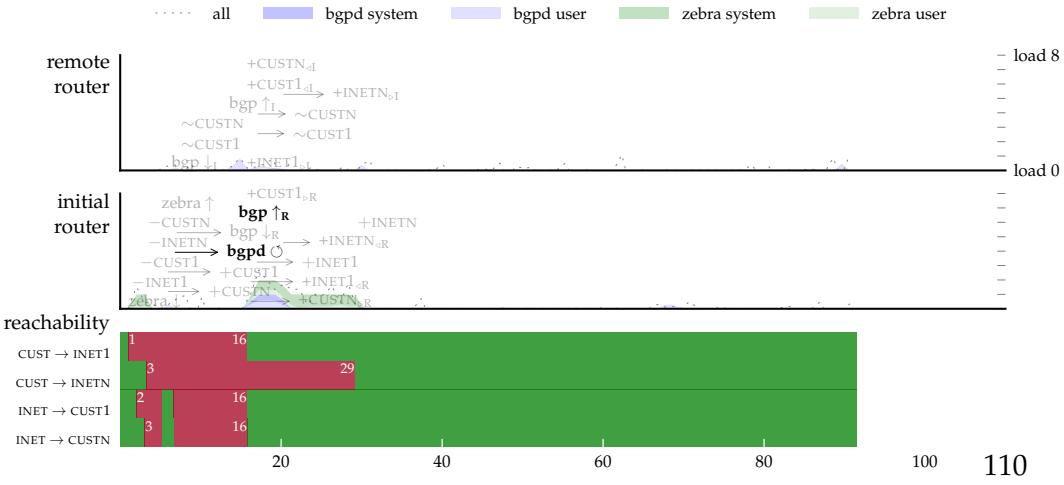
We next analyze the benefits of faster hardware for a BGP customer with dynamic routing, using the system charts of Figure 2.8. From this figure, we make the following observations:

1. With High Spec hardware, as illustrated in Figure 2.7(b), there is no significant period of time during which all nodes in the system are idle. During the times when the initial router is idle, the customer router is busy, and vice versa.
2. The High Spec hardware reduces the post-restart wall-clock time required by the `zebra` and `bgpd` processes on the customer router from about 74 seconds to about 10 seconds. This is a greater factor of improvement than observed for computation on the initial router, but still far short of the 64-fold improvement we might expect from historical norms.
3. As with the statically routed customer, the routing software lacks sufficient thread-level parallelism to fully utilize the CPU hardware.

[11] Both `zebra` and `bgpd` are structured as multithreaded programs. However, they both use cooperative, user-level threads. Thus, the operating system cannot schedule the threads for simultaneous execution on the hardware.

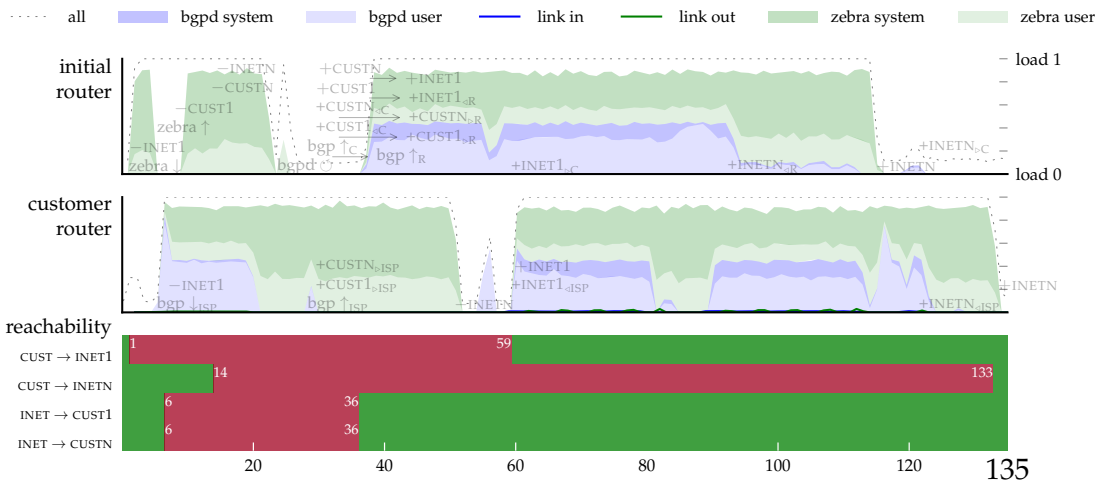


(a) Low Spec nodes

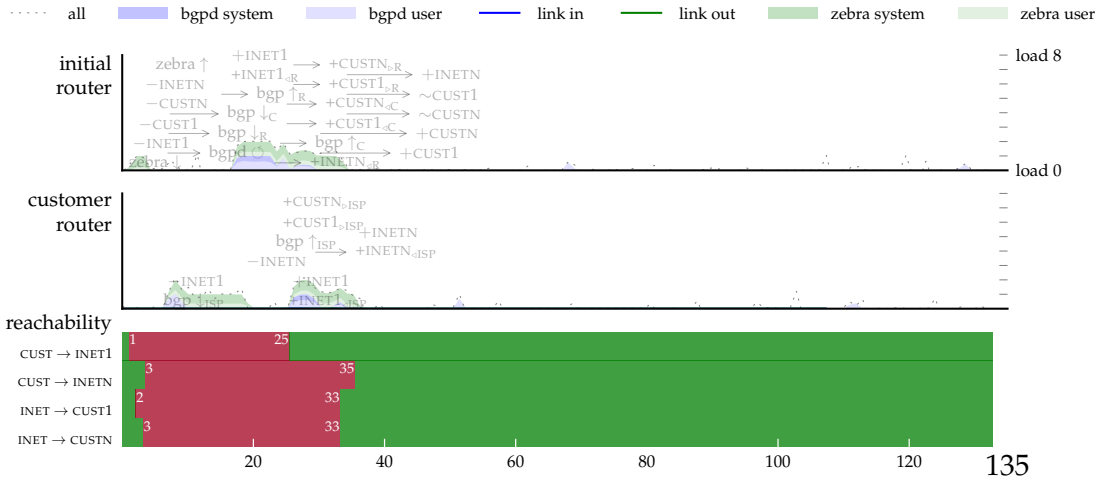


(b) High Spec nodes

Figure 2.7: Comparison of restart behavior with a single statically routed customer, for different computing hardware. The customer router is omitted from the charts, as it does not contribute significantly to the observed behavior. The charts are presented on a common scale for ease of comparison; as the High Spec hardware completes the experiment more quickly than the Low Spec hardware, no data exists for the area at the right of the High Spec chart. Full versions of these charts are provided as Figures A.1 and A.4.



(a) Low Spec nodes



(b) High Spec nodes

Figure 2.8: Comparison of restart behavior with a single BGP customer with dynamic routing, for different computing hardware. The remote router is omitted from the charts, as it does not contribute significantly to the observed behavior. Full versions of these charts are provided as Figures A.3 and A.5.

2.4 Conclusion

In this chapter, we set out to understand the impact of restarting one router in a system of BGP routers. Using real-world traces, and 850 MHz Pentium III processors, we showed that the outage times vary from 110 to 143 seconds, depending on routing strategy.

Digging more deeply, we identified several avenues for improvement, including: i) isolating the customer from the restart-related computation on the initial router, ii) reducing computation on the customer router, iii) optimizing BGP processing on the ISP router, and iv) reducing the time taken to re-establish BGP peerings. We pursue these improvements in Chapters 3, 4, 5, and 6, respectively.

Of course, before pursuing these changes, it is important to consider whether our problems could be solved more simply through the use of faster computing hardware. To that end, we repeated our experiments with 3 GHz Xeon processors, and with 2.4 GHz quad-core Xeon processors. The systems represent five years, and a full decade, of advancement in computing hardware, respectively. Unfortunately, however, neither system can achieve "five-nines" reliability for customers using BGP with dynamic routing, in the face of a routing software upgrade once every 2½ months.

Before proceeding, we note that in subsequent chapters, we focus solely on BGP customers using dynamic routing. We choose to focus on these customers as i) they have the largest outage times, and ii) they subsume the effects observed for statically routed customers, and BGP customers with default routing.

All problems in computer science can be solved by another level of indirection.

David Wheeler

3 Rehoming

Our experiments of Chapter 2, which quantified the effects of restarting an access router, revealed that such a restart can cause mean outage times of approximately 110 to 140 seconds, depending on the routing strategy employed by the customer. Our analysis of those experiments identified several bottlenecks, including restart-induced computation on the initial router.

In this chapter, we seek to eliminate that bottleneck. Our strategy for doing so is to migrate, or *rehome*, the customer from the router to be restarted, known as the initial router, to a spare router, known as the target router. After the migration is complete, the initial router can be restarted without disrupting connectivity for the customer.

To that end, we introduce two rehoming schemes, called "naïve", and "clean shutdown". We show that the simpler scheme, naïve, eliminates ISP-side computation as a bottleneck. However, it introduces complications which result in a net increase in outage times. The clean shutdown scheme, though somewhat more complex, avoids these complications. Thus, clean shutdown both eliminates ISP-side computation as a bottleneck, and achieves a net reduction in outage times.

The remainder of this chapter is structured as follows:

- In Section 3.1, we speak to the need for our solution to be deployable and usable, and the constraints those goals imply.
- In Section 3.2, we introduce the naïve rehoming scheme, evaluate it experimentally, and identify avenues for improvement.
- In Section 3.3, we present and evaluate the clean shutdown scheme, which addresses problems of the naïve scheme. We then suggest further avenues for improvement.
- In Section 3.4, we summarize our findings, and preview our next step in reducing downtime.

bridge	initial	target
	start	
move link		enable NIC
		enable BGP
	done	

Figure 3.1: Naïve rehoming procedure. The column for each node lists the steps taken on that node. Gray horizontal bars indicate barriers used to synchronize actions across nodes. Steps between barriers are unordered with respect to actions of other nodes.

3.1 Rehoming Goals

Our primary goal, of course, is to increase Internet reliability in general, and reduce the downtime caused by planned maintenance specifically. However, there are two important subgoals that we highlight here: deployability and usability. To maximize deployability, we will seek minimally intrusive changes to the existing software. And, in particular, we require that our solution not require any modifications to the software on the customer router.

To maximize usability, we require that our solution not require any action on the part of the customer. In particular, establishing the BGP peering session between the customer router and the target router should not require any reconfiguration of the customer router. This both, avoids customer frustration, and limits the possibility of errors. The latter is particularly relevant, given that operator error is a significant source of failures in computer systems [65].

This second goal drives an important design decision. Specifically, we choose to have the target router use the same IP address, on its customer-facing interface, as the initial router uses on its customer-facing interface. In Section 3.2.5, we will identify and resolve a complication that this decision introduces for BGP session establishment.

3.2 Naïve rehoming

In this section, we present experimental results for rehoming with a very simple rehoming procedure, which we name *naïve*. In this procedure, which we illustrate in Figure 3.1, the bridge and the target router simultaneously execute commands to reconfigure layer 2 and layer 3 network elements. Specifically: the bridge reconfigures the layer 2 link between the customer router and the ISP routers, by removing the initial router from the bridge group containing the customer router, and then adding the target router to the same group[1]; the target router reconfigures layer 2 connectivity by enabling its interface to the customer, and reconfigures layer 3 connectivity by enabling its BGP peering with the customer.

[1]By default, a new port in a bridge group starts in learning mode. In this mode, the bridge listens for traffic on the port, but does forward any frames out through the port. In order to avoid the outage this would cause for traffic from the customer to the Internet, we use the `brctl` command to configure the "forwarding delay" on the port to zero.

	restart	naïve
internet to CUST1	58.19	150.24
internet to CUSTN	54.15	150.25
customer to INET1	73.69	65.12
customer to INETN	129.67	64.57
any	143.17	156.43

Table 3.1: Comparison of mean outage times, in seconds, and over ten trials, for router restart and naïve rehoming. Data for router restart are copied from Table 2.5.

3.2.1 High-level comparison to router restart

We compare the outage times for router restart and naïve rehoming in Table 3.1. The results are rather surprising. While the outage times for traffic outbound from the customer to the Internet have improved, the outages times for traffic inbound from the Internet to the customer are significantly longer than before. Additionally, the overall outage times are longer with naïve rehoming than with router restart. To understand why, we again return to our system charts.

3.2.2 In-depth comparison of trials with the shortest overall outage times

We present system charts for the trials of router restart and naïve rehoming with the shortest overall outages times in Figures 3.2 and 3.3, respectively. We first make some general observations about the differences between these two experiments, and then attempt to explain why some of the outage times have increased.

Our general observations are as follows:

1. As desired, eliminating the restart on the initial router has largely eliminated CPU activity on that router.

2. The pattern of CPU utilization exhibited by the `bgpd` process on the remote router, following the re-establishment of the BGP peering between the remote router and the initial router (denoted by **bgp** $\uparrow_i$), is no longer present on the remote router. However, a similar pattern is now seen on the target router, following **bgp** $\uparrow_c$. We hypothesize that this pattern reflects the work done to generate and transmit routing advertisements to the peer router.

3. For rehoming, there is a surprising amount of CPU load due to the `zebra` process on the target router. We will examine this more deeply in Chapter 5.

4. The total CPU load on the customer router appears to be roughly the same between router restart and naïve rehoming. As indicated in the conclusion of Chapter 2, we will address CPU load on the customer router in Chapter 4.

While these observations are useful for understanding the system generally, they do not explain the increase in outage times. To understand the increase in outage times for traffic inbound to the customer, we focus on the Figure 3.3, and note that the outages for both CUST1 and CUSTN end at **timeout**$_c$, the time that the initial router times out its peering with the customer router. It is only following this timeout that the remote router updates its FIB entries for the customer routes, as

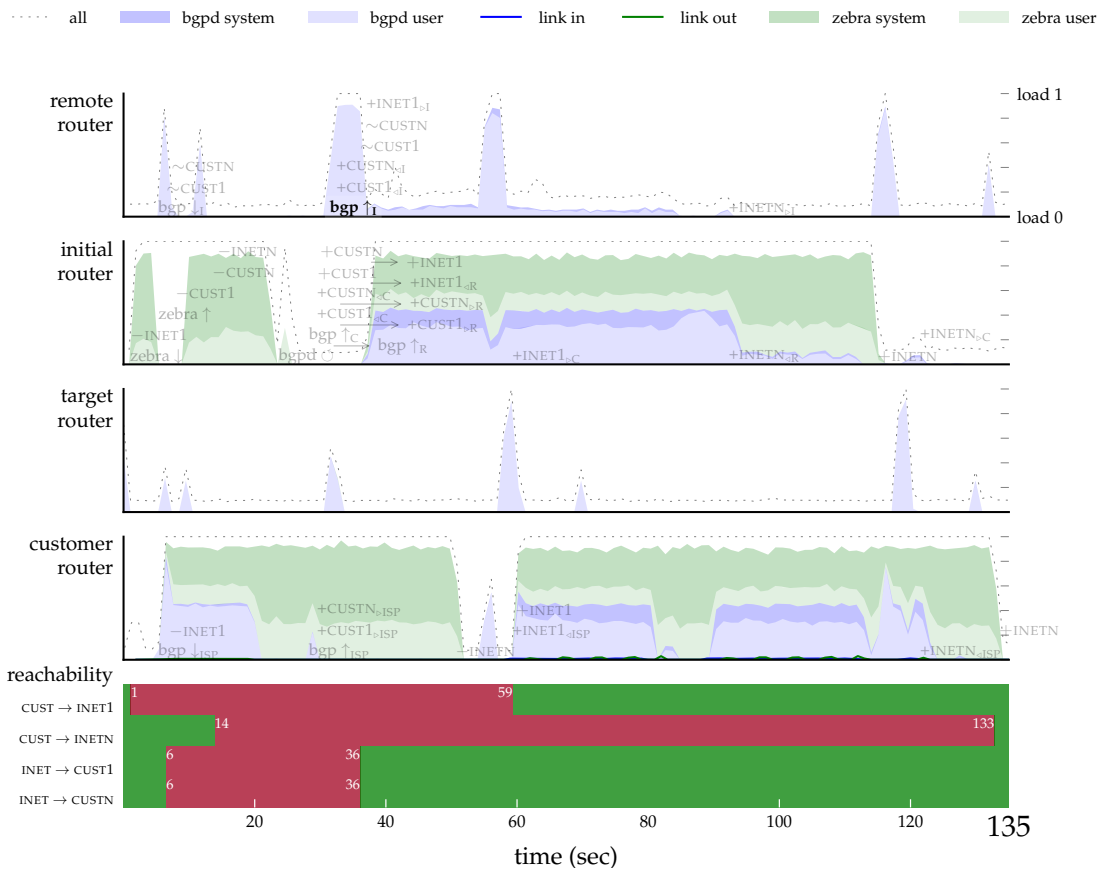


Figure 3.2: Partial system chart for router restart, for the trial with the minimum overall outage time. This chart repeats data from Figure 2.6(c), adding the timeseries for the remote and target routers. The complete system chart is available as Figure A.3.

denoted by the **∼CUST1** and **∼CUSTN**. This is true despite the fact that the fact that the remote router has received new advertisements for CUST1 and CUSTN much earlier, as denoted by **+CUST1**$_{\triangleleft T}$ and **+CUSTN**$_{\triangleleft T}$.

The cause of the delay between the remote router's receipt of advertisements for CUST1 and CUSTN, and its updating of the corresponding FIB entries, is as follows. Until the initial router times out its peering with the customer router, and generates withdrawals for CUST1 and CUSTN[2], the remote router has two routes to the customer prefixes: one through the initial router, and another through the target router. Because the routes have the same path attributes, the remote router selects between them based on the tie-breaking rules in the BGP decision process [69]. Specifically, the remote router selects the route through the initial router, because that router has a lower `router-id` than the target router.

This hypothesis is confirmed by Figure 3.4, in which we configure the initial router to have a

[2]These withdrawals are denoted -CUST1$_{\triangleright R}$ and -CUSTN$_{\triangleright R}$ on the time-series for the initial router, respectively.

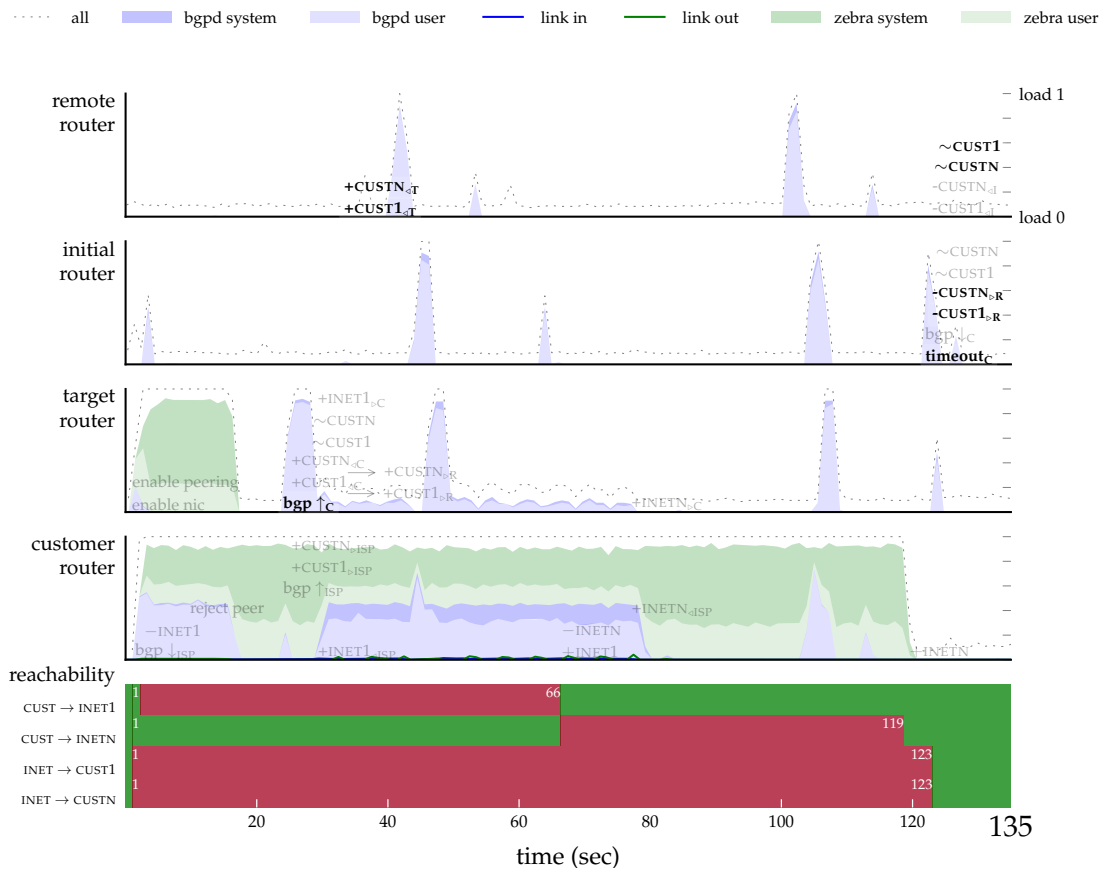


Figure 3.3: Partial system chart for naïve rehoming, for the trial with the minimum overall outage time. Note that the bridge node is omitted, as it does not contribute significantly to the observed behavior. The complete system chart is available as Figure A.6.

higher `router-id` than the target router. With this change in place, the outages for traffic to the customer end as the remote router receives advertisements from the target router, as denoted by **+CUST1**$_{\triangleleft T}$ and **+CUSTN**$_{\triangleleft T}$, rather than being delayed until the remote router has received withdrawals from the initial router.[3]

We see, then, that the ordering of `router-id`s between the initial router and the target router can significantly alter the outage times experienced by traffic inbound to the customer. Returning to Figures 3.2 and 3.3, however, we note that the increase in outage time for inbound traffic does not explain the increase in overall outage time. In fact, focusing on the trials with the shortest outages, the total outage time for naïve rehoming is lower than that for router restart. To understand the increase in mean outage times, we turn instead to the system charts for the trials of router restart

[3]In our initial experiments, we set `router-id`s based on the public-facing IP addresses assigned to nodes by Emulab. After noticing a discrepancy between experiments, we modified our code to always generate the worst case behavior for rehoming, by setting the `router-id` of the initial router to be lower than that of the target router. Except for the experiment illustrated in Figure 3.4, all experiments in this dissertation have a lower `router-id` on the initial router.

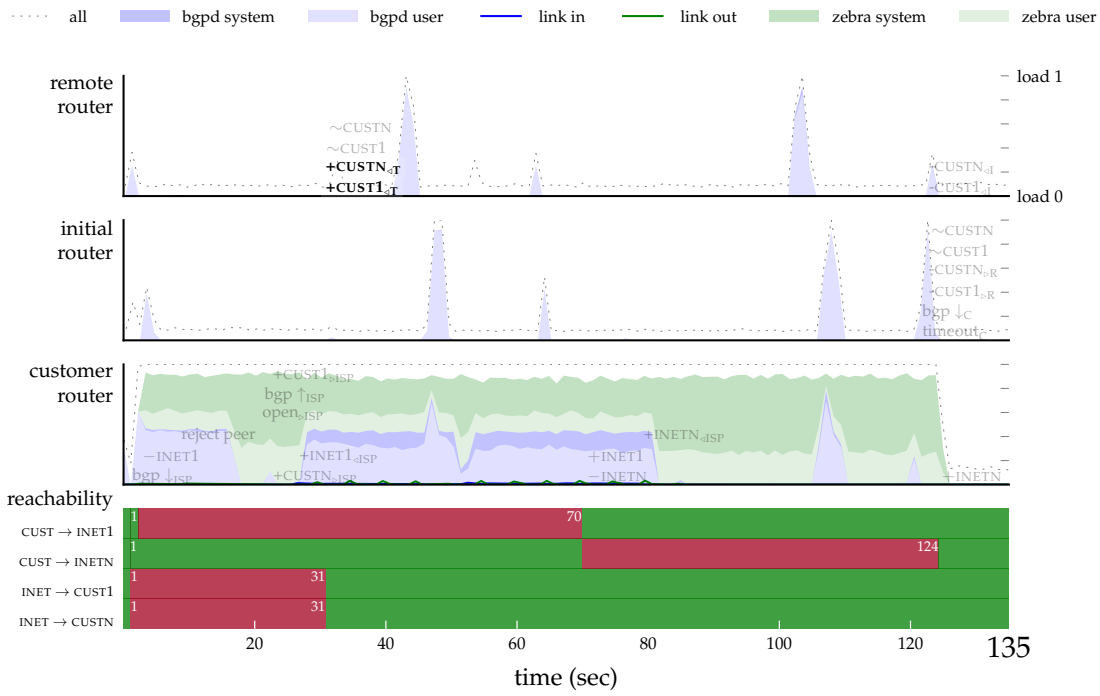


Figure 3.4: Partial system chart for naïve rehoming, with the initial router configured to have a higher router-id than the target router. This chart illustrates the trial with the minimum overall outage time. The full system chart is provided as Figure A.7.

and naïve rehoming with the longest overall outage times.

3.2.3 Comparison of trials with the longest overall outage times

To compare the trials with the longest overall outage times, we present Figure 3.5. Looking at this figure, we make two observations:

1. With naïve rehoming, the outage time for traffic inbound to CUST1 and CUSTN alone, is greater than the total outage time for router restart. Because this outage time is determined in large part by the initial router's timing out of its peering with the customer router, it is important for us to understand this timeout behavior.

2. A significant portion of the increase in outage times is due to delays in the establishment of a new peering session between the ISP router and the customer router. In the router restart experiment, the new session is established approximately 50 seconds after the start of the experiment, as denoted by **bgp** $\uparrow_{\text{ISP}}$. If we could bring the session up earlier, we could likely improve outage times by shifting some of the computation that occurs between time 140 and 215 seconds on the customer into the idle time between 100 seconds and 140 seconds.

Accordingly, we now turn to understanding BGP timeout behavior, and then to understanding session establishment behavior.

3.2.4 BGP timeout behavior

To understand BGP timeout behavior, we compare the trials of naïve rehoming with the shortest and longest outage times, as presented in Figure 3.6. In the trial with the shortest outage time, we observe that the initial router times out the BGP peering with the customer router, as denoted by the **timeout**$_c$ annotation, near time 120 seconds. For the trial with the longest outage time, **timeout**$_c$ occurs at approximately 180 seconds.

What causes this difference in timeout detection? That is, why does the initial router sometimes take 120 seconds to time out its peering with the customer router, but take 180 seconds at other times? The answer lies in the BGP timeout mechanism. Per the BGP specification [69],

> 6.5. Hold Timer Expired Error Handling
>
> If a system does not receive successive KEEPALIVE, UPDATE, and/or NOTIFICATION messages within the period specified in the Hold Time field of the OPEN message, then the NOTIFICATION message with the Hold Timer Expired Error Code is sent and the BGP connection is closed.

Quagga defaults to a Hold Time of 180 seconds, and a KEEPALIVE interval of 60 seconds. In cases where we began rehoming immediately after the customer router transmitted a keepalive, we would expect the initial router to wait approximately 180 seconds to time out its peering with the customer router. Similarly, in the cases where we began rehoming immediately before the customer router was about to transmit its next keepalive (i.e., approximately 60 seconds after the previous keepalive), we would expect the initial router to wait approximately 120 seconds to time out the peering. Because we conduct multiple trials of each experiment, with randomized sleep intervals between trials, we are able to observe this full range of behaviors.

3.2.5 BGP session establishment behavior

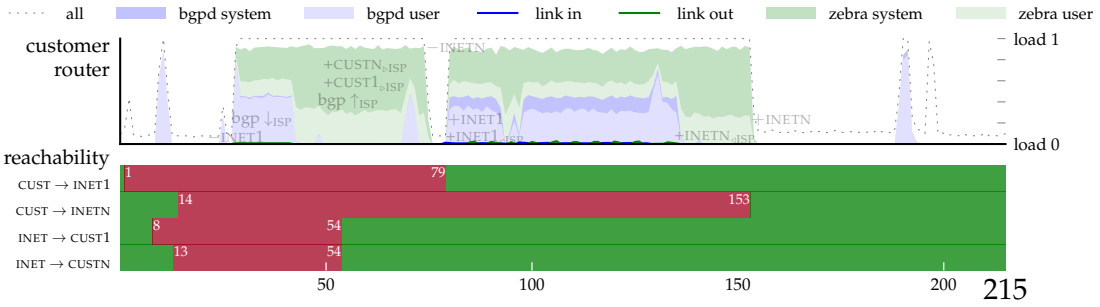
We now turn our attention to the delay in establishing the BGP peering between the target router and the customer router. Focusing on Figure 3.5(b), we note the presence of four **reject peer** annotations on the time-series for the customer router, indicating that the customer router rejected four attempts, by its ISP-side peer, to open a peering session.

The first three of these four rejections are consistent with Section 6.8 of the BGP specification [69], which states:

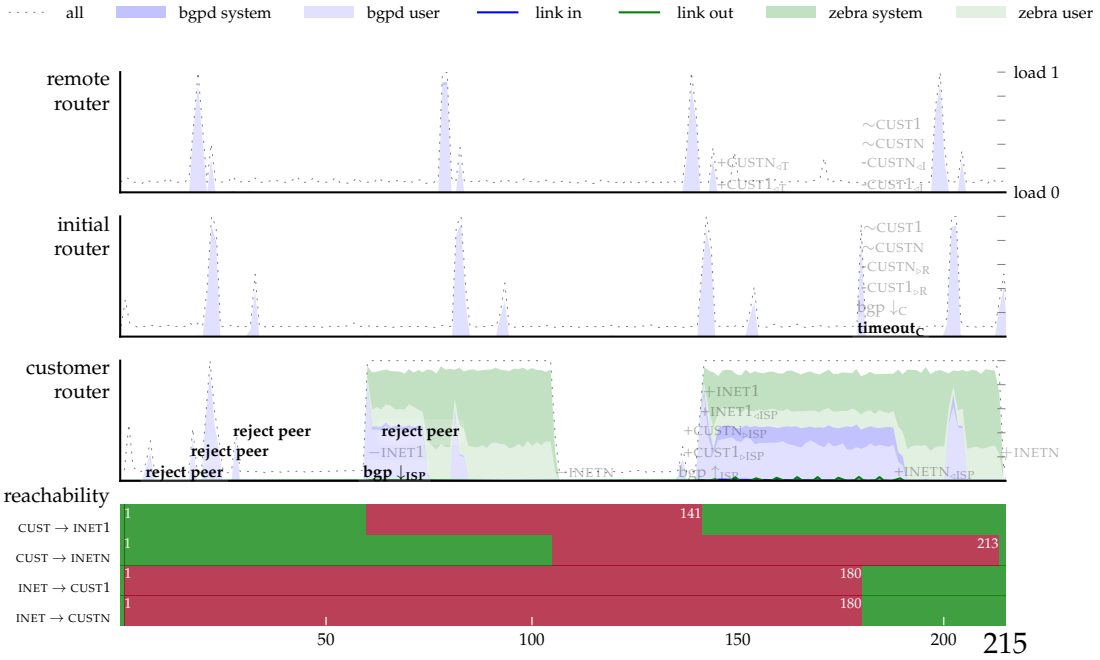
> Unless allowed via configuration, a connection collision with an existing BGP connection that is in the Established state causes closing of the newly created connection.

However, the last of these rejections occurs after **bgp** $\downarrow_{\text{ISP}}$, which indicates that the customer router has taken down its side of the peering with the initial router. To understand this unexpected behavior, we turn our attention to the log file messages generated by `bgpd` on the customer router. Therein, we find the exact text of the logfile messages which were denoted as **reject peer** on the system chart:

```
BGP: %NOTIFICATION: sent to neighbor 10.1.8.2 2/3 (OPEN Message Error/Bad BGP Identifier)
4 bytes c0 a8 00 02
BGP: %NOTIFICATION: sent to neighbor 10.1.8.2 2/3 (OPEN Message Error/Bad BGP Identifier)
4 bytes c0 a8 00 02
```

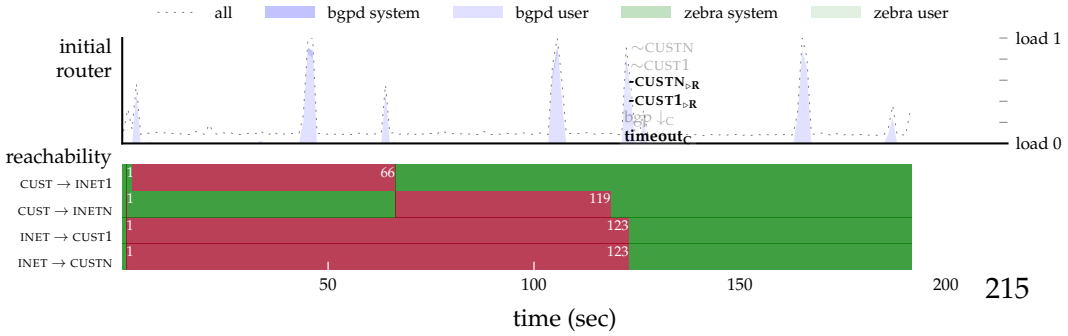


(a) router restart

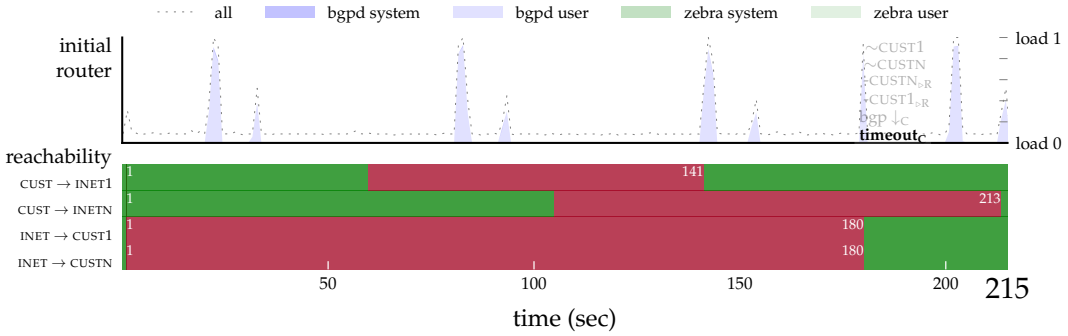


(b) naïve rehoming

Figure 3.5: Partial system charts comparing router restart and naïve rehoming, for the trials with the longest overall outage times. For the complete system charts, see Figures A.8 and A.9.



(a) shortest outage



(b) longest outage

Figure 3.6: Partial system charts comparing the trials of naïve rehoming with the shortest and longest outages. Note that **timeout**$_c$ occurs at about 120 seconds in the case of the short outage, and at about 180 seconds in the case of the longest outage. For the complete system charts, see Figures A.6 and A.9.

```
1  /* remote router-id check. */
2  if (remote_id.s_addr == 0
3      || ntohl (remote_id.s_addr) >= 0xe0000000
4      || ntohl (peer->local_id.s_addr) == ntohl (remote_id.s_addr))
5    {
6      if (BGP_DEBUG (normal, NORMAL))
7        zlog_debug ("%s bad OPEN, wrong router identifier %s",
8                    peer->host, inet_ntoa (remote_id));
9      bgp_notify_send_with_data (peer,
10                                BGP_NOTIFY_OPEN_ERR,
11                                BGP_NOTIFY_OPEN_BAD_BGP_IDENT,
12                                notify_data_remote_id, 4);
13     return -1;
14   }
```

Listing 3.1: Source code that might generate "Bad BGP Identifier" messages, from the function bgp_open_receive in bgp_packet.c.

BGP: %NOTIFICATION: sent to neighbor 10.1.8.2 2/3 (OPEN Message Error/Bad BGP Identifier) 4 bytes c0 a8 00 02
BGP: %NOTIFICATION: sent to neighbor 10.1.8.2 2/3 (OPEN Message Error/Bad BGP Identifier) 4 bytes c0 a8 00 02

To better understand these messages, we examine the source code for bgpd, and identify two segments of source code that might generate these log files messages. These are reproduced in Listings 3.1 and 3.2. The code of Listing 3.1 can not be the source of the Bad BGP Identifier messages in our experiments, because the router-id on the target is 192.168.0.2. This is neither equal to 0.0.0.0 (as checked at line 2), nor greater than 224.0.0.0 (as checked at line 3), nor equal to the customer router's router-id (as checked at line 4).[4]

Instead, based on the comment at line 7 in Listing 3.2, we infer that the customer router is rejecting the peering request from the target router because the target router's router-id differs from that of the initial router. To verify this hypothesis, we patch bgpd on the target router to support spoofing of its router-id on a per-connection basis, with the patch of Listing 3.3.[5] This patch is modeled after existing code in Quagga, which enables the administrator to modify the local AS number on a per-peer basis.

After patching the source code, we configured the target router to use the same router-id as the initial router, for its peering with the customer router.[6] For its other peerings, namely the peerings with the initial router and the remote router, the target router continues to use its own router-id. With this patch in place, we return to the question of why the customer router rejects the target router's peering request even after the customer router has terminated its peering with the initial router.

After confirming that the trial, of this new experiment, with the longest outage time also exhibits

[4] In this experiment, the customer router's router-id was 155.98.36.62.

[5] We choose to modify the router-id on a per-connection basis, because we do not want to change the target router's router-id on its peering with the remote router. Doing so might confuse the remote router, because both the initial router and the target router would have the same router-id.

[6] Note that this patch is applied only on the target router.

```
int as = 0;

realpeer = peer_lookup_with_open (&peer->su, remote_as, &remote_id, &as);

if (! realpeer)
  {
    /* Peer's source IP address is check in bgp_accept(), so this
       must be AS number mismatch or remote-id configuration
       mismatch. */
    if (as)
      {
        if (BGP_DEBUG (normal, NORMAL))
          zlog_debug ("%s bad OPEN, wrong router identifier %s",
                      peer->host, inet_ntoa (remote_id));
        bgp_notify_send_with_data (peer, BGP_NOTIFY_OPEN_ERR,
                                   BGP_NOTIFY_OPEN_BAD_BGP_IDENT,
                                   notify_data_remote_id, 4);
      }
    else
      {
        if (BGP_DEBUG (normal, NORMAL))
          zlog_debug ("%s bad OPEN, remote AS is %u, expected %u",
                      peer->host, remote_as, peer->as);
        bgp_notify_send_with_data (peer, BGP_NOTIFY_OPEN_ERR,
                                   BGP_NOTIFY_OPEN_BAD_PEER_AS,
                                   notify_data_remote_as, 2);
      }
    return -1;
  }
```

Listing 3.2: Source code that generates the "Bad BGP Identifier" messages observed in our experiments. This code is from the function bgp_open_receive in bgp_packet.c.

```
  stream_putw (s, (local_as <= BGP_AS_MAX) ? (u_int16_t) local_as
                                           : BGP_AS_TRANS);
  stream_putw (s, send_holdtime);              /* Hold Time */
  stream_put_in_addr (s, &peer->local_id); /* BGP Identifier */
  if (peer->change_local_id.s_addr)      /* BGP Identifier */
    stream_put_in_addr (s, &peer->change_local_id);
  else
    stream_put_in_addr (s, &peer->local_id);

  /* Set capability code. */
  bgp_open_capability (s, peer);
```

Listing 3.3: Core source code for patch to enable router-id spoofing. This portion of the patch modifies the function bgp_open_send in bgp_packet.c. Unmodified lines are shown in gray, removed lines are shown in gray with a horizontal bar, and added lines are shown in bold. Modified lines are indicated by a deletion followed by an addition. The complete patch is provided as Listings B.1 through B.10.

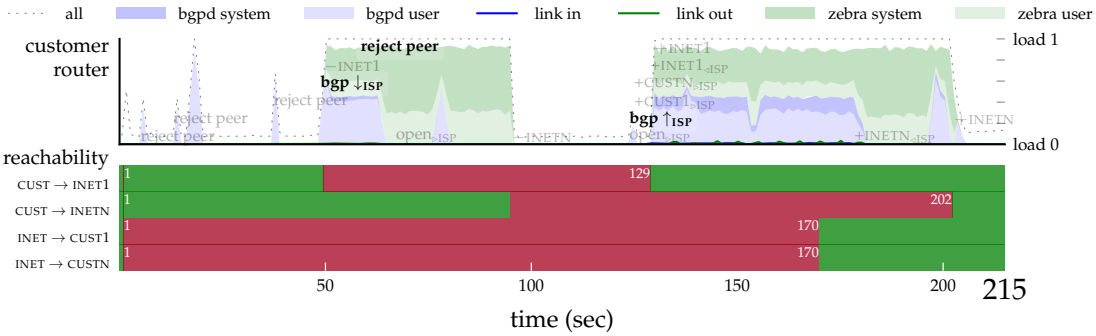


Figure 3.7: Partial system chart for naïve rehoming with router-id spoofing, for the trial with the longest overall outage time. For the complete system chart, see Figure A.10.

a **reject peer** after **bgp** $\downarrow_{ISP}$ (see Figure 3.7), we examine the log file messages for bgpd on the customer router. Therein, we find the following messages:

> BGP: %NOTIFICATION: sent to neighbor 10.1.8.2 6/5 (Cease/Connection Rejected) 0 bytes
> BGP: %NOTIFICATION: sent to neighbor 10.1.8.2 6/5 (Cease/Connection Rejected) 0 bytes
> BGP: %NOTIFICATION: sent to neighbor 10.1.8.2 6/5 (Cease/Connection Rejected) 0 bytes
> BGP: %NOTIFICATION: sent to neighbor 10.1.8.2 6/5 (Cease/Connection Rejected) 0 bytes

Unfortunately, these messages fail to explain why the customer router rejects the fourth connection attempt. In order to gather more data, we re-run the experiment, with the "debug bgp" and "debug bgp events" options enabled for bgpd on the customer router. Examining the log files for that experiment, we find two additional messages, which explain the rejection of the fourth connection attempt:

> BGP: 10.1.8.2 peer status is Established close connection
> BGP: 10.1.8.2 peer status is Clearing close connection

Of the 33 "Connection Rejected" messages logged over the ten trials of this experiment, 23 were preceded by the "peer status is Established" message, while 10 were preceded by the "peer status is Clearing" message. Based on this data, and the observation that the fourth connection attempts in Figures 3.5(b) and 3.7 occur while `bgpd` is busy on the CPU, we conclude that the fourth **reject peer** event is due to `bgpd` refusing a new connection until it has finished clearing out the state from the old connection.[7]

3.2.6 Avenues for improvement

Based on the above analysis, we conclude that naïve rehoming, which does not explicitly tear down the BGP peering between the initial router and the customer router, prolongs outages due to the time taken by the initial router and the customer router to detect the peering failure.

On the initial router, the delay in the detection of peering loss means that the initial router does not withdraw its routes for the customer prefixes until it (the initial router) has timed out its peering with the customer router. This, in turn, means that the remote router continues to route traffic for the customer through the initial router.

On the customer router, the delay in peering loss detection increases the time taken to establish a BGP peering between the customer router and the target router. This, in turn, delays the receipt and processing of route advertisements from the target router, prolonging the outage for traffic from the customer to the Internet.

An obvious way to eliminate these problems is to explicitly shut down the peering between the customer router and the initial router during rehoming. We now turn our attention to this idea, which we call "clean shutdown".

3.3 Clean shutdown rehoming

As with the naïve rehoming procedure, we begin our presentation by detailing the procedure itself. We then present an overview of the experimental results, followed by an in-depth analysis of the experiments. Finally, we conclude the section with a discussion of avenues for improvement.

3.3.1 Rehoming procedure

We illustrate the clean shutdown rehoming procedure in Figure 3.8. As compared to the naïve rehoming procedure, there are two differences. First is that we explicitly disable the peering session on the initial router before reconfiguring the layer two link. The second is that we explicitly disable the customer-facing interface on the initial router.

Note that after disabling the peering, we wait for `bgpd` to issue another command prompt. We do this to give `bgpd` the opportunity to complete the termination of the peering session. In particular, we do not want to reconfigure layer two connectivity before the initial router has transmitted the BGP NOTIFICATION CEASE message. Were we to reconfigure the layer two link before the transmission of the NOTIFICATION CEASE message, the customer router will not know the session had been terminated.

[7] Clearing out the old state would include, for example, deleting the $\approx 300,000$ routes advertised by the initial router from `bgpd`'s RIB, and instructing `zebra` that these routes are no longer available.

bridge	initial	target
	start	
	disable peering	enable NIC
	wait for prompt	enable peering
	peering terminated	
move link	disable NIC	
	done	

Figure 3.8: Clean shutdown rehoming procedure. The column for each node lists the steps taken on that node. Gray horizontal bars indicate barriers used to synchronize actions across nodes; thin lines delimit multiple steps on a single node. Steps between barriers are unordered with respect to actions of other nodes.

	router restart	naïve	clean shutdown
internet to CUST1	58.19	150.24	37.52
internet to CUSTN	54.15	150.25	37.50
customer to INET1	73.69	65.12	58.82
customer to INETN	129.67	64.57	61.61
any	143.17	156.43	120.20

Table 3.2: Comparison of mean outage times, in seconds, and over ten trials, for router restart, naïve rehoming and clean shutdown rehoming. Data for router restart and naïve rehoming are copied from Table 3.1.

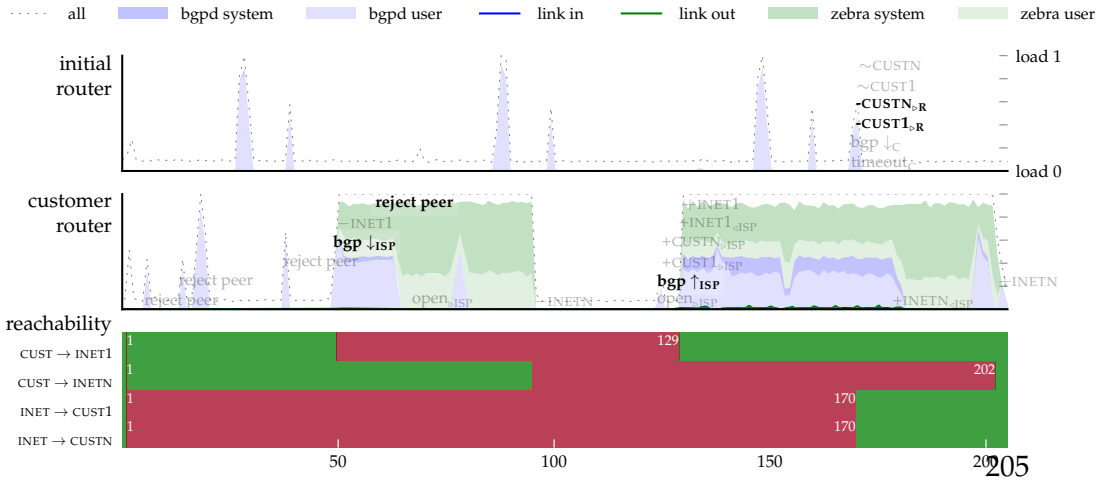
3.3.2 Empirical results

We summarize the empirical results in Table 3.2. From the data in the table, we see that clean shutdown rehoming performs much better than naïve rehoming for traffic to CUST1 and CUSTN, and much better than router restart for traffic to INETN, with improvements of 75% and 52%, respectively. Differences in overall outage times are less dramatic, with clean shutdown providing a 16% improvement over router restart, and a 23% improvement over naïve rehoming.

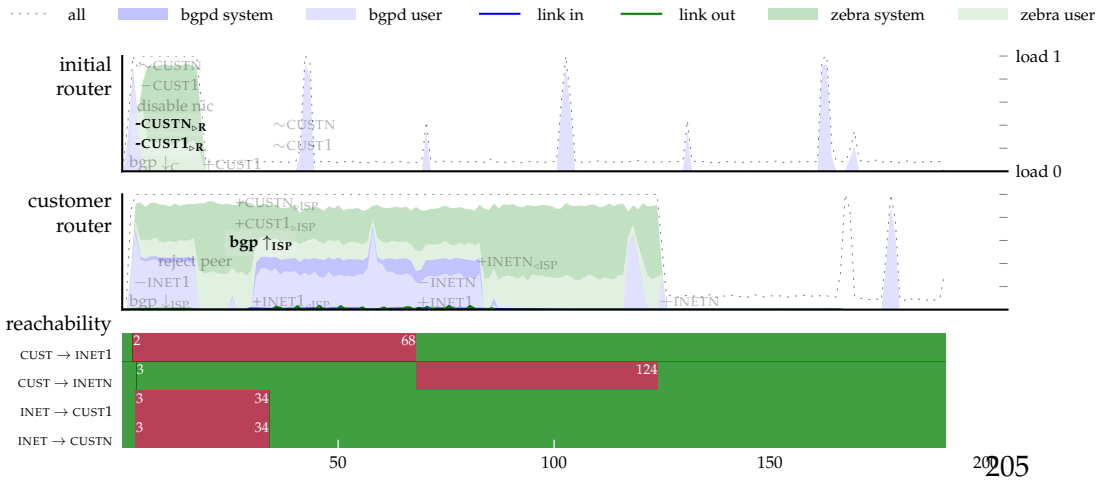
To verify that clean shutdown addresses the issues identified in Section 3.2.6, we compare the system charts for naïve rehoming and clean shutdown rehoming. We present these charts in Figure 3.9. We first observe that the **-CUST1**$_{\triangleright R}$ and **-CUSTN**$_{\triangleright R}$ events on the initial router occur much earlier with clean shutdown, demonstrating that clean shutdown addresses the route withdrawal delay on the initial router. We then observe that the **bgp** $\uparrow_{ISP}$ event on the customer router occurs much earlier with clean shutdown, demonstrating that clean shutdown addresses the session establishment delay.

3.3.3 Avenues for improvement

Although the clean shutdown procedure provides some improvement over the naïve procedure, there is still significant room for improvement. If, for example, an ISP were to target "five nines" reliability, a single rehoming would consume over one third of the annual outage budget.



(a) naïve rehoming with router-id spoofing



(b) clean shutdown rehoming

Figure 3.9: Partial system charts comparing naïve rehoming with router-id spoofing, and clean shutdown rehoming, for the trials with the longest overall outage times. Events referenced from the text are shown in bold. For the complete system charts, see Figures A.10 and A.11.

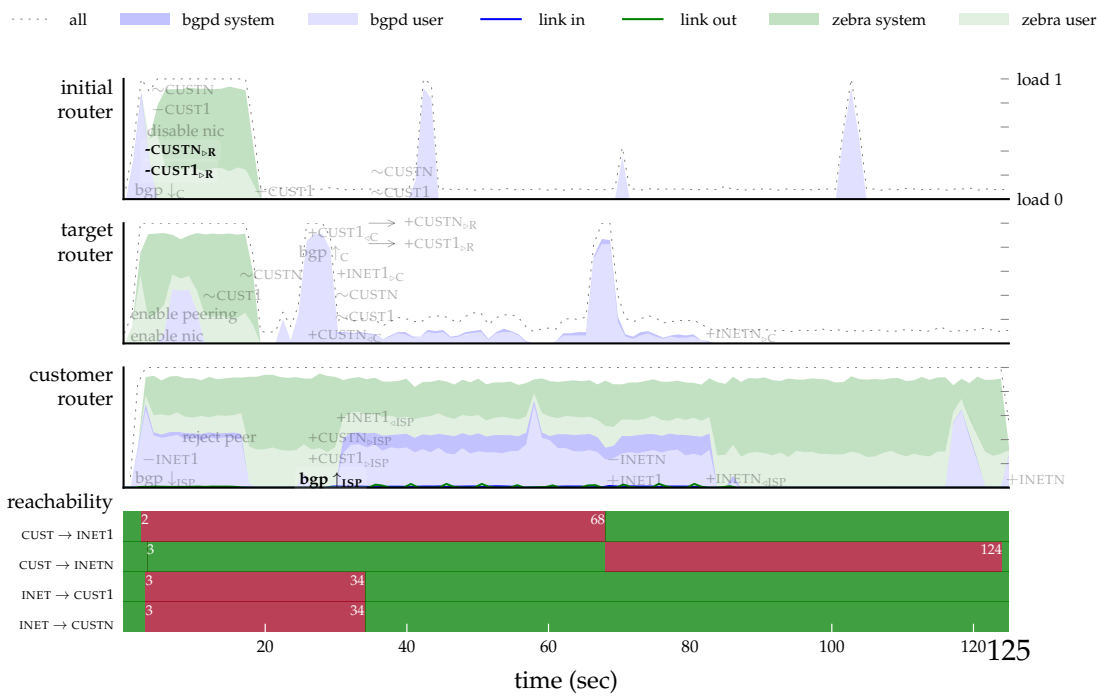


Figure 3.10: Partial system chart for clean shutdown rehoming, for the trial with the longest outage time. The complete system chart is provided as Figure A.11. Note that the variance in outage times for this experiment is low, with a minimum of 118.92 seconds and a maximum of 121.58 seconds. Thus the outage time for this trial is only slightly larger than the mean outage time of 120.20 seconds, given in Table 3.2.

Accordingly, we consult a partial system chart of clean rehoming, Figure 3.10, to identify the avenues for improvement. From this figure, we see that we have achieved our goal of eliminating the initial router as a bottleneck in the rehoming process. Substantially all of the outage time can be explained by CPU utilization at the customer router.

An obvious approach for further improvements, would, then, be to optimize the routing software for this use case. For example, the figure shows that a significant fraction of the CPU time is spent by `zebra` running kernel functions. This is likely due to calls to add and remove routes from the kernel's forwarding information base. Accordingly, it might be possible to reduce outage time by optimizing the user/kernel interface for route manipulation.

We might, however, be able to do better. We observe that much of the work that the customer router is doing is not strictly necessary. Assuming that the initial router and target router have approximately the same routing table, the customer router spends all of its time removing routes, only to add the same routes back shortly thereafter. If we could instruct the customer router to retain its routes during rehoming, we could completely eliminate the cost of route manipulation. Moreover, such an optimization would not depend on a specific user/kernel interface.[8]

3.4 Conclusion

In this chapter, we set out to eliminate ISP-side computation as a source of downtime during planned maintenance on an access router. We proposed a simple scheme, called naïve rehoming, and found that it met this goal. Unfortunately, however, it introduced complications related to BGP session timeout behavior, and BGP session establishment behavior. As a result, naïve rehoming results in longer outages than simple router restart.

To address this problem, we devised a second rehoming scheme, called clean shutdown. This scheme coordinates the actions amongst the routers, and ensures that the BGP peering between the initial router and the customer router is terminated before the link is migrated. Consequently, it avoids the session timeout and session establishment complications observed with the naïve scheme. Thus, clean shutdown both eliminates the ISP-side bottleneck, and results in a net reduction in outage time, as compared to router restart.

Following the elimination of the ISP-side bottleneck, we noted that nearly all of the remaining outage time could be accounted for by computation on the customer router. We observed that while we might tackle this bottleneck by optimizing the kernel's FIB updates, it is possible that we could eliminate the need for the FIB updates entirely. To do so, we would need a mechanism for instructing the customer router to retain the routes learned from the initial router, until the rehoming process was complete.

As it turns out, there is an existing mechanism for instructing a BGP peer to retain routes across a peering failure. This mechanism is known as Graceful Restart [73]. Accordingly, in the next chapter, we turn our attention to Graceful Restart, and how it can help reduce down time during rehoming.

[8] Or, to encompass hardware routers, a specific control plane/forwarding plane interface.

Waste not, want not.

4 Graceful Rehoming

Based on our experiments in Chapter 3, we know that rehoming can eliminate ISP-side computation as the dominant source of downtime during planned maintenance. Specifically, we saw, in Figure 3.10, that the remaining downtime can be explained in terms of computation on the customer router. We then suggested, in Section 3.4, that we could eliminate this bottleneck by instructing the customer router to retain the routes learned from the initial router, during the rehoming process. We further noted that there is an existing mechanism, called Graceful Restart [73], which may be useful in this endeavor.

Accordingly, in this chapter, we evaluate the combination of Graceful Restart and rehoming, which we call Graceful Rehoming. Before we begin, we note a potential pitfall in our use of Graceful Restart. Namely, while Graceful Restart was designed for restoring a peering to the same router, we seek to restore the peering on to a different router. Thus, complications may arise. For example, can we be sure that the customer router will reach a routing state that is consistent with the ISP, even if there are inconsistencies between the RIBs on the initial router?

The remainder of this chapter is structured as follows:

- In Section 4.1, we explain the Graceful Restart facility, give an example of how it functions in normal operation, and then describe how it can be used to safely restore a peering session onto a different router.
- In Section 4.2, we explain configuration details about our use of Graceful Restart, and discuss some of the issues we found in Quagga's implementation of the feature.
- In Section 4.3, we evaluate the use of Graceful Restart with clean shutdown. We show that clean shutdown is, unfortunately, incompatible with Graceful Restart.
- In Section 4.4, we evaluate the use of Graceful Restart with naïve rehoming and router-id spoofing. We find that this configuration yields a substantial improvement in outage times for traffic from the customer to the Internet. Unfortunately, however, outage times for traffic from the Internet to the customer remain an issue.

- In Section 4.5, we improve the outage times for traffic to the customer by using BGP's LOCAL_PREF attribute to accelerate the remote router's selection of routes through the target router.
- In Section 4.6, we consider an alternative solution to the incompatibility discovered in Section 4.3, and argue for the advantages of the solution developed in Sections 4.4 and 4.5.
- In Section 4.7, we consider our improvements in the context of the overall reliability goal, and identify opportunities for further improvement.
- In Section 4.8, we summarize and conclude.

4.1 Introduction to Graceful Restart

To explain Graceful Restart, we first give describe the roles of BGP peers during Graceful Restart, and the mechanisms that Graceful Restart provides to minimize disruption during a peering loss. After the overview, we walk through an example of Graceful Restart in normal operation, explain how it can be used to safely reduce down time during rehoming, and then illustrate the use of Graceful Restart in the rehoming process.

4.1.1 Overview of Graceful Restart

At a high level, Graceful Restart defines two roles, and provides four key signaling mechanisms. We begin with the roles, as they are necessary to explain how the mechanisms are implemented. There are two roles in Graceful Restart. Each router is either a *restarting router*, or a *receiving router*. As expected, a router is a restarting router if it is recovering from a crash, and a receiving router otherwise.

In addition to these two roles, Graceful Restart defines four mechanisms. These mechanisms are used at peering session establishment, and only at the time the session is established. We first enumerate the mechanisms, and then explain how they are used.

The four mechanisms are 1) a way for a router to signal to a peer that the peer should retain advertised routes if the peering fails, 2) a way to signal whether or not the router retained its state, relative to the prior peering, 3) a way to signal that the router intends to wait for routing information from peers, before sending its own routing updates, and 4) a way to signal that a router has completed transmission of its current routing state.

Regarding the first mechanism, when should a router ask its peer to retain routes across a failure? Because this signaling is done at the start of a peering session, the cause of the peering failure cannot be anticipated. Thus, it is typically left as a configuration option whether or not a router should ask its peers to retain routes.

The second mechanism, which enables a router to inform its peer whether or not state was preserved across the peering loss, comes into play when the peer is functioning as a receiving router. In this case, the signal determines when the receiving router deletes routes learned from the prior peering (referred to hereafter as *stale routes*). If the state was not preserved, the receiving router will delete stale routes immediately as the new peering session is established. Otherwise, the receiving router will wait until the restarting router has transmitted its state.

The third mechanism and fourth mechanisms are the most subtle of the set. The key idea behind these two mechanisms is that a restarting router is likely to reconsider many of its routing decisions as it learns routes from its multiple peers. If the router were to transmit its routing information

immediately, many routes would be advertised repeatedly, as the router's decisions changed. To reduce route churn, the router instead waits until all peers have signaled that they have completed the transmission of their routing information.[1]

Note that, in general, a router may play either role during Graceful Restart, without regard to the role played by the other router. For example, if a peering loss were due to a transient connectivity failure, rather than a router reboot, both peers would assume the role of receiving router. Similarly, if two routers rebooted simultaneously, both routers would assume the role of restarting router.

4.1.2 Graceful Restart in Practice

Having given an overview of Graceful Restart, we now describe Graceful Restart in normal operation, explain how it can be safely used for rehoming, and then illustrate the use of Graceful Restart during rehoming.

Normal Operation

We now work through an example of Graceful Restart in normal operation, as depicted in Figure 4.1. To aid the understanding of the state changes illustrated in that figure, we also provide flowcharts of the operation of a restarting router and a receiving router, in Figures 4.3 and 4.4 respectively.[2]

For ease of explication, we consider a system of only two routers. Accordingly, the routers peer with each other, but not with any other routers. We assume that the restart process completes without exception. This means, for example, that the timers illustrated in the flowcharts due not expire.

At the beginning of our example, the peering between the restarting router (router A) and the receiving router (router B) has been established, the peers have indicated to each other that routes should be retained across any failures, and the peers have transmitted their routes to each other. The local state on each router is as depicted in step 1 of Figure 4.1.

Next, at step 2, router A is configured to filter advertisements for INET1. Accordingly, A deletes INET1 from its RIB. However, before it can send the corresponding BGP UPDATE to router B, router A fails. Some time later, router B detects the failure of router A.[3] Upon detecting the failed peering, router B retains the routes advertised by router A, but marks them as stale. The system state is now as illustrated at step 3.

Subsequently, router A restarts, and the two routers reestablish their peering. Router A indicates that it has retained its routing state, and that, as the restarting router, it will wait for updates from router B, before sending its own routing information. Router B indicates that it has also retained its routing state (this is trivially true, as the router did not crash), but that, as the receiving router, it will not wait for updates from router A. This yields the system state depicted at step 4.

At this point, step 5, router B transmits its routing information base, or RIB, to router A. Because router B's RIB included CUST1 and CUSTN, router A no longer considers these prefixes stale. Next, at step 6, router B sends router A an "end-of-RIB marker", to indicate that it has finished sending all of its routing information.

[1] Of course, if both routes were to behave this way, the peering session would deadlock. Thus, if a peer has signaled its intention to wait for routes from others, others will not wait for its routes.

[2] The flowcharts illustrate the most relevant portions of Graceful Restart. For a complete specification, we refer the reader to [73].

[3] The mechanism for failure detection is outside the scope of the Graceful Restart specification.

Having received the end-of-RIB marker from all of its peers, router A proceeds to transmit its own RIB to router B. As router B receives advertisements from router A, it removes the stale marking from INETN. These actions are illustrated as step 7. Finally, in the last step, step 8, router A sends its end-of-RIB marker, causing router B to delete the router for INETN.

Safety of Graceful Rehoming

From the preceding example, we see that Graceful Restart can be used to safely achieve our goals. Namely, because routes are not removed due to the peering loss, we can use Graceful Restart to eliminate the time that `zebra` spends removing routes from, and adding routes to, the kernel's FIB. Also, because stale routes are removed at the end of the Graceful Restart process, we can be confident that the customer router will have correct routing state at the end of rehoming, even if the routing information at the initial and target routers differ, or if routing state changes while rehoming is in progress.

Illustration of Graceful Rehoming

We illustrate the operation of Graceful Rehoming in Figure 4.2. In this example, we assume an inconsistency between the initial router, and the target router. Despite the inconsistency, however, the customer router converges to a RIB which correctly reflects the routing state of the target router.

Because this example is similar to illustration of Graceful Restart, we do not belabor the steps here. However, we highlight three important steps as follows: at step 3, traffic destined to the customer prefixes can be delivered properly; at step 6, traffic from the customer to INET1 will no longer be drawn in to a "blackhole"; and at step 8, other customers of the initial router can again reach the customer being rehomed.

4.2 Experiment Setup

In order to conduct our experiments with Graceful Restart, we needed to make some modifications to Quagga. We also needed to decide the value of the forwarding state parameter in the Graceful Restart capability advertisement. We detail these changes, and our choice for the forwarding state parameter, herein.

4.2.1 Code Modifications

Note that, while Quagga 0.99.16 can be configured to recognize and support a request from a peer to retain advertised routes across a peering failure, it does not itself request that its peers do the same. Additionally, during the course of earlier experiments, we found several defects in Quagga's support for Graceful Restart. To address these issues, we have modified Quagga as follows:

- We added the ability for Quagga to request that its peers retain routes across a peering failure. (See Listing B.11.)
- We fixed a defect in Quagga's parsing of Graceful Restart capability advertisements. (See Listing B.12.)

	restarting peer (A)		receiving peer (B)	
step	RIB-in B	loc-RIB	RIB-in A	loc-RIB
1. start	CUST1 CUSTN	INET1 INETN CUST1 CUSTN	INET1 INETN	INET1 INETN CUST1 CUSTN
2. A configured to filter INET1	CUST1 CUSTN	~~INET1~~ INETN CUST1 CUSTN	INET1 INETN	INET1 INETN CUST1 CUSTN
3. A fails	<down>	<down>	<down>	INET1* INETN* CUST1 CUSTN
4. A restarts, and peering is restored	<empty>	INETN CUST1* CUSTN*	<empty>	INET1* INETN* CUST1 CUSTN
5. B sends RIB	CUST1 CUSTN	INETN CUST1 CUSTN	<empty>	INET1* INETN* CUST1 CUSTN
6. B sends end-of-RIB	CUST1 CUSTN	INETN CUST1 CUSTN	<empty>	INET1* INETN* CUST1 CUSTN
7. A sends RIB	CUST1 CUSTN	INETN CUST1 CUSTN	INETN	INET1* INETN CUST1 CUSTN
8. A sends end-of-RIB	CUST1 CUSTN	INETN CUST1 CUSTN	INETN	~~INET1~~ INETN CUST1 CUSTN

Figure 4.1: Example of Graceful Restart in operation. The "RIB-in" contains routes received from the identified peer. The "loc-RIB" contains routes that have been selected for use by this router. These routes may have been originated locally, or learned from peers. Asterisks denote stale routes, and gray backgrounds highlight changes between steps.

	initial router (I)		customer router (C)		target router (T)	
step	RIB-in C	loc-RIB	RIB-in ISP	loc-RIB	RIB-in C	loc-RIB
1. start	CUST1 CUSTN	INET1 INETN CUST1-I CUSTN-I	INET1 INETN	INET1 INETN CUST1 CUSTN	<down>	INETN CUST1-I CUSTN-I
2. T opens peering with C	CUST1 CUSTN	INET1 INETN CUST1-I CUSTN-I	<empty>	INET1* INETN* CUST1 CUSTN	<empty>	INETN CUST1-I CUSTN-I
3. C sends RIB	CUST1 CUSTN	INET1 INETN CUST1-I CUSTN-I	<empty>	INET1* INETN* CUST1 CUSTN	CUST1 CUSTN	INETN CUST1-T CUSTN-T
4. C sends end-of-RIB	CUST1 CUSTN	INET1 INETN CUST1-I CUSTN-I	<empty>	INET1* INETN* CUST1 CUSTN	CUST1 CUSTN	INETN CUST1-T CUSTN-T
5. T sends RIB	CUST1 CUSTN	INET1 INETN CUST1-I CUSTN-I	INETN	INET1* INETN CUST1 CUSTN	CUST1 CUSTN	INETN CUST1-T CUSTN-T
6. T sends end-of-RIB	CUST1 CUSTN	INET1 INETN CUST1-I CUSTN-I	INETN	~~INET1~~ INETN CUST1 CUSTN	CUST1 CUSTN	INETN CUST1-T CUSTN-T
7. I detects failed peering	<empty>	INET1 INETN CUST1-I* CUSTN-I*	INETN	INETN CUST1 CUSTN	CUST1 CUSTN	INETN CUST1-T CUSTN-T
8. Restart Time or Stale Route Time expires at I	<empty>	INET1 INETN CUST1-T CUSTN-T	INETN	INETN CUST1 CUSTN	CUST1 CUSTN	INETN CUST1-T CUSTN-T

Figure 4.2: Example of Graceful Restart applied to rehoming. Where present, suffixes on the loc-RIB entries denote the next-hop router for that prefix. Note that the target router initially learns of the customer routes through a peering with the initial router. The RIB-in columns for this peering are omitted due to space. Further note that the total order depicted in this figure is for illustrative purposes. Relative ordering of events on different nodes may differ. For example, while step 4 necessary follows step 3, step 5 might precede step 3.

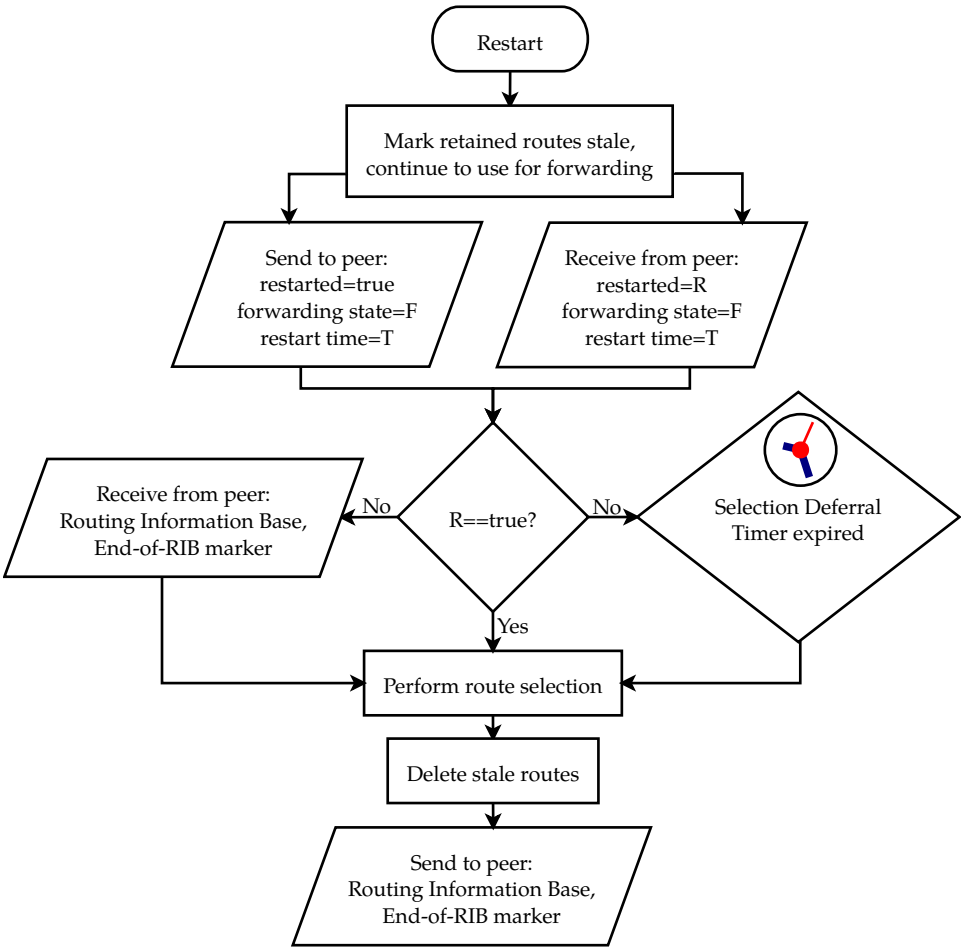


Figure 4.3: Flowchart for a restarting router during Graceful Restart. The clock indicates processing that may occur due to a timer-driven event. R, F, and T denote variables that depend on the state of the router sending the Graceful Restart capability advertisement.

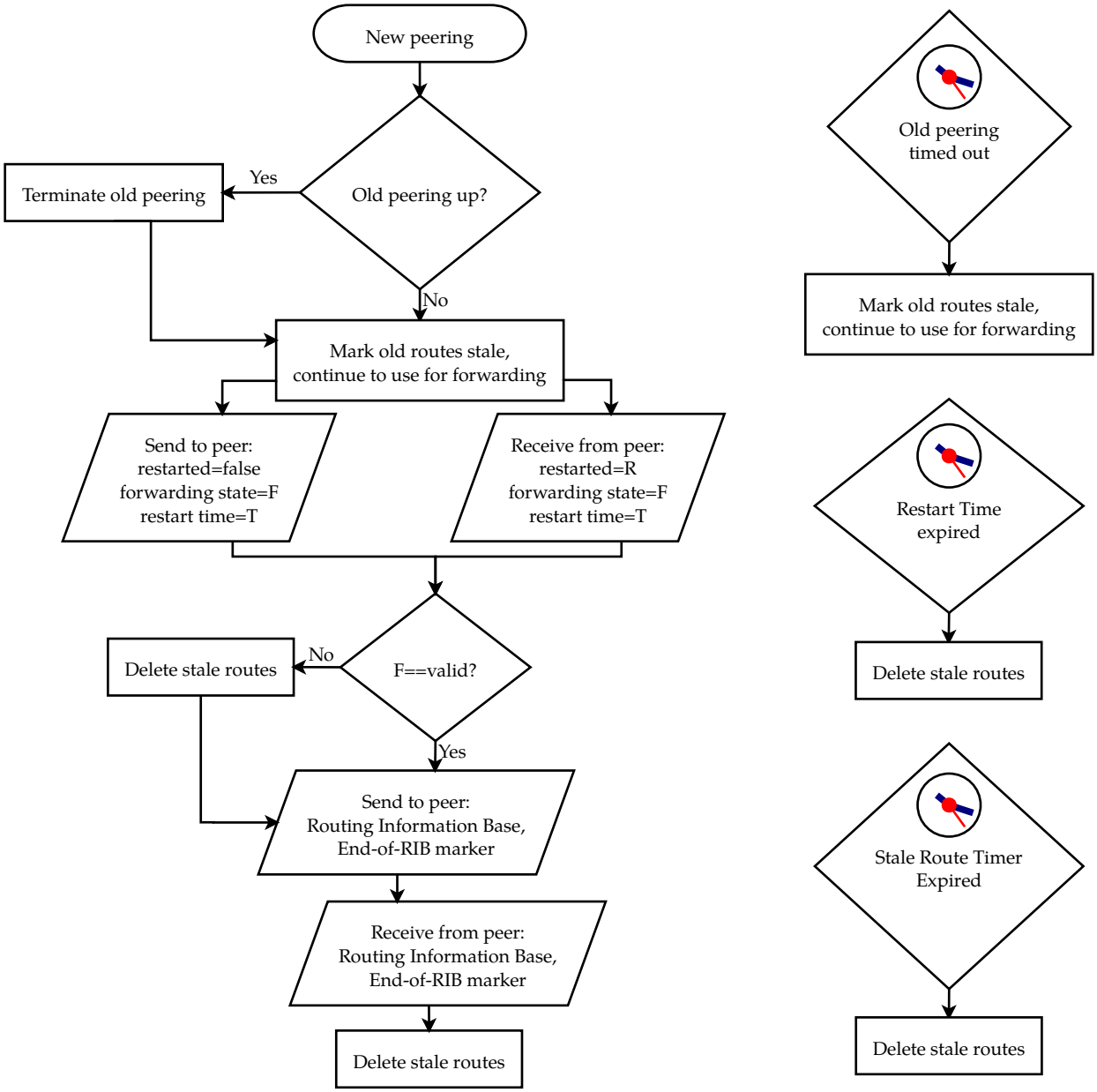


Figure 4.4: Flowchart for a receiving router during Graceful Restart. Clocks indicate processing that may occur due to timer-driven events. R, F, and T denote variables that depend on the state of the router sending the Graceful Restart capability advertisement.

	clean shutdown	clean shutdown + spoofing + graceful
internet to CUST1	37.52	30.85
internet to CUSTN	37.50	30.90
customer to INET1	58.82	54.08
customer to INETN	61.61	63.16
any	120.20	117.30

Table 4.1: Comparison of mean outage times, in seconds, and over ten trials, for clean shutdown, and clean shutdown with router-id spoofing and graceful restart. Data for graceful restart are copied from Table 3.2.

- We fixed a defect in the handling of new peering requests that arrive while an old session is still active. Our change makes Quagga compliant with Section 4.2 of the Graceful Restart specification [73]. (See Listing B.13.)
- We resolved a race condition in Quagga's code for marking routes as stale. (See Listing B.14.)
- We resolved a problem in Quagga's code for deleting stale routes. (See Listing B.15.)

All of our results with Graceful Restart include these modifications. Note that these modifications are applied to all the routers in the experiment, including the customer router. However, with a more complete, and more throughly tested, implementation of Graceful Restart, no changes would be necessary on the customer router.

4.2.2 Forwarding State Parameter

One question that arises in our use of Graceful Restart is whether or not the target router should indicate that state was retained. Strictly speaking, we cannot be sure that the initial router and target router have consistent state. So we might be inclined to indicate that state was not retained. Doing so, though, would cause a receiving router to delete the routes from the previous peering, prior to receiving routes on the new peering. This argues for indicating that state was retained.

Fortunately, as indicated in the flowcharts of Figures 4.3 and 4.4, a peer will always delete stale routes at the end of the Graceful Restart process. Consequently, we can safely signal that routes were retained, without fear of introducing persistent routing inconsistencies. Thus, we set that forwarding state parameter to indicate that forwarding state was retained.

4.3 Graceful Restart with clean shutdown

We present the results of rehoming with Graceful Restart and clean shutdown in Table 4.1. Surprisingly, the data show that adding Graceful Restart to the clean shutdown rehoming procedure provides little benefit. To investigate, we consult Figure 4.5, a partial system chart for the trial with the lowest outage time.

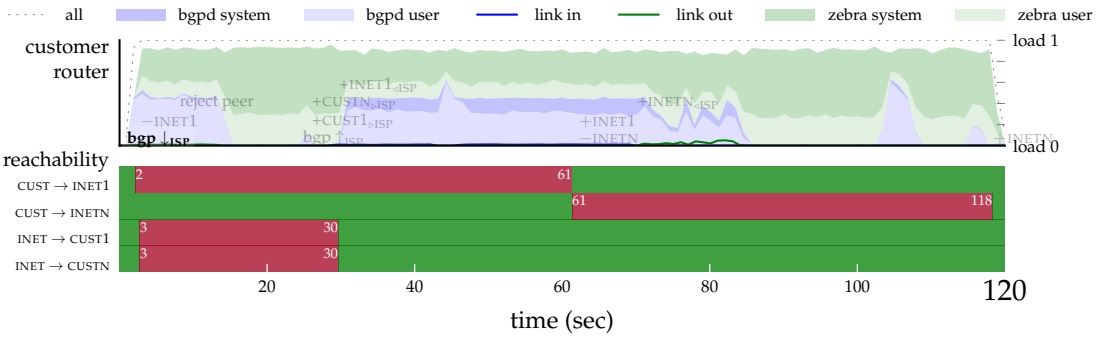


Figure 4.5: Partial system chart for clean shutdown rehoming with router-id spoofing and Graceful Restart, for the trial with the minimal overall outage time. The full system chart is available as Figure A.12.

The figure illustrates that, as the customer router detects the peering loss (denoted by the **bgp** $\downarrow_{\text{ISP}}$ annotation), the customer router deletes the routes that the initial router had advertised. Unfortunately, however, nothing in the figure explains why the customer router does so. To answer this question, we consult the specification for Graceful Restart [73].

We find, in Section 5 of the specification, that if a peering is terminated due to a BGP NOTIFICATION message, the peer receiving the message must terminate the peering session, and delete all routes received through that session. Thus, our clean shutdown procedure is incompatible with Graceful Restart.

The same section of the specification, however, offers a way forward. Specifically, it indicates that, if graceful restart is enabled for a peering, and a new session establishment request is received for that peering, the router receiving the request should terminate the old peering session, and accept the new peering session. This essentially reverses the behavior of BGP without Graceful Restart, which would maintain the old peering session, and reject the new one.

With this knowledge in hand, we revisit our naïve rehoming procedure. Our hope is that naïve rehoming can be combined with Graceful Restart to eliminate computation on the customer router as a cause of down time during rehoming. Note that, based on the lesson learned in Section 3.2.5, we use naïve rehoming with router-id spoofing.

4.4 Graceful Restart with naïve rehoming

As we can see in Table 4.2, the combination of naïve with router-id spoofing and Graceful Restart provides a significant benefit. Specifically, the outage time for traffic from the customer to INET1 and INETN drops to tens of milliseconds. This is a dramatic improvement over both naïve, and clean shutdown. Unfortunately, however, the outage time for traffic from the Internet to CUST1 and CUSTN remains substantial. The net result is that mean overall outage times are greater for naïve with Graceful Restart, than for clean shutdown.

To understand why, we examine Figure 4.6. As with Figure 3.3, we observe that the outages for traffic to CUST1 and custn continue until the remote router receives route withdrawal messages from the initial router (**-CUST1**$_{\text{cl}}$ and **-CUSTN**$_{\text{cl}}$), and updates its FIB ($\sim$**CUST1** and $\sim$**CUSTN**). The problem,

	clean shutdown	naïve	naïve + spoofing + graceful
internet to CUST1	37.52	150.24	158.46
internet to CUSTN	37.50	150.25	158.47
customer to INET1	58.82	65.12	0.06
customer to INETN	61.61	64.57	0.05
any	120.20	156.43	158.48

Table 4.2: Comparison of mean outage times, in seconds, and over ten trials, for clean shutdown, naïve, and naïve with router-id spoofing and Graceful Restart. Data for graceful restart and naïve are copied from Table 3.2.

as noted in Section 3.2.2, is that the remote router will prefer the routes advertised by the initial router, as long as they are still viable.

Two straightforward possibilities for addressing this problem are i) change the remote router's preferences, or ii) force the routes advertised by the initial router to be invalidated sooner. The former possibility can be realized, for example, by using BGP's LOCAL_PREF attribute to instruct the remote router that the routes to CUST1 and CUSTN advertised by the target router should be favored over those advertised by the initial router. The latter possibility could be realized, for example, by instructing the initial router to terminate its BGP session with the customer router, after the layer-two connectivity had been reconfigured.[4]

We choose to employ LOCAL_PREF, to address a subtle issue that we have thus far ignored. Namely, if the initial router withdraws advertisements for CUST1 and CUSTN before the remote router has received new advertisements for them (from the target router), then the remote router will propagate these withdraws to its external BGP peers. This would expose effects of rehoming to systems outside of the ISP's network. We discuss this further in Sections 7.1.3, and 8.3.1.

By using LOCAL_PREF instead, we avoid the need for the initial router to withdraw its routes. We simply let both routes exist in the network, but instruct the remote router to choose the target router's advertisements for CUST1 and CUSTN over those of the initial router. As long as the advertisements from the target router are received before the initial router times out its peering with the customer, external peers need not be informed of a routing change.

4.5 Graceful Restart with local-preference

The LOCAL_PREF attribute can be applied in a number of different ways. Accordingly, before presenting our experimental results, we first discuss the range of alternatives, and explain the configuration that we choose to evaluate. Following that discussion, we present both high-level results, and in-depth analysis.

[4]Terminating the peering before reconfiguring layer-two would risk having the customer router delete the routes advertised by the initial router, as observed in our experiments with clean shutdown and Graceful Rehoming.

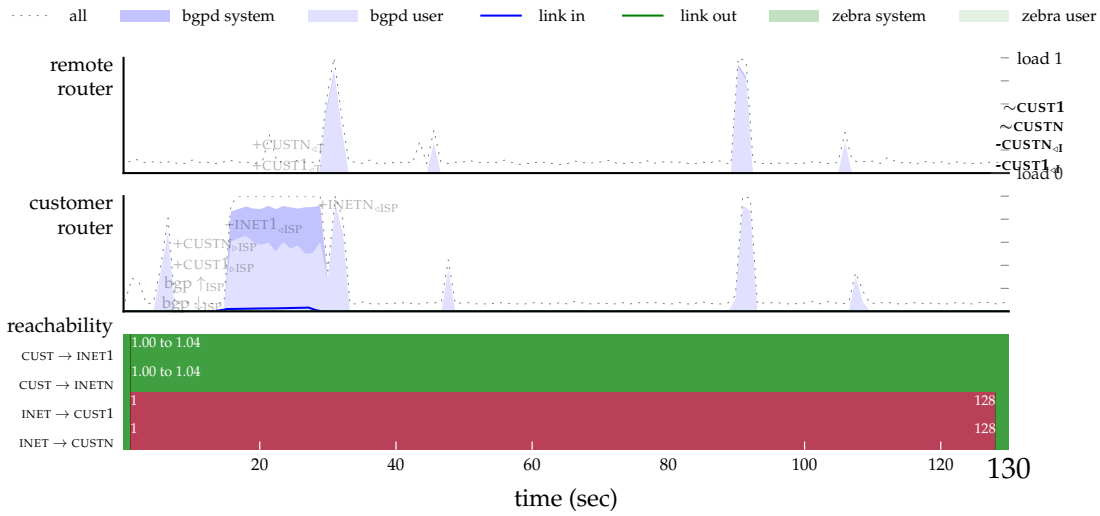


Figure 4.6: Partial system chart for naïve rehoming with router-id spoofing and Graceful Restart. The trial with the minimal outage time is depicted. The full system chart is available as Figure A.13.

4.5.1 Configuration Notes

There are two principal choices with respect to the use of the LOCAL_PREF attribute. These are: which routers will set the attribute; and whether the attribute should be set "inbound", meaning on receipt of the advertisements for the customer routes, or "outbound", meaning before the propagation of these advertisements to peers.

Considering the three ISP-side routers in our experiment (the initial router, target router, and remote router), we first rule out setting LOCAL_PREF on the remote router, for two reasons. First, in real world topologies, there will be many remote routers. Thus, this approach would be overly burdensome, in terms of the changes required to the network configuration. Second, at the remote router, it is non-trivial to even identify which routes belong to the customer being rehomed.

The choice between initial router and target router is less clear. We could configure the initial router to set a lower-than-default LOCAL_PREF for its routes to the customer prefixes, or we could configure the target router to set a higher-than-default LOCAL_PREF for these routes. We somewhat arbitrarily choose to apply the LOCAL_PREF attribute on the target router, but note that this has the benefit of keeping configuration changes located on the "active" router for a customer.

The remaining choice is whether to apply the LOCAL_PREF setting inbound, or outbound. We apply this setting inbound, for reasons similar to those driving our choice not to apply changes to LOCAL_PREF on the remote router. First, if we were to apply the setting outbound, it would need to be applied on all peerings, except for the one with the customer router. Second, in order to apply the setting on the outbound side, we would need to do additional work to identify the routes learned from the peering with the customer.[5]

[5]Another potential reason for choosing to apply LOCAL_PREF inbound would be to force the target router to prefer its route for the customer prefixes, learned from its peering with the customer router, over that of the initial router. However, we do not need LOCAL_PREF to force this decision, as EBGP routes are preferred over equivalent IBGP routes by default.

	clean shutdown	naïve + spoofing + graceful	naïve + spoofing + graceful + localpref
internet to CUST1	37.52	158.46	23.17
internet to CUSTN	37.50	158.47	21.77
customer to INET1	58.82	0.06	0.05
customer to INETN	61.61	0.05	0.05
any	120.20	158.48	23.18

Table 4.3: Outage times, in seconds, and over ten trials, for naïve rehoming using router-id spoofing, Graceful Restart and local-preference. For ease of comparison, the mean outage times for clean shutdown, and naïve with router-id spoofing and Graceful Restart (but without local-preference) are copied here from Table 4.2.

4.5.2 Empirical Results

We begin our presentation of the results with Table 4.3. This table presents outages times with LOCAL_PREF, and compares them to the outage times of the clean shutdown, and naïve without LOCAL_PREF. At a high level, we see that this new method of rehoming reduces downtime for traffic both inbound to, and outbound from, the customer.

As desired, the application of LOCAL_PREF has dramatically improved the outage time for CUST1 and CUSTN, as compared to naïve rehoming with Graceful Restart, but without LOCAL_PREF. Perhaps surprisingly, however, we find that this new configuration also improves outage times for these prefixes over clean shutdown. To understand why, we turn to Figure 4.7.

Therein, we examine the timing of the events critical to the propagation of CUSTN from the customer router to the remote router. In particular, we are interested in the **bgp** $\downarrow_{\text{ISP}}$ and **bgp** $\uparrow_{\text{ISP}}$ events on the customer router, the **+CUSTN**$_{\triangleleft C}$ and **+CUSTN**$_{\triangleright R}$ events on the target router, and the end of the outage for traffic to CUSTN, on the reachability graph. Focusing on the events, we make the following observations:

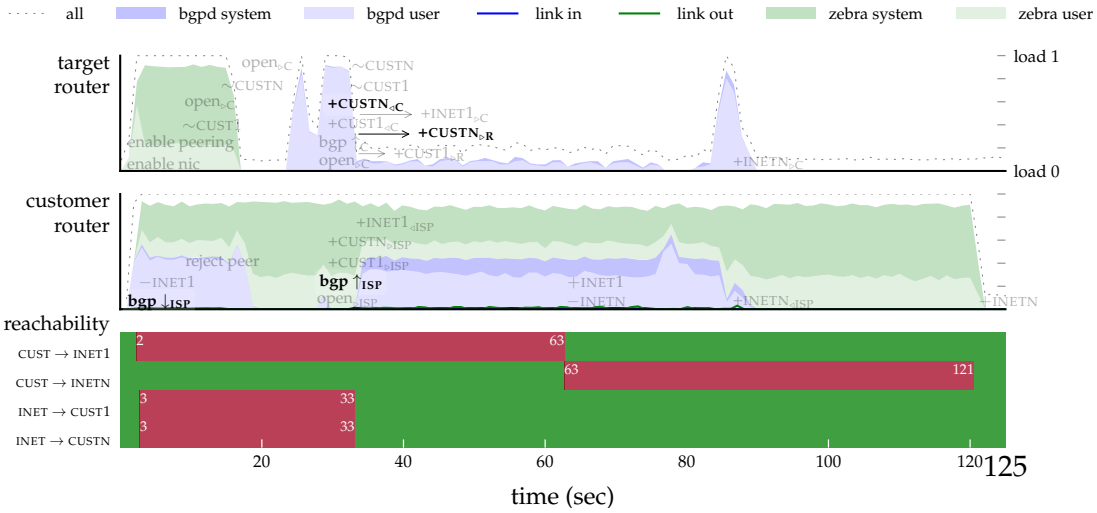
1. The time between **bgp** $\downarrow_{\text{ISP}}$ and **bgp** $\uparrow_{\text{ISP}}$ is much longer for clean shutdown, than for naïve. This can be explained by the fact that, as explained in Section 4.3, Graceful Restart enables the customer router to accept a new peering immediately. In contrast, as explained in Section 3.2.5, without Graceful Restart, the router may reject new peerings while it clears state from previous ones.

2. The time between **bgp** $\uparrow_{\text{ISP}}$ and **+CUSTN**$_{\triangleleft C}$ appears about the same for both experiments.

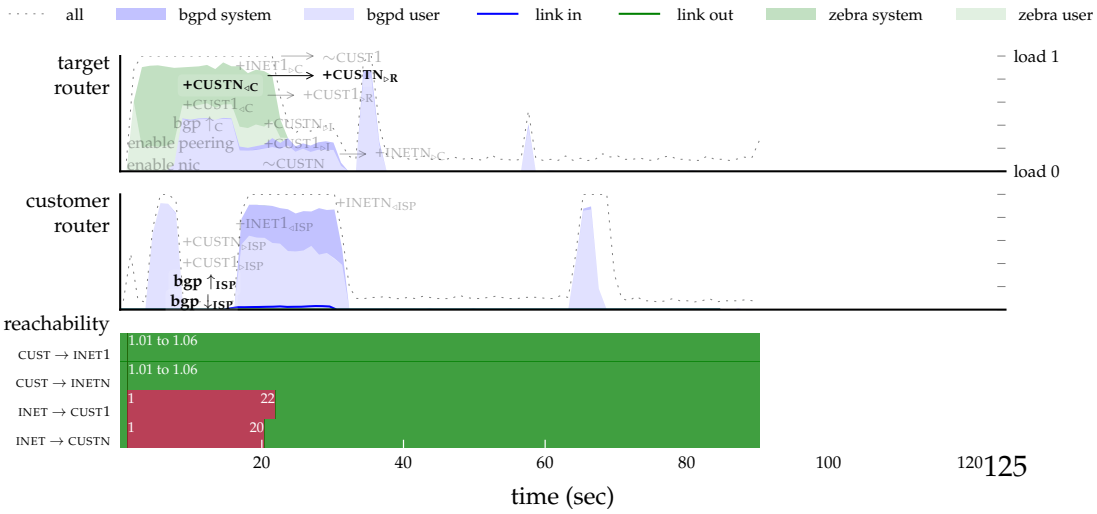
3. The time between **+CUSTN**$_{\triangleleft C}$ and **+CUSTN**$_{\triangleright R}$ is considerably longer for naïve, than for clean shutdown. This may be due to contention for the CPU. In the clean shutdown case, we observe that `zebra` is not active during this interval. In the naïve case, in contrast, both `zebra` and `bgpd` are consuming CPU cycles between **+CUSTN**$_{\triangleleft C}$ and **+CUSTN**$_{\triangleright R}$.

4. In both cases, the outage for traffic to CUSTN ends shortly after **+CUSTN**$_{\triangleright R}$.

5. The reduction in the delay between **bgp** $\downarrow_{\text{ISP}}$ and **bgp** $\uparrow_{\text{ISP}}$ is larger than the reduction in the



(a) clean shutdown with router-id spoofing



(b) naïve rehoming with router-id spoofing, Graceful Restart, and LOCAL_PREF

Figure 4.7: Partial system charts for clean shutdown with router-id spoofing, and naïve rehoming with router-id spoofing, Graceful Restart and LOCAL_PREF, for the trials with the minimal overall outage times. For the complete system charts, see Figures A.14 and A.15.

length of the INET1 → CUSTN outage. This is likely due to the CPU contention identified in observation 3.

Based on the above observations, we conclude that the improvement in outage times, over clean shutdown, for traffic to CUST1 and CUSTN, is due to the reduction in time taken to establish the new peering session.

Before proceeding, we make one more observation about the effects of Graceful Restart. Focusing on the customer router timeseries in Figure 4.7(b), we see that while Graceful Restart has eliminated the computation for the `zebra` process, the `bgpd` process still consumes a significant amount of CPU time. The key, however, is that much of that computation is no longer on the critical path. That is, connectivity is restored long before the computation completes.

This parallels the optimization strategy we will pursue in Chapters 5 and 6. Namely, in Chapter 5, we seek out code optimization, to reduce computation time on the target router. After exhausting the most promising code optimizations, we continue, in Chapter 6, with scheduling optimizations to move computation off the critical path.

4.6 Design discussion

Having demonstrated the feasibility of employing Graceful Restart to improve down time during rehoming, we now pause for a moment, and consider our design choices. In particular, we ask if the constraints that led us to switch from clean shutdown to naïve rehoming are fundamental, or if we might do better to relax those constraints.

In Section 4.3, we identified an incompatibility between our clean shutdown rehoming procedure, and the Graceful Restart specification. Namely, because Graceful Restart requires a peer receiving a BGP NOTIFICATION message to delete all routes from the peer transmitting the NOTIFICATION, clean shutdown can not eliminate the work done by the customer router to delete, and then re-install routes.

An alternative solution, then, would be to revise Graceful Restart to allow the customer to retain routes in this case. However, we argue against such a revision, for two reasons. First, in cases where a network operator terminates a peering session without the intent to restore that peering, it is important that the customer router remove the associated routes promptly. Second, terminating the BGP session on the initial router would cause it inform its peers that the customer routes were no longer valid. This would expose the effects of rehoming to systems outside of the ISP, as explained in Section 4.4.

4.7 Avenues for improvement

As shown in Table 4.3, we can leverage Graceful Restart to dramatically reduce the downtime during rehoming, as compared to clean shutdown. Specifically, the total outage time falls from a mean of 120.20 seconds, to a mean of 23.18 seconds. Thus, with a target of "five nines" reliability, a single rehoming event would consume only 7.35% of the annual outage budget.

Nonetheless, we would like to do better. Consider, for example, the impact of our current rehoming design when used to reduce the impact of router software upgrades. If we assume that rehoming is used to move customers to a temporary router, and then back to their original router, each upgrade would require two rehomings, and thus consume 14.7% of the annual outage budget.

Based on Cisco's IOS release history [25], and the above assumption, keeping routing software up to date would require 70% of the annual outage budget. This would leave little room for outages due to hardware upgrades, unexpected failures, and the like.

To identify avenues for improvement, we consult Figure 4.7(b). Examining the reachability graph and the timeseries for the target router, we note that the outage time can be largely explained by CPU processing delays on the target router. That is, with Graceful Restart having eliminated much of the processing on the customer router, the target router is once again the bottleneck. Thus, to further reduce outage time, we should optimize the computation on the target router.

4.8 Conclusion

We set out, in this chapter, to employ Graceful Restart to reduce the down time caused by computation on the customer router. Our initial results were discouraging: we found that clean shutdown, the rehoming procedure that performed best in Chapter 3, was incompatible with Graceful Restart. Pushing forward, however, we found that the naïve rehoming procedure, in combination with router-id spoofing, Graceful Restart, and BGP LOCAL_PREF, reduces mean overall down time to under 25 seconds.

Recall that in Section 2.3.1, we found that the mean overall time for "bgp + dynamic" customers, on Low Spec hardware, was 143 seconds. Thus, we now have a rehoming process that yields an 82% reduction in outage time over router restart. Even compared to High Spec hardware, as evaluated in Section 2.3.2, our rehoming process yields a 55% improvement.

Nonetheless, work remains. Despite our improvements, the frequency of routing software updates means that an ISP targeting "five-nines" reliability could spend nearly three-fourths of its annual outage budget on software updates alone. In order to address this, we must solve the problem of delays caused by computation on the target router. Accordingly, in Chapter 5, we will examine where the CPU cycles are spent, and whether it is feasible to optimize `zebra` and `bgpd` to reduce their CPU demands.

Measure twice, cut once.

5 On Code

Having shown, in Chapter 3, that rehoming can remove the ISP-side router as the source of outage time during software upgrades, we proceeded, in Chapter 4, to optimize BGP processing on the customer router. We showed that Graceful Restart could eliminate CPU delays due to the `zebra` process, and shift much of the computation of the `bgpd` process off of the critical path. With these improvements in the behavior of the customer router, we find that the ISP-side router is, once again, the bottleneck in the system.

Accordingly, we now turn our attention to the activities of the `zebra` and `bgpd` processes on the target router. Through the use of existing profiling tools, logfiles, and two new pieces of instrumentation, we identify hotspots in these processes. We then develop and empirically evaluate two simple patches to improve their performance. We show that these patches can reduce the CPU time used by `zebra` and `bgpd` by 87%, and 20%, respectively. These processing improvements yield a reduction in overall outage time of 48%, as compared to the best results of Chapter 4.

The remainder of this chapter is structured as follows:

- In Section 5.1, we introduce OProfile [58], one of the principal tools we use in our optimization efforts.
- In Section 5.2, we use profiling data from OProfile, log file messages generated by `zebra`, existing and new scheduling statistics, and inspection of source code to identify a defect in Quagga's work queue code. We show that a single line change resolves this defect, consequently yielding a 87% reduction in the CPU time used by `zebra`, during rehoming.
- In Section 5.2, we use OProfile data and new instrumentation to identify a performance issue due to the sizing of hash tables in `bgpd`. We show that enlarging some of its hash tables can reduce the CPU time required by `bgpd`, during rehoming, by 20%.
- In Section 5.4, we summarize and conclude.

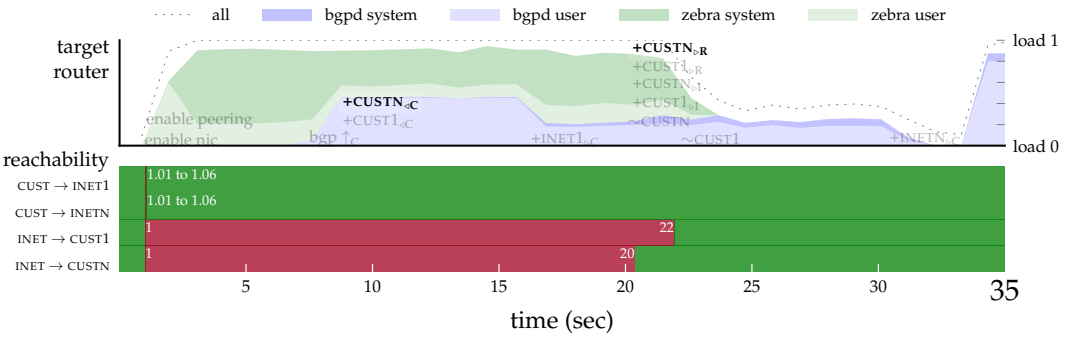


Figure 5.1: Partial system chart for naïve rehoming with Graceful Restart and LOCAL_PREF, for the trial with the minimal overall outage time. This chart repeats data from Figure 4.7(b), but with the time scale narrowed to provide greater detail. For the complete system chart, see Figure A.15.

5.1 CPU Profiling With OProfile

A good profiling tool is essential for any CPU optimization task. For profiling CPU time in our work here, we use OProfile [58]. OProfile is a hardware performance counter-based, whole-system profiler. It is similar, in many respects, to the Digital Continuous Profiling Infrastructure [8]. OProfile has three key features that make it well suited for our task, as we detail below.

First, OProfile supports whole-system profiling, including time spent in kernel mode. The ability to profile kernel-mode computation is vital to our efforts, as can be seen in Figure 5.1, a system chart for our rehoming solution with Graceful Restart. In this figure, we see that most of the CPU time used by `zebra` is attributed to system mode (dark green), rather than user mode (light green).

Second, OProfile allows profiling to be started and stopped at arbitrary points in time. In contrast, process profiling tools like `gprof` [31], report profiling data for the entire process lifetime. In our case, that would include, for example, the computation required to load the Internet routes on to the target router from the remote router. As can be seen in Figure 5.2, that computation dwarfs the computation required during rehoming.[1]

Third, because OProfile uses hardware performance counters, it can provide fine-grained reporting with low overhead. Low profiling overhead is valuable generally, because it enables profiling runs to complete in a reasonable amount of time. In our case, low overhead profiling is important for an additional reason. Namely, large perturbations in running time might change the influence of BGP timers on system behavior.

OProfile offers a wealth of performance metrics, including counts of clock cycles, instructions retired, cache misses, branch mispredictions, memory stalls, pipeline stalls, and more. In our optimization efforts, we use the simplest of these metrics: clock cycle counts. Any of these counts can be reported on a per-binary, per-function, or per-line basis; we use the per-function and per-line data.

Based on our measurements of CPU utilization from `/proc/stat` data, we find that `zebra` uses an average of 13.54 seconds during the rehoming process, while `bgpd` uses an average of 11.90

[1]We note that Figure 5.2 presents measurements of the initial router, rather than the target router. However, the two routers are symmetric with regard to their roles relative to the remote router.

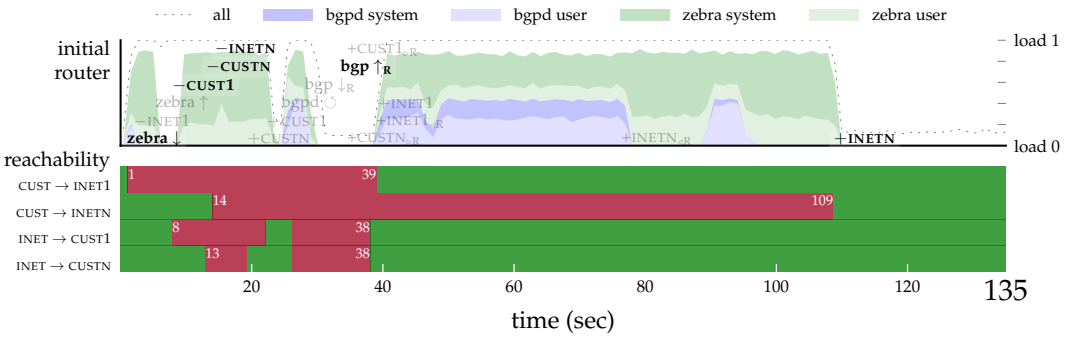


Figure 5.2: Partial system chart for router restart with a single static customer. Focusing on **bgp** $\uparrow_R$, near time 40 seconds, and $+$**INETN**, near 110 seconds, we see the initial router requires approximately 70 seconds to learn Internet routes from the remote router. This chart repeats Figure 2.6(a). For the complete system chart, see Figure A.1.

seconds.[2] Accordingly, we first attempt to optimize `zebra`, and then pursue optimizations in `bgpd`.

5.2 Optimizing `zebra`

In this section, we undertake the optimization of the CPU time used by `zebra`. We first use OProfile data to locate functions that account for a significant fraction of CPU time. After finding that the function with the most CPU time is not easily optimized, we use log file data, existing and new instrumentation, and source code inspection, to devise a strategy for reducing the frequency with which the hot spot function is called. We then assess our improvements, and suggest opportunities for further optimization.

5.2.1 Finding hot-spots with OProfile

We begin our optimization efforts by using OProfile to identify "hot spots": locations in the software which account for significant fractions of the total program execution time. Note that, as seen in Figure 5.1, the program execution time for `zebra` includes a significant amount of time spent executing kernel code. Fortunately, OProfile can generally attribute kernel-mode execution time to the appropriate user process.[3]

In Table 5.1, we present OProfile for execution time by function name. We truncate the data to focus on the top ten functions. Examining the data, we note that the list of top ten functions is very surprising. While we might expect `zebra` to be exercising functions related to the inspection or manipulation of the kernel's routing information, `zebra` is, instead, spending one-third of its execution time in the function `acpi_pm_read_verified`. This is promising, in that optimizing this single piece of code could significantly improve performance. At the same time however, kernel code is often difficult to optimize. And any such optimization would be specific to the Linux kernel.

[2]For details on the collection of `/proc/stat` data, see Section 2.1.5.

[3]A notable exception is time spent processing hardware interrupts. For example, time spent processing an input packet would not necessarily be attributed to the process that eventually receives the packet.

function	object file	CPU seconds
`acpi_pm_read_verified`	vmlinux	4.50
`system_call`	vmlinux	0.52
`getnstimeofday`	vmlinux	0.41
`copy_to_user`	vmlinux	0.36
`clock_gettime`	librt-2.9.so	0.32
`thread_fetch`	libzebra.so.0.0.0	0.26
`nexthop_active_update`	zebra	0.25
`do_select`	vmlinux	0.25
`prefix_match`	libzebra.so.0.0.0	0.24
`vdso`	vdso	0.23

Table 5.1: Top ten functions called by `zebra`, in terms of CPU time. Note that `vdso` refers to a page in memory which contains the CPU instructions used to initiate system calls [67, 75].

Nonetheless, we proceed to examine the code for `acpi_pm_read_verified`, provided here as Listing 5.1. Note that this listing includes per-line execution time data from OProfile, to the left of each line of source code. As noted in the caption for the listing, the reference to the `BUILDIO` macro generates the `inl` function. With this information, and the comment at lines 13–18, we conclude that nearly all of the time for `acpi_pm_read_verified`, and approximately 34% of `zebra`'s execution time during rehoming, is spent reading the time from the hardware clock.

As expected, then, the opportunities for optimizing this function are limited. Thus, we turn our attention instead, to improving execution time by reducing the number of times this function is called. To do so, we need to determine which function, or functions, call `acpi_pm_read_verified`. An ideal approach to determining this would be to use a call-graph profiler. And indeed, OProfile does offer call-graph profiling. Unfortunately, in our experiments, we were unable to obtain call-graph data from OProfile.[4]

Faced with the inability to generate a call-graph profile, we take a closer look at the data of Table 5.1. Therein, we note that in addition to `acpi_pm_read_verified`, the list of top ten functions includes `system_call`, and `do_select` in the kernel binary, and `thread_fetch` in the `zebra` binary. Based on this list, and the knowledge that the `select` system call is often used to wait until network sockets are ready for input/output, or until a timeout occurs, we hypothesize that the calls to `acpi_pm_read_verified` are due to the `thread_fetch` function.

Accordingly, we turn our attention to the OProfile-annotated source for `thread_fetch`, excerpted here as Listing 5.2. Therein, we observe that the `thread_fetch` does call `select`. But we see that the function explicitly sets the file descriptors to be watched by `select` (at lines 2–4), and that the function also explicitly sets the timeout parameter for the system call (at lines 8 and 13, as appropriate). Thus, the `thread_fetch` function exhibits no obvious errors in its invocation of the `select` system call.

We find ourselves, then, at a standstill. We know that `acpi_pm_read_verified` is a hot spot in `zebra`. And we believe this hot-spotting is caused by excessive calls to `select` by `thread_fetch`. Having found, however, no obvious defects in `thread_fetch`, we must look elsewhere. Accordingly, we now switch gears. In the next subsection, we use logfile messages to understand the

[4]Specifically, no matter how large the callgraph depth we specified when starting OProfile, the output data contained only a single level of stack information.

```
33.9092 :BUILDIO(l, , int)

        :static inline u32 read_pmtmr(void)
        :{
        :       /* mask the output to 24 bits */
 0.0180 :       return inl(pmtmr_ioport) & ACPI_PM_MASK;
        :}

        :u32 acpi_pm_read_verified(void)
 0.1248 :{ /* acpi_pm_read_verified total:  38329 34.4193 */
        :       u32 v1 = 0, v2 = 0, v3 = 0;
        :
        :       /*
        :        * It has been reported that because of various broken
        :        * chipsets (ICH4, PIIX4 and PIIX4E) where the ACPI PM clock
        :        * source is not latched, you must read it multiple
        :        * times to ensure a safe value is read:
        :        */
        :       do {
        :               v1 = read_pmtmr();
        :               v2 = read_pmtmr();
        :               v3 = read_pmtmr();
 0.1787 :       } while (unlikely((v1 > v2 && v1 < v3) || (v2 > v3 && v2 < v1)
        :                         || (v3 > v1 && v3 < v2)));
        :
        :       return v2;
 0.1895 :}
```

Listing 5.1: Source code of `acpi_pm_read_verified` and related functions, as annotated by OProfile. The number at the left of each line indicates the percentage of total program execution time attributed to that line. `BUILDIO` is a macro which is used to generate the `inl` function. Note that both `read_pmtmr` and `inl` are inline functions. Accordingly, in the per-function report of Table 5.1, their execution times appear as part of `acpi_pm_read_verified`, rather than being broken out separately. The annotated source is from the trial with the minimal total outage time.

```
1          :      /* Structure copy.  */
2   0.3215 :      readfd = m->readfd;
3   0.3403 :      writefd = m->writefd;
4   0.3314 :      exceptfd = m->exceptfd;
5          :
6          :      /* Calculate select wait timer if nothing else to do */
7   0.0045 :      quagga_get_relative (NULL);
8          :      timer_wait = thread_timer_wait (&m->timer, &timer_val);
9          :      timer_wait_bg = thread_timer_wait (&m->background, &timer_val_bg);
10         :
11 9.0e-04 :      if (timer_wait_bg &&
12         :          (!timer_wait || (timeval_cmp (*timer_wait, *timer_wait_bg) > 0)))
13         :       timer_wait = timer_wait_bg;
14         :
15  0.0377 :      num = select (FD_SETSIZE, &readfd, &writefd, &exceptfd, timer_wait);
```

Listing 5.2: Excerpted source code of `thread_fetch`, as annotated by OProfile. This code executes within the body of a `while(1)` loop. The number at the left of each line indicates the percentage of total program execution time attributed to that line. In this excerpt, no single line accounts for even 1% of the execution time. The annotated source is from the trial with the minimal total outage time.

processing that `zebra` performs during our rehoming process.

5.2.2 Gleaning behavior from logfile messages

Having failed to identify a viable optimization strategy from CPU profiling data alone, we now turn to a very naïve method to gain more insight into program behavior. Specifically, we repeat our experiment, with all possible debugging options enabled for `zebra`. While the overhead of generating and recording the debug messages does threaten to perturb system behavior, the logfile messages may prove valuable nonetheless.

Running our rehoming process with full debugging enabled for `zebra` generates a 190 MB log file, with about 1.9 million lines. To reduce this data to a manageable size, we sample the first message of each 10 second interval, where each interval begins at a timestamp that is evenly divisible by 10 seconds. We present the reduced data in Listing 5.3.

This listing suggests that `zebra` performs four phases of processing. First, at time 22:34:44 (line 2), `zebra` receives a RTM_NEWADDR message from the kernel, indicating that an IP address has been added to a network interface. Second, at time 22:34:50 (line 3), many prefixes are queued for processing via `rib_meta_queue_add`. Third, at time 22:35:20 (line 6), many prefixes are queued for processing via `rib_queue_add`. Finally, at time 22:35:40 (line 8), processing is performed via `rib_process`.

Apart from providing us with a sliver of insight into what `zebra` is doing, these log file messages also offer another angle of attack for our theory about `acpi_pm_read_verified`, and the scheduling code in `zebra`. Namely, if there were a problem with the work queue referenced in the log messages, the problem might manifest as an excessive number of calls to `select`. To test this hypothesis, we re-run the experiment, capturing work queue statistics, before and after rehoming, using the

```
     1 22:34:17 ZEBRA: Vty connection from 127.0.0.1                                  1
127898 22:34:44 ZEBRA: netlink_parse_info: netlink-listen type RTM_NEWADDR(20), ...   2
178339 22:34:50 ZEBRA: rib_meta_queue_add: 74.62.144.0/20: queued rn 0xba4d2e28 into ... 3
233116 22:35:00 ZEBRA: rib_meta_queue_add: 160.130.51.0/24: queued rn 0xbb5b7cd0 ...   4
270351 22:35:10 ZEBRA: rib_meta_queue_add: 207.163.110.0/23: queued rn 0xbcceb9f8 ...  5
268155 22:35:20 ZEBRA: rib_queue_add: 89.34.100.0/23: work queue added                 6
210926 22:35:30 ZEBRA: rib_queue_add: 200.68.64.0/22: work queue added                 7
 45396 22:35:40 ZEBRA: rib_process: 41.235.187.0/24: Updating existing route, select ... 8
 51661 22:35:50 ZEBRA: rib_process: 65.205.251.0/24: Updating existing route, select ... 9
 49359 22:36:00 ZEBRA: rib_process: 72.95.99.0/24: rn 0xba520e90 dequeued              10
 92984 22:36:10 ZEBRA: rib_process: 86.106.75.0/24: Updating existing route, select ... 11
109338 22:36:20 ZEBRA: rib_process: 124.105.64.0/19: Updating existing route, select ... 12
112422 22:36:30 ZEBRA: rib_process: 192.85.78.0/24: rn 0xbbd5d688 dequeued            13
127223 22:36:40 ZEBRA: rib_process: 202.255.102.0/24: rn 0xbc843378 dequeued          14
 18464 22:36:50 ZEBRA: rib_process: 217.23.70.0/24: Updating existing route, select ... 15
     2 22:37:02 ZEBRA: zebra message comes from socket [11]                           16
     3 22:38:02 ZEBRA: zebra message comes from socket [11]                           17
     1 22:42:26 ZEBRA: Vty connection from 127.0.0.1                                  18
```

Listing 5.3: Sampling of logfile messages generated by `zebra` during rehoming. The first entry of every 10 second interval is provided. The numbers at left indicate the total number of log file messages during each interval. Ellipsis denote that a line has been truncated to fit the page width. These logfile messages come from the trial with the minimal overall outage time.

`show work-queues` command in `zebra`.

We present the data from `show work-queues` in Table 5.2. While `zebra` provides many statistics on the work queues, the statistic of particular interest to our optimization task is the number of times the work queue was run. This statistic is listed under the "Q. Runs Total" column in the table. Examining the data in the table, we observe that the work queue was run approximately 315,000 times during the course of rehoming. Given that our ISP-side RIB trace has 316,592 prefixes (see Table 2.3), this suggests that `zebra` is processing only one prefix per queue run. And, depending on how the work queue interacts with the program's scheduler, `zebra` may be processing only one prefix for each call that it makes to `select`.

		List	(ms)	Q. Runs	Cycle Counts		
	P	Items	Hold	Total	Best	Gran.	Avg.
before rehoming		0	10	5370247	1	1	1
after rehoming		0	10	5686145	1	1	1

Table 5.2: Work-queue statistics for the `route_node processing` workqueue in `zebra`. These statistics come from the `show work-queues` command in `zebra`'s command-line interface, for the trial with the minimal overall outage time. Note that the "before rehoming" statistics reflect the side-effects of previous trials, and work done during experiment setup.

```
1     if (wq->cycles.granularity == 0)
2       wq->cycles.granularity = WORK_QUEUE_MIN_GRANULARITY;
3
4   ++(wq->qlen_hist[MIN(get_order(listcount(wq->items)),
5                        sizeof(wq->qlen_hist) / sizeof(wq->qlen_hist[0]))]);
6   for (ALL_LIST_ELEMENTS (wq->items, node, nnode, item))
7   {
8     assert (item && item->data);
```

Listing 5.4: Core source code for patch to capture work queue length statistics. The complete patch is provided as Listings B.16–B.19.

```
1
2   wq->runs++;
3   wq->cycles.total += cycles;
4   wq->yields += yielded;
5
6 #if 0
7   printf ("%s: cycles %d, new: best %d, worst %d\n",
```

Listing 5.5: Core source code for patch to capture work queue yield counts. The complete patch is provided as Listings B.16, B.18, and B.19.

5.2.3 Deeper insight from instrumentation and code inspection

While the work queue statistics have provided a promising lead, they do not tell us why the work queue is run often. For example, the work queue might have to be run so often because there is only a single item in the queue each time. On the other hand, it might be the case that the queue has multiple items, but the queue processing code yields after processing a single item.

To provide more insight in to the behavior of the work queues, we instrumented `zebra` to capture two additional statistics about its work queues. The first of these statistics is a histogram, with exponentially-sized bins, of the work queue length before each run of the queue. The second of these statistics is a count of the number of times the workqueue processing code determined that it should yield before processing any further entries. These statistics are captured via the patches of Listings 5.4 and 5.5, respectively.

Over ten trials, we found that the average yield count was 89.5, and that there was never more than one item in the queue. These statistics are seemingly contradictory. If the queue length never contains multiple items, why is the yield count non-zero? This contradiction can be explained by noting that the determination of whether or not to yield is made before checking if there are any remaining items on the queue. Thus, a work queue run which exceeds the scheduling quantum will be counted in the yield count, even if there are no further items to process.

With the contradiction resolved, an important question remains. Namely, why is the queue so small? Given the logfile messages of Listing 5.3, we would expect all the prefixes to be present on the queue before `rib_process` is called. To answer this question, we examine the source code of `rib_queue_add`, which we provide here as Listing 5.6. The comment at lines 15–22 of that listing explains that prefixes are not queued directly onto the work queue itself. Instead, the work queue

```
/* Add route_node to work queue and schedule processing */
static void
rib_queue_add (struct zebra_t *zebra, struct route_node *rn)
{

  if (IS_ZEBRA_DEBUG_RIB_Q)
    {
      char buf[INET6_ADDRSTRLEN];

      zlog_info ("%s: %s/%d: work queue added", __func__,
                 inet_ntop (rn->p.family, &rn->p.u.prefix, buf, INET6_ADDRSTRLEN),
                 rn->p.prefixlen);
    }

  /*
   * The RIB queue should normally be either empty or holding the only
   * work_queue_item element. In the latter case this element would
   * hold a pointer to the meta queue structure, which must be used to
   * actually queue the route nodes to process. So create the MQ
   * holder, if necessary, then push the work into it in any case.
   * This semantics was introduced after 0.99.9 release.
   */
  if (!zebra->ribq->items->count)
    work_queue_add (zebra->ribq, zebra->mq);

  rib_meta_queue_add (zebra->mq, rn);
}
```

Listing 5.6: Source code of `rib_queue_add`, from `zebra_rib.c`. The log file messages in Listing 5.3 suggest that this function is called frequently during rehoming.

holds a pointer to a meta-queue, which then contains the queued prefixes. As such, the work queue either holds a single item, or is empty.

While the comment in `rib_queue_add` explains why our histogram reported that the work queue never contained more than one item, we have yet to find an explanation for why `zebra` processes, on average, only a single prefix for each work queue run. Based on the log file messages of Listing 5.3, we would now expect the prefixes to be queued on the meta-queue. And, presumably, the function servicing the meta-queue would process multiple items on each meta-queue run.

To determine whether or not this is, in fact, the case, we examine the source code for the function which services the meta-queue, `meta_queue_process`. We provide the code for this function here as Listing 5.7. Examining the main loop of this function, at lines 11–16, we see that `meta_queue_process` processes at most a single item, from the lowest numbered of `MQ_SIZE` subqueues.[5] Afterwards, at line 17, the function checks if the meta queue contains any more items. If so, the function returns `WQ_REQUE`, indicating that the current work queue item should be enqueued back on to the queue from which it was taken.

[5]The purpose of the subqueues is to enforce a priority ordering amongst routing protocols. For example, RIB updates sent to `zebra` from OSPF are processed before the RIB updates sent from BGP. For further details, see [49].

```
1  /* Dispatch the meta queue by picking, processing and unlocking the next RN from
2   * a non-empty sub-queue with lowest priority. wq is equal to zebra->ribq and data
3   * is pointed to the meta queue structure.
4   */
5  static wq_item_status
6  meta_queue_process (struct work_queue *dummy, void *data)
7  {
8    struct meta_queue * mq = data;
9    unsigned i;
10
11   for (i = 0; i < MQ_SIZE; i++)
12     if (process_subq (mq->subq[i], i))
13       {
14         mq->size--;
15         break;
16       }
17   return mq->size ? WQ_REQUEUE : WQ_SUCCESS;
18 }
```

Listing 5.7: Source code of `meta_queue_process`, from `zebra_rib.c`. Our reading of the `zebra` source code indicates that this function services the items enqueued by `rib_queue_add` (see Listing 5.6).

Depending on how this requeue functionality interacts with the thread scheduler implemented in `thread_fetch`, our understanding of `meta_queue_process` may validate our theory that excessive calls to `select` are the root cause of the disproportionately large amount of CPU time used by `acpi_pm_read_verified`. Specifically, if `thread_fetch` calls `select` after the requeuing action, rather than immediately rerunning the work queue processing thread, then it would follow that `zebra` executes one call to `select` for every prefix.

To understand how requeuing interacts with the thread scheduling code in `zebra`, we examine the source code for the function that processes work queues, `work_queue_run`. To facilitate that examination, we present an abstracted version of `work_queue_run` as Listing 5.8. In this listing, we focus specifically on lines 2, 9, 19, and 39.

Line 2 defines a loop that iterates over all of the elements of the work queue. Line 9 calls the function that processes items for the work queue in question, and captures the return value in `ret`. If `ret` equals `WQ_REQUE`, line 19 requeues the item. Line 39 schedules the work queue processing thread for execution, if any items remain on the work queue after exiting the for loop. Note, however, that line 39 is outside the body of the for loop defined at line 2. Hence, requeued items should be processed before rescheduling the work queue processing thread.[6]

Before concluding that the problem lies elsewhere, however, we check for one more possible error. Note that the code at line 19 modifies the very same linked list that the for loop at line 2 is actively iterating over. Without due care, this combination could very well cause erroneous behavior. To determine whether or not that is the case here, we inspect the code for `work_queue_item_requeue`, and `ALL_LIST_ELEMENTS`. We provide the source for this function and macro as Listings 5.9 and 5.10, respectively.

[6]Note that the `break` statement at line 20 breaks out of the `switch` construct, rather than the for loop.

```
int work_queue_run (struct thread *thread) { ...
  for (ALL_LIST_ELEMENTS (wq->items, node, nnode, item)) { ...
    if (item->ran > wq->spec.max_retries) { ...
      work_queue_item_remove (wq, node);
      continue;
    } ...

    do {
      ret = wq->spec.workfunc (wq, item->data);
      item->ran++;
    }
    while ((ret == WQ_RETRY_NOW) && (item->ran < wq->spec.max_retries));

    switch (ret) {
      case WQ_QUEUE_BLOCKED: ...
      case WQ_RETRY_LATER: goto stats;
      case WQ_REQUEUE: {
        item->ran--;
        work_queue_item_requeue (wq, node);
        break;
      }
      case WQ_RETRY_NOW: ...
      case WQ_ERROR: ...
      case WQ_SUCCESS:
      default: {
        work_queue_item_remove (wq, node);
        break;
      }
    } ...

    cycles++; ...
    if ( !(cycles % wq->cycles.granularity) && thread_should_yield (thread)) {
      goto stats;
    }
  }

stats: ...
  if (listcount (wq->items) > 0)
    work_queue_schedule (wq, 0);
  else if (wq->spec.completion_func)
    wq->spec.completion_func (wq); ...
}
```

Listing 5.8: Abstracted source code of `work_queue_run`, from `workqueue.c`. Ellipsis denote that one or more lines have been omitted. Note, however, that all lines affecting the control flow of the function have been retained in this listing. Our reading of the `zebra` source code indicates that this function interfaces between the scheduling code in `thread_fetch` and the meta-queue servicing code in `meta_queue_process` (see Listing 5.7).

```
1 static void
2 work_queue_item_requeue (struct work_queue *wq, struct listnode *ln)
3 {
4   LISTNODE_DETACH (wq->items, ln);
5   LISTNODE_ATTACH (wq->items, ln); /* attach to end of list */
6 }
```

Listing 5.9: Source code of `work_queue_item_requeue`, from `workqueue.c`. This function is called by `work_queue_run` (see Listing 5.8), when `meta_queue_process` (see Listing 5.7) returns before processing all the items on the meta-queue.

```
1 /* List iteration macro.
2  * Usage: for (ALL_LIST_ELEMENTS (...) { ... }
3  * It is safe to delete the listnode using this macro.
4  */
5 #define ALL_LIST_ELEMENTS(list,node,nextnode,data) \
6   (node) = listhead(list); \
7   (node) != NULL && \
8     ((data) = listgetdata(node),(nextnode) = listnextnode(node), 1); \
9   (node) = (nextnode)
```

Listing 5.10: Source code of `ALL_LIST_ELEMENTS`, from `linklist.h`. This macro is used to iterate over queued items in `work_queue_run` (see Listing 5.8).

From these listings, we observe that `work_queue_item_requeue` simply removes the given item from the workqueue, and then appends the item to the tail of the list. Assuming, then, that the linked list manipulation macros `LISTNODE_DETACH` and `LISTNODE_ATTACH` operate correctly, the code for `work_queue_item_requeue` is error-free. Rather than inspect those macros now, we proceed under the assumption that they are correct, and turn our attention to `ALL_LIST_ELEMENTS`.

We first observe that the code for `ALL_LIST_ELEMENTS`, is not quite as simple as the code for `work_queue_item_requeue`. As noted in the comment at line 3, this code is intended to correctly handle the case where the current list item is removed. To do so, the code pre-fetches the next node pointer at the top of the loop (using the code at line 8), and uses the prefetched pointer to set the current node pointer before the next loop iteration (using the code at line 9).

In most cases, this code should perform correctly. Namely, if the current element is deleted, the loop should proceed to iterate over subsequent elements. However, in our particular case, where the work queue contains a single element, this code will fail to provide the desired iteration semantics. Because the queue contains a single element, the prefetched value, at line 8 of the source code for `ALL_LIST_ELEMENTS`, will be `NULL`. Accordingly, list iteration will terminate, even if the call to `work_queue_item_requeue` properly requeues the work quite item inside the body of the for loop.

```
                                                                                1
++(wq->qlen_hist[MIN(get_order(listcount(wq->items)),                           2
                     sizeof(wq->qlen_hist) / sizeof(wq->qlen_hist[0]))]);       3
while (listcount(wq->items))                                                    4
for (ALL_LIST_ELEMENTS (wq->items, node, nnode, item))                          5
{                                                                               6
  assert (item && item->data);                                                  7
```

Listing 5.11: Complete source code for patch to resolve the hot-spot in `zebra`. This patch prevents `work_queue_run` from returning prematurely. The patch is also available as Listing B.41.

	before `while`	with `while`	absolute change	% change
work-queue runs	315898.50	90.50	−315808.00	−100
total CPU seconds	13.49	1.74	−11.75	−87
overall outage seconds	23.27	15.91	−7.36	−32

Table 5.3: Improvement due to the patching of `work_queue_run`, with the change of Listing 5.11. All measurements reported are mean values over ten trials. Note that the data for "before `while`" come from the experiment of Section 5.2.3.

5.2.4 Resolving the hot-spot, and assessing our improvements

We now test our theories by applying a change which should, if our theories are correct, remedy the observed behavior. This patch, provided here as Listing 5.11, simply wraps the for loop with a `while` loop. The outer loop continues so long as the work queue is non-empty. Note that there are two cases where the for loop should terminate even though the queue is non-empty. These are at lines 16 and 33 of Listing 5.8. However, in both cases, the code exits the loop via an explicit `goto`, rather than a `break` statement. Hence, our change should not alter the behavior of `work_queue_run` in these cases.

We present the results of our change in Table 5.3. This table provides a before and after comparison of the system behavior. From this table, we make three observations. First, the number of work queue runs is dramatically reduced. Second, the total CPU time used by `zebra` is reduced by 11.75 seconds, or 87%. Third, the overall outage time is also significantly reduced, by 7.36 seconds.

The second and third observations, however, raise some questions of their own. First, why is the improvement in outage time greater than the total CPU usage of `acpi_pm_read_verified`, which was reported as 4.50 seconds in Table 5.1? Second, why is there a large gap, of 4.39 seconds, between the improvement in `zebra` processing time, and the improvement in overall outage time? We now consider these questions in turn.

To explain the greater-than-expected improvement, we present Table 5.4, a before and after comparison of the top ten functions, in terms of CPU time used. Comparing Tables 5.4(a) and 5.4(b), we note that, in addition to `acpi_pm_read_verified`, many other functions no longer rank in the top ten. Rather than simply reduce the cost due to `acpi_pm_read_verified`, we have reduced the cost due to all of the functions on the path to `acpi_pm_read_verified`, including `thread_fetch`, `system_call`, and others.

function	CPU seconds
acpi_pm_read_verified	4.50
system_call	0.52
getnstimeofday	0.41
copy_to_user	0.36
clock_gettime	0.32
thread_fetch	0.26
nexthop_active_update	0.25
do_select	0.25
prefix_match	0.24
vdso	0.23

(a) before patch

function	CPU seconds
route_lock_node	0.21
meta_queue_process	0.20
prefix_match	0.20
rib_queue_add	0.19
nexthop_active_update	0.17
route_node_match	0.17
route_next	0.15
malloc_consolidate	0.10
work_queue_run	0.04
_int_malloc	0.04

(b) with patch

Table 5.4: Top ten functions called by zebra, before and after our patch to work_queue_run, in terms of CPU time. Subtable (a) repeats data of of Table 5.1.

Having resolved the question of how our optimization efforts performed better than might be expected, according to our best data at the outset, we now turn to the question of why the optimization is not quite as effective as might now be expected. Namely, we want to understand the gap between the reduction in CPU time used by zebra, and the reduction in overall outage time.

To do so, we examine the system chart for this experiment, and contrast it to the system chart for an experiment before our patch. We present these paired charts as Figure 5.3. A comparison of the CPU activity on the target router, between the two subfigures, yields a simple explanation for this gap. Namely, there is now a significant amount of idle time on the target router, as zebra completes its work by about 5 seconds after the start of the experiment, while bgpd does not ramp up its work until approximately 7 seconds after the start of the experiment.

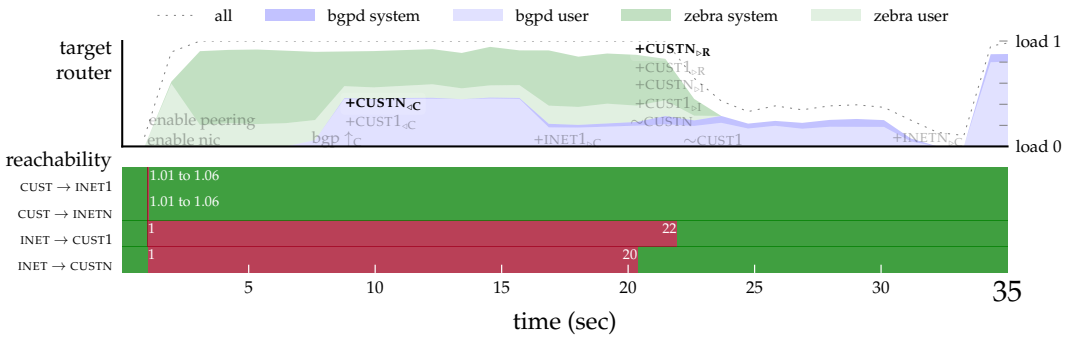
5.2.5 Avenues for improvement

The data of Table 5.3 and Figure 5.3 suggest two principal avenues for improvement. First, based on Table 5.3, we could attempt to further optimize the CPU time used by zebra. Second, based on Figure 5.3(b), we could attempt to optimize the idle time between the **enable nic** and **+cust1**$_{cc}$ events during rehoming.

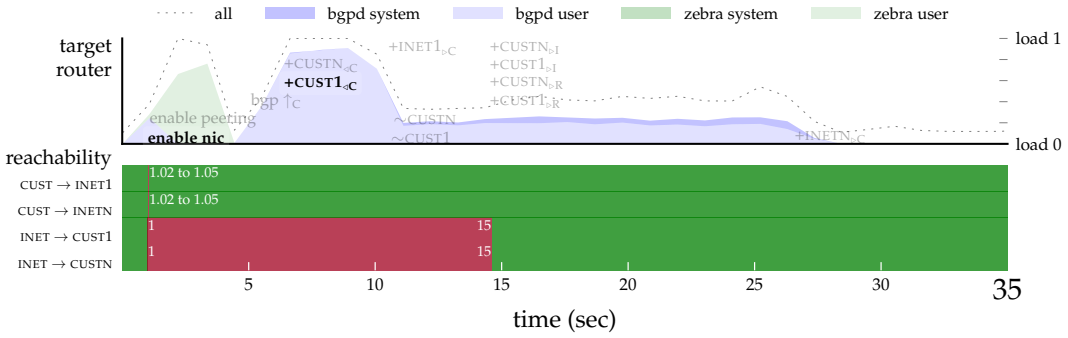
From Table 5.3, we expect that the CPU time optimization would yield an outage time improvement of no more than 1.74 seconds. Moreover, such optimization could be difficult, as none of the top ten functions in Table 5.4(b) individually accounts for more than 0.21 seconds of CPU time.

In comparison, based on the gap between the improvement in CPU usage due to the patch of Listing 5.11, and the improvement in outage time due to the same patch, we believe that addressing the idle time could yield an improvement of up to 4.39 seconds.

Given the greater opportunity of improvement with the latter approach, we set aside further CPU optimization for zebra. Instead, we will pursue idle time reductions. Before doing so, however, we first examine and optimize the CPU utilization of bgpd.



(a) before `while` patch



(b) with `while` patch

Figure 5.3: Partial system charts for rehoming with and without the patch of Listing 5.11, for the trials with the minimal overall outage times. Subfigure (a) repeats the chart of Figure 5.1. For the complete system charts, see Figures A.15 and A.16.

function	object file	CPU seconds
hash_get	libzebra.so.0.0.0	1.98
_int_malloc	libc-2.9.so	0.70
malloc_consolidate	libc-2.9.so	0.69
bgp_route_next	bgpd	0.49
bgp_best_selection	bgpd	0.34
calloc	libc-2.9.so	0.32
free	libc-2.9.so	0.26
_int_free	libc-2.9.so	0.25
bgp_scan_timer	bgpd	0.24
hash_release	libzebra.so.0.0.0	0.22

Table 5.5: Top ten functions called by bgpd, in terms of CPU utilization. Values reported are means over ten trials. Note that the hash_get function alone accounts for approximately 17% of the total CPU time used by bgpd during rehoming.

5.3 Optimizing bgpd

We now turn our attention to optimizing the CPU time used by bgpd. As with our optimization efforts for zebra, we begin by using OProfile to detail where CPU time is being spent. We then instrument bgpd to collect statistics validating the hypothesis suggested by the OProfile data. After confirming the hypothesis, we develop a patch to optimize the offending code, assess our improvements, and suggest opportunities for further optimization.

5.3.1 Finding the hot-spot

We present the top ten functions called by bgpd, in terms of CPU time used, as Table 5.5. As with zebra, we find a significant hot-spot in the code. Namely, the hash_get function accounts for 1.98 seconds of CPU time, or about 17% of the total CPU time used by bgpd during rehoming.

For more detail on where the time is being spent, we turn, once again, to line-by-line profiling data from OProfile. We present the data here as Listing 5.12. We observe that a single line of code, Line 16 of this listing, accounts for approximately 15% of the total CPU time used by bgpd. This line of code checks if a hash bucket contains the item that is being requested by the caller of hash_get. In order to carry out this task, the code first checks if the hash value of the bucket (expression backet->key) matches the hash value of the item specified by the caller (expression key). If the values match, the code proceeds with a full comparison of the bucket's key (expression backet->data), and the item's key (expression data).[7]

While the bottleneck could be either the initial check, or the call to the full comparison function, Table 5.5 suggests that the initial check is the culprit. Our reasoning is as follows. Observe that the full comparison function is addressed via a variable (in the expression (*hash->hash_cmp)). Because the function to be called is not known at compile time, the callee cannot be inlined. Hence, while the cost of jumping to the full comparison function might affect the CPU time attributed to this line of code, the actual execution time of that function will be accounted for separately. Thus,

[7]Note that the keys are opaque to the hash_get function, and extracted from the item data by the hash_cmp function specific to the given hash table. Hence the confusing reference to an item's key through a symbol named data.

```
       :/* Lookup and return hash backet in hash.  If there is no                1
       :   corresponding hash backet and alloc_func is specified, create new     2
       :   hash backet.  */                                                       3
       :void *                                                                    4
       :hash_get (struct hash *hash, void *data, void * (*alloc_func) (void *))   5
 0.1383 :{ /* hash_get total:  17151 16.5817 */                                   6
       :  unsigned int key;                                                       7
       :  unsigned int index;                                                     8
       :  void *newdata;                                                          9
       :  struct hash_backet *backet;                                             10
       :                                                                          11
 0.0164 :  key = (*hash->hash_key) (data);                                        12
       :  index = key % hash->size;                                               13
       :                                                                          14
 1.1186 :  for (backet = hash->index[index]; backet != NULL; backet = backet->next)  15
15.1847 :    if (backet->key == key && (*hash->hash_cmp) (backet->data, data))     16
 0.0087 :      return backet->data;                                               17
       :                                                                          18
 0.0271 :  if (alloc_func)                                                        19
       :    {                                                                     20
       :      newdata = (*alloc_func) (data);                                     21
 0.0048 :      if (newdata == NULL)                                                22
       :       return NULL;                                                       23
       :                                                                          24
 0.0068 :      backet = XMALLOC (MTYPE_HASH_BACKET, sizeof (struct hash_backet));  25
9.7e-04 :      backet->data = newdata;                                            26
 0.0174 :      backet->key = key;                                                  27
       :      backet->next = hash->index[index];                                  28
9.7e-04 :      hash->index[index] = backet;                                       29
 0.0019 :      hash->count++;                                                      30
 0.0039 :      return backet->data;                                                31
       :    }                                                                     32
       :  return NULL;                                                           33
 0.0512 :}                                                                        34
```

Listing 5.12: Line-by-line CPU time for `hash_get`, as called by `bgpd`. The number at the left of each line indicates the fraction of total program execution time attributed to that line. This listing reports data for the trial with the minimal outage time.

```
1         backet->next = hash->index[index];
2         hash->index[index] = backet;
3         hash->count++;
4         hash->high_count = MAX(hash->high_count, hash->count);
5         return backet->data;
6       }
7     return NULL;
```

Listing 5.13: Core source code for patch to capture hash table statistics. The complete patch is provided as Listings B.20–B.40.

name	items now	max items ever	primary buckets
cpu_record	15	15	1011
aspath	48482	48482	32767
attr	107591	107593	1024
baa (null) 1/1	0	53795	1024
baa (null) 1/1	0	1	1024
baa (null) 1/1	0	1	1024

Table 5.6: Hash table statistics for bgpd, at the completion of rehoming. The "attr" hash relates to path attributes received from peers, while the "baa" hashes are used for packing together outbound updates for prefixes which share path attributes. There is one "baa" hash for each peer and address family. The peer names appear as "(null)" due to a bug in our hash statistics patch. The address family "1/1" refers to IPv4 Unicast. Hash statistics for other address families are omitted here, as the item count is always zero. Due to a bug which prevented the capture of statistics for the trial with the minimal overall outage time, we present statistics from an alternate trial here.

such a function ought to appear in the top ten list of Table 5.5. Since none of the function names in Table 5.5 seem likely candidates for use as `hash_cmp`, we conclude that the full comparison functions are unlikely to be bottlenecks.

To confirm this hypothesis, we repeat our experiment, with some additional instrumentation. The core code for this new instrumentation, provided here as Listing 5.13, simply maintains a count of the maximum number of items ever present in each hash table instance. The data generated by this new instrumentation, provided here as Table 5.6, demonstrate that some of the hash tables in bgpd are significantly undersized. For example, the "attr" hash contains, at peak, over 100,000 items. Because the hash contains only 1024 hash buckets, a successful lookup would require, on average, searching approximately 50 items.

5.3.2 Resolving the hot-spot, and assessing our improvements

To remedy this problem, we patch bgpd to allocate a larger number of hash buckets for these two hash tables. For the "baa" hashes, the number of buckets is chosen to be slightly larger than the number of unique path attributes in our trace, as reported in Table 2.3. For the "attr" hash, we

```
static void
attrhash_init (void)
{
  attrhash = hash_create (attrhash_key_make, attrhash_cmp);
  /* NB: make the hash twice as large as the number of expected attributes,
     to account for copies made due to, e.g., next-hop-self.
     (mukesh.20100815) */
  attrhash = hash_create_size (BGP_ATTR_UNIQ_COUNT_EST * 2,
                               attrhash_key_make, attrhash_cmp);
}
```

Listing 5.14: Core source code, part 1 of 2, for patch to resolve the hot-spot in bgpd. This code increases the size of the hash table used to store received path attributes.

```
        FIFO_INIT (&sync->withdraw);
        FIFO_INIT (&sync->withdraw_low);
        peer->sync[afi][safi] = sync;
        peer->hash[afi][safi] = hash_create (baa_hash_key, baa_hash_cmp);
        if ((afi == AFI_IP) && (safi == SAFI_UNICAST)) {
          peer->hash[afi][safi] = hash_create_size (BGP_ATTR_UNIQ_COUNT_EST,
                                                    baa_hash_key, baa_hash_cmp);
        } else {
          peer->hash[afi][safi] = hash_create (baa_hash_key, baa_hash_cmp);
        }
      }
}
```

Listing 5.15: Core source code, part 2 of 2, for patch to resolve the hot-spot in bgpd. This code increases the size of the hash tables used when packing outbound advertisements for IPv4 Unicast addresses. The complete patch is provided as Listings B.42–B.44.

double this count to accommodate a detail of our bgpd configuration.[8] The core code for this patch is provided here as Listings 5.14, and 5.15.

To assess the impact of this change, we present Table 5.7, which compares memory utilization, CPU time, and outage time, before and after application of the patch. We find that the patch yields a 20% reduction in the CPU time used by bgpd, and a 19% reduction on overall outage time, at a cost of an approximately 1% increase in memory utilization.

While these improvements are welcome, the raw differences ("absolute change" in Table 5.7) do raise some questions. First, why has memory usage increased? Second, why is the reduction in CPU time used (2.46 seconds, according to Table 5.7) greater than the total time used by the `hash_get` function (1.98 seconds, as reported in Table 5.5)? Third, why is the improvement in overall downtime greater than the reduction in CPU time used? We consider each of these questions in turn.

[8] Specifically, our configuration uses the `next-hop-self` directive to modify the next-hop attribute of received routes, replacing the received value with the receiving router's own IP address. This requires the creation of a new path attribute for each received route.

	before resizing	with resizing	% change	absolute change
max resident set size (MiB)	117.42	118.41	+1	+0.99
total CPU seconds	12.93	10.47	−20	−2.46
overall outage seconds	15.86	12.91	−19	−2.95

Table 5.7: Cost of, and improvement due to, the resizing of hash tables. All measurements reported are mean values over ten trials.

A naïve explanation for the increased memory utilization is that it results directly from the increase in the number of hash buckets for the "attr" and "baa" hashes. Given that we now allocate 109,000 additional buckets when creating the "attr" hash, and an additional 54,000 buckets when creating each of the "baa" hashes listed above, and that each hash bucket consumes 12 bytes, we might expect an increase of 3.10 MiB.[9] However, this both, disagrees with the observed change, and neglects an important detail.

The neglected detail is that, if the additional hash buckets are not allocated at the time the hash table is created, they are allocated on demand, and chained in to the existing hash table via next pointers from the primary hash buckets for each hash value. Thus, our change only increases memory utilization for hash tables that did not previously exceed their initial allocation.

Focusing on the hash tables that do not exceed their initial allocation, specifically, the second and third "baa" hashes listed in Table 5.6, we expect memory use to increase by 108,000 buckets of 12 bytes each, or 1.24 MiB. This is within approximately 25% of the observed value. The remaining difference might be explained, for example, by the fact that the large initial allocation of hash buckets avoids the fragmentation that can occur with small dynamic memory allocations.

For the greater-than-expected reduction in CPU time, our measurements do not provide robust data with which to draw a conclusion. However, we speculate that the more compact layout of hash table buckets resulting from the large initial allocation may have reduced the cache footprint of the hash table, thereby improving the performance of code outside of the `hash_get` function. We leave verification of this hypothesis for future work.

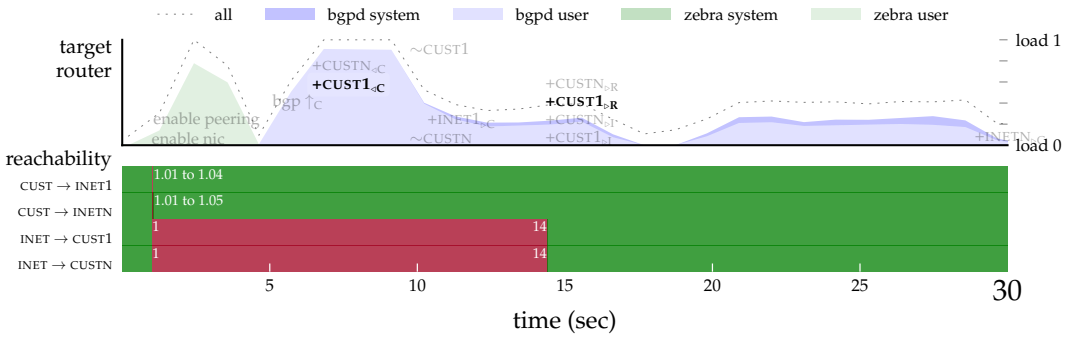
To understand why the reduction in outage time is greater than the reduction in CPU time, we consult Figure 5.4, which presents system charts for experiments with and without the increase in initial hash table size. Comparing the minimal trials for these conditions, we observe that, in the latter case, there is a shorter delay between the time at which routes are received from the customer router (**+cust1**$_{\ll}$), and the time when they are sent upstream, to the remote router (**+cust1**$_{\triangleright R}$).

It is unclear, from the data presented here, whether the difference in this route propagation delay is caused by the increase in hash table size, or due to other reasons. The difference might, for example, result from timer variation. In Chapter 6, we will more throughly investigate the cause of this delay, and how to reduce it.

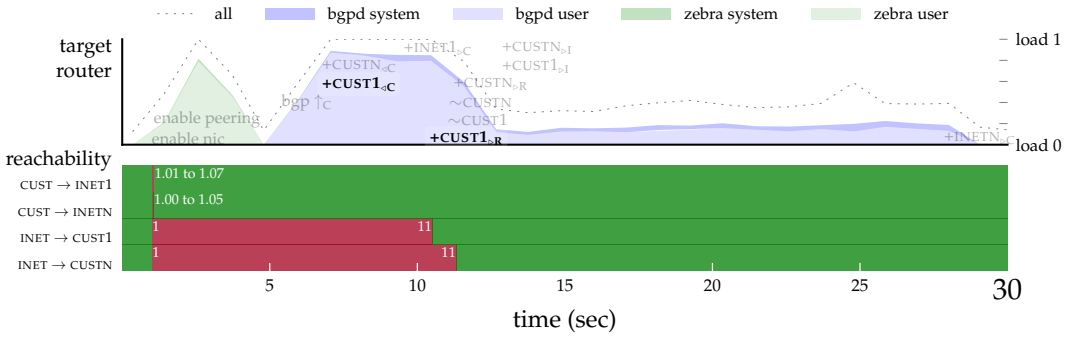
5.3.3 Avenues for improvement

We begin our search for further improvements with CPU profiling data. Specifically, we consult Table 5.8, which presents the top ten functions, in terms of CPU time used, after our patches to in-

[9] Each bucket contains an `unsigned int` key, a pointer to the data item, and a next pointer used for chaining overflow buckets. Thus, on the IA-32 platform, a single bucket consumes 12 bytes.



(a) default hash table sizing



(b) with resized hash tables

Figure 5.4: Partial system charts for rehoming with and without the hash table resizing patches of Listings 5.14 and 5.15, for the trials with the minimal overall outage times. For the complete system charts, see Figures A.17 and A.18.

function	object file	CPU seconds
`malloc_consolidate`	libc-2.9.so	0.79
`_int_malloc`	libc-2.9.so	0.70
`bgp_route_next`	bgpd	0.56
`bgp_best_selection`	bgpd	0.39
`calloc`	libc-2.9.so	0.32
`bgp_scan_timer`	bgpd	0.27
`_int_free`	libc-2.9.so	0.26
`free`	libc-2.9.so	0.25
`prefix_match`	libzebra.so.0.0.0	0.23
`bgp_process`	bgpd	0.22

Table 5.8: Top ten functions called by `bgpd`, in terms of CPU utilization, with resized hash tables. Values reported are means over ten trials.

crease the sizes of the "attr" and "baa" hashes. Therein, we observe that no single function accounts for more than 0.79 seconds of CPU time. Thus, we turn our focus away from micro-optimization, and consider broader optimizations instead.

To search for higher-level optimizations, we consult Figure 5.4(b). Examining the timeseries for the target router, we observe that a significant fraction of the outage time can be attributed to the delay between the **+cust1**$_{\triangleleft C}$ event, when the target router receives the first advertisement from the customer router, and the **+cust1**$_{\triangleright R}$ event, when the target router propagates this advertisement upstream to the remote router.

The reason for this delay is unclear. But given that the customer router advertises only nine routes, it is unlikely that the CPU time used during this interval is due principally to processing of the route advertisement from the customer, or the propagation of these advertisement upstream.[10] In Chapter 6, we will study what causes this delay, and what we might due to reduce it.

5.4 Conclusion

In this chapter, we set out to address the target router as a cause of downtime during rehoming. We chose, in particular, to focus on improvements that we could achieve by optimizing the code for `zebra` and `bgpd`.

For `zebra`, the path to optimization was intricate. We consulted a variety of data, including statistics on CPU use by function, CPU use by line of source code, log file messages, existing instrumentation of the work queue code in `zebra`, and new instrumentation of the same code. We inspected source code across three layers of scheduling code: the core scheduler, implemented in `thread_fetch`; the work queue scheduler, implemented in `work_queue_run`, and the meta-queue scheduler, implemented in `meta_queue_process`. The gains from these efforts were substantial: a 87% reduction in the CPU time used during rehoming, and a 32% reduction in mean overall outage time.

For `bgpd`, the road from profiling data to optimization was substantially shorter. We were able to proceed rapidly from the listing of top ten functions, by CPU time, to the profiler-annotated source

[10] For further details on the customer routes, see Table 2.3.

for `hash_get`, the single function consuming the most CPU time, to a patch with implemented new instrumentation to verify our hypothesis about the cause of the hot spot in `hash_get`. After developing a patch to address this hot spot, we noted a 20% reduction in the CPU time used during rehoming, and a 19% reduction in mean overall outage time.

With the benefit of both of these two micro-optimizations, we have reduced the mean overall outage time caused by rehoming from 23.18 seconds to 12.91 seconds. In the context of routing software upgrades, we can now say that keeping a router up-to-date, assuming the historical release schedule for Cisco IOS, and assuming that customers are moved to a temporary router, and back, would consume 40% of the annual outage budget. Were rehoming to be applied to other maintenance events as well, we could support seven such events while staying within the outage budget.

While these numbers are a marked improvement from our starting point, and could allow for five-nines reliability given the rate of software updates today, we would prefer a solution with more head room. This head room would be useful i) to cope with effects not observed in our laboratory setup, ii) to allow for a faster pace of software updates in the future, and iii) to permit a greater margin of error.

To provide this greater head room, we identified two possibilities for further improvement. In Section 5.2.5, we noted that there is a considerable amount of idle time for the CPU on the target router. In Section 5.3.3, we observed a significant delay between the receipt of routes from a customer, and the propagation of those routes upstream. In Chapter 6, we pursue improvements on both of these fronts.

Life is about timing.

Carl Lewis

6 On Timing

Thus far, we have employed a variety of techniques to reduce the down time caused by routing software upgrades, from approximately 140 seconds, to about 25 seconds.[1] Specifically, we began, in Chapter 3, by leveraging the level of indirection provided by reconfigurable transport networks to isolate the ISP customer from the effects of restarting the initial router. We continued our optimization efforts, in Chapter 4, by using Graceful Restart to eliminate unnecessary computation on the customer router. We then turned, in Chapter 5, to micro-optimizations to reduce the CPU time expended on the target router. We concluded Chapter 5 having exhausted the most promising micro-optimizations.

For further improvements, we turn now to scheduling. Our efforts here are guided by two insights from Chapter 5. First, we observed, in Section 5.2.5, that a significant fraction of the remaining downtime can be attributed to the delay between enabling the customer-facing network interface on the target router, and the establishment of the BGP peering session between the target router and the customer router. Second, we noted in, Section 5.3.3, that the delay between the time that the target router receives advertisements from the customer router, and the time that the target router propagates these routes to the remote router, is out of proportion with the very small number of routes originated by the customer router.

Based on these observations, we first address the delay in establishment of the BGP peering session between the target router, and the customer router. We demonstrate that this change reduces the down time during rehoming by 42%. We then focus on the delay between receiving and propagating customer routes. We develop a series of scheduling changes which further reduce the down time of rehoming by 53%.

The remainder of this chapter is structured as follows:

- In Section 6.2, we introduce the framework that we will use to evaluate the effectiveness of our changes. This framework decomposes the down time in to session establishment delay,

[1] We assume that our techniques are used to rehome a customer to a target router, and then, following the upgrade of the initial router, back to the initial router.

component	from	to	time (sec)
session establishment	outage start	**bgp** $\uparrow_c$	6.72
route reception	**bgp** $\uparrow_c$	**+custn**$_{\triangleleft c}$	1.61
route processing	**+custn**$_{\triangleleft c}$	$\sim$**custn**	1.77
route propagation	$\sim$**custn**	**+custn**$_{\triangleright r}$	2.73
other	**+custn**$_{\triangleright r}$	outage end	0.08

Table 6.1: Breakdown of outage time before any improvements. All values are means over ten trials. Because we detect the $\sim$**custn** event by polling once-per-second, we adjust the raw route processing value by subtracting 0.5 seconds, and adjust the raw route propagation value by adding 0.5 seconds, before reporting them here.

route reception delay, route processing delay, and route propagation delay.

- In Section 6.2, we identify the source of session establishment delay, develop a patch to address it, and demonstrate the effectiveness of our patch.
- In Section 6.3, we identify the source of route propagation delay, and develop a patch to address it. Observing that the patch does not actually improve route propagation delay, we hypothesize that this failure is due to deferred processing of routing updates.
- In Section 6.4, we confirm our hypothesis that routing updates are not processed immediately, and develop a patch to address the issue. We observe that the patch, in conjunction with the patch of Section 6.3, improves route propagation delay. We find that it does not, however, improve route processing delay. We conclude the section with a hypothesis that this failure is due to computation time required by other code in `bgpd`.
- In Section 6.5, we identify a likely cause of the hypothesized computation delay, and develop a patch to move that computation off of the critical path. We demonstrate the effectiveness of the patch, discuss alternative approaches, and then consider avenues for further improvement.
- In Section 6.6, we summarize and conclude. Our primary conclusion is that the patches of this chapter yield a viable rehoming solution. The solution greatly reduces the availability impact of routing software upgrades. And, if extended for use in other scenarios, the solution could provide five-nines availability even if a router were to fail, or be taken offline, once every 8 days.

6.1 Evaluation Framework

Our primary metric for judging the effectiveness of our optimizations is, of course, the down time observed by the customer during rehoming. However, in order to better understand the system behavior, and the means by which our optimizations affect that behavior, we introduce an additional framework for evaluating our optimizations.

In this additional framework, we decompose the downtime into a set of components. These components are: session establishment, route reception, route processing, route propagation, and "other". Each component is defined by a start event, and an end event. For an example of our decomposition, and the events defining the components, we present Table 6.1. This table decomposes the outage time prior to any scheduling optimizations.

With the exception of the "outage start" and "outage end" events, the events used in the decomposition are observed on the target router, and come from the set of events illustrated on our system charts, and enumerated in Table A.1. The "outage start" and "outage end" events reflect the time of the first missing ICMP packet, and the time of the packet after the last missing ICMP packet, respectively. The observation point for these events is the customer sink.

Note that because the ∼CUSTN event is based on polling for a change once-per-second, we adjust the route processing times by subtracting 0.5 seconds from our raw data, and we adjust the route propagation times by adding 0.5 seconds to our raw data. This compensates for the fact that, in expectation, we record the ∼CUSTN event 0.5 seconds after it occurs.

Before proceeding, we make two further observations. First, with the exception of the **bgp** $\uparrow_c$ event, all of the events defining the component-wise decomposition of down time are external to the routing software. Thus, they reflect an abstract model of BGP processing, rather than any implementation decisions specific to Quagga. Second, session establishment clearly dominates all other components of down time. Hence, it is a good starting point for the optimization efforts of this chapter.

6.2 Optimizing Session Establishment

Our goal, in this section, is to reduce the downtime caused by delays in the establishment of the peering session between the target router and the customer router. To do so, we analyze log file data to determine the cause of the delay, and develop a patch to address the problem. We then evaluate the effectiveness of our patch, and identify avenues for further improvement.

6.2.1 Finding the problem

In order to understand the cause of the problem, we consult log file data for the experiment with the minimal outage time. We examine all of the log messages generated by `bgpd` on the target router, between the start of the outage for traffic to CUSTN, and the successful establishment of the peering with the customer router. Therein, we find the following messages:

```
18:21:39.293833 BGP: Vty connection from 127.0.0.1
18:21:43.608685 BGP: %ADJCHANGE: neighbor 10.1.8.3 Up
```

The first log message denotes the time at which our experiment code logged in to `bgpd`, to enable the peering with the customer router. The second log message indicates when `bgpd` completed establishment of the new peering session. We observe a delay of 4.31 seconds between these two events. In order to gain further insight in to the cause of the delay, we repeat the experiment, but with all possible debugging options enabled for `bgpd`.

With debugging enabled, we examine the log messages from the start of the outage for traffic to CUSTN, until the target router has sent its BGP OPEN message to the customer router, for the trial with the minimal overall outage time. This subsequence of messages corresponds to 5.10 seconds of the 6.12 second session establishment delay observed in this trial.[2] We present the log messages in Listing 6.1.

Examining Listing 6.1, we focus on lines 4–8. We observe that `bgpd` does not attempt to establish the peering until the message at line 8, and that this occurs shortly after the expiration of the

[2]We omit the 36 messages corresponding to the remaining 1.02 seconds, for clarity of presentation.

```
1  10:38:26.360195 BGP: Zebra rcvd: router id update 10.1.8.2/32
2  10:38:26.360601 BGP: Zebra rcvd: interface eth2 address add 10.1.8.2/24
3  10:38:26.388306 BGP: Zebra rcvd: interface eth2 up
4  10:38:26.441180 BGP: Vty connection from 127.0.0.1
5  10:38:27.480817 BGP: Zebra rcvd: interface eth2 up
6  10:38:31.452902 BGP: 10.1.8.3 [FSM] Timer (start timer expire).
7  10:38:31.453058 BGP: 10.1.8.3 [FSM] BGP_Start (Idle->Connect)
8  10:38:31.453149 BGP: 10.1.8.3 [Event] Connect start to 10.1.8.3 fd 12
9  10:38:31.453323 BGP: 10.1.8.3 [FSM] Non blocking connect waiting result
10 10:38:31.453365 BGP: 10.1.8.3 went from Idle to Connect
11 10:38:31.459217 BGP: 10.1.8.3 [FSM] TCP_connection_open (Connect->OpenSent)
12 10:38:31.459330 BGP: 10.1.8.3 open active, local address 10.1.8.2
13 10:38:31.459474 BGP: 10.1.8.3 sending OPEN, version 4, my as 701, holdtime 180, id ...
```

Listing 6.1: Log file messages from bgpd, from the start of the outage for traffic to CUSTN, until bgpd has sent its BGP OPEN message to the customer router. For clarity of presentation, we omit messages relating to other peers. Ellipsis denote that a line has been truncated to fit the page width. These logfile messages come from the trial with the minimal overall outage time.

start timer, at line 6. The timer expiration itself comes 5.01 seconds after line 4, where our experiment script logs in to bgpd, to enable the peering with the customer router. This suggests that the session establishment delay is largely due to the start timer. Other trials further support this hypothesis, with the start timer accounting for a mean of 6.56 seconds out of a 8.13 second mean session establishment delay observed with debugging enabled.

6.2.2 Our Patch

Based on our examination of log file data above, we believe that bgpd does not attempt to establish a new peering session immediately after the operator enables the peering. Instead, bgpd sets a timer, and waits for the timer to expire before opening the peering session. The reason for this behavior is unclear, though we speculate that the delay might help avoid the premature establishment of a peering that it in the midst of being configured.[3]

To resolve this problem, we developed a patch which allows an operator to request that a BGP peering be established as soon as possible, rather than waiting for a timer. This patch provides similar functionality to the ManualStart event described in Sections 8.1.2 and 8.2.2 of the BGP-4 specification [69], and would not be required with a BGP implementation that supports the ManualStart event. We present the core of this patch in Listing 6.2.

Before proceeding to evaluate the effectiveness of this patch, we explain a subtlety of this patch, and its implication. Specifically, the patch does not directly call the function that establishes a new peering session. Instead, it sets the session establishment timer (`peer->v_start`) to zero, and generates an event to force the finite state machine for the BGP peering into the Stop state.

The reason for this implementation is that the current state of the finite state machine may not allow for the establishment of a new peering session. For example, if the router is already in the

[3]The Quagga CLI does not provide support for changing multiple configuration parameters atomically. Thus, one must first issue a command to create a peering, and then set each of the parameters for the peering one-by-one. Without the delay, bgpd might open a peering, only to reset it soon after, due to parameter changes.

```
int
peer_open (struct peer *peer)
{
  /* force back to idle, but set timer to zero, for immediate open */
  peer->v_start = 0;
  BGP_EVENT_ADD (peer, BGP_Stop);
  return 0;
}
```

Listing 6.2: Core source code for our patch to improve BGP session establishment time. This patch adds a new function `peer_open`, to `bgpd.c`. The complete patch is provided as Listings B.45–B.52.

	before patch	with patch
internet to CUST1	12.82	7.54
internet to CUSTN	12.90	7.54
customer to INET1	0.05	0.04
customer to INETN	0.05	0.04
any	12.91	7.55

Table 6.2: Comparison of mean outage times, in seconds, and over ten trials, before, and with, the patch of Listing 6.2.

midst of establishing a peering session at the time the operator issues the command to establish a peering, directly calling the session establishment code might confuse the router's state.

A consequence of this implementation is that when `bgpd` is in the Clearing state, as observed in Section 3.2.5, there may be a delay until the new peering can be established. This occurs because `bgpd` will not transition out of its Clearing state until it has finished cleaning up the state of the previous BGP session.

6.2.3 Evaluation and Avenues for Improvement

To evaluate the effectiveness of our patch, we present Tables 6.2 and 6.3. The former shows that our patch reduces the mean outage time of rehoming by 5.36 seconds, or 42%. The latter shows that, as expected, the improvement in down time comes from a reduction in the time required to establish the new peering session.

Table 6.3 also provides guidance for further optimizations. We observe that, while session establishment does require a non-negligible amount of time, route propagation accounts for a significantly larger proportion of the total down time. Specifically, while session establishment requires 1.69 seconds, route propagation 86% more time, or 3.14 seconds. Accordingly, we next shift our focus to route propagation delay.

component	from	to	before patch	with patch
session establishment	outage start	**bgp** $\uparrow_c$	6.72	1.69
route reception	**bgp** $\uparrow_c$	**+custn**$_{\triangleleft c}$	1.61	1.36
route processing	**+custn**$_{\triangleleft c}$	$\sim$**custn**	1.77	1.28
route propagation	$\sim$**custn**	**+custn**$_{\triangleright r}$	2.73	3.14
other	**+custn**$_{\triangleright r}$	outage end	0.08	0.08

Table 6.3: Breakdown of outage time after application of the patch in Listing 6.2. All values are means over ten trials. Because we detect the $\sim$**custn** event by polling once-per-second, we adjust the raw route processing value by subtracting 0.5 seconds, and adjust the raw route propagation value by adding 0.5 seconds, before reporting them here. We repeat the data of Table 6.1, in the "before patch" column, for ease of reference.

6.3 Optimizing Route Propagation

Having optimized session establishment, we found that the largest remaining bottleneck was due to route propagation. Accordingly, we now focus our efforts on optimizing route propagation. We first examine log file data to determine the cause of route propagation, which is that route propagation is driven by periodic timers. We then develop a patch to address the problem, by transmitting routing updates as soon as the customer router has sent its End-of-RIB marker to the target router.[4]

Unfortunately, in our evaluation of the patch, we find that it does not directly improve route propagation times. Using log file data to diagnose this failure, we find that the code from our patch to accelerate route propagation likely runs before routing updates have been processed. We conclude that the patch itself is likely correct, but that its effect will be seen only after addressing route processing delay.

6.3.1 Finding the problem

Because the events that define our decomposition of outage time, as well as the events on our system charts, are generally based on observations external to Quagga, they do not provide the detail necessary to understand the source of route propagation delay within `bgpd`. Accordingly, we turn instead to log file data.

Taking a similar approach to our efforts towards understanding `zebra` in Section 5.2.2, we repeat our experiment with all possible debugging options enabled for `bgpd`. This generates a log file of about 20 MB, containing approximately 316,000 log messages. To reduce this data to a manageable size, we extract those messages most directly related to $\sim$**custn** and **+custn**$_{\triangleright r}$, the events defining route propagation delay.

To gather messages related to $\sim$**custn**, we extract messages from `bgpd` to `zebra`, as `bgpd` uses `zebra` to effect changes to the kernel FIB. To gather messages relating to **+custn**$_{\triangleright r}$, we extract messages that contain the IP address of the remote router. We then truncate the data, removing messages before the first message to `zebra`, and after the last BGP UPDATE transmitted to the remote

[4]Note that this was not feasible prior to the development of Graceful Restart, as the End-of-RIB marker was introduced as part of Graceful Restart [73].

```
05:34:34.253537 BGP: Zebra send: IPv4 route add 128.2.0.0/16 nexthop 10.1.8.3 metric 0
05:34:34.254096 BGP: Zebra send: IPv4 route add 209.129.244.0/23 nexthop 10.1.8.3 ...
05:34:34.254300 BGP: Zebra send: IPv4 route add 204.194.28.0/22 nexthop 10.1.8.3 ...
05:34:34.254446 BGP: Zebra send: IPv4 route add 192.80.210.0/24 nexthop 10.1.8.3 ...
05:34:34.254591 BGP: Zebra send: IPv4 route add 192.58.107.0/24 nexthop 10.1.8.3 ...
05:34:34.254731 BGP: Zebra send: IPv4 route add 192.12.32.0/24 nexthop 10.1.8.3 ...
05:34:34.254882 BGP: Zebra send: IPv4 route add 128.237.0.0/16 nexthop 10.1.8.3 ...
05:34:35.117124 BGP: 10.1.6.3 [FSM] Timer (routeadv timer expire)
05:34:35.119214 BGP: 10.1.6.3 send UPDATE 128.2.0.0/16
05:34:35.119251 BGP: 10.1.6.3 send UPDATE 128.237.0.0/16
05:34:35.119283 BGP: 10.1.6.3 send UPDATE 192.12.32.0/24
05:34:35.119315 BGP: 10.1.6.3 send UPDATE 192.58.107.0/24
05:34:35.119347 BGP: 10.1.6.3 send UPDATE 192.80.210.0/24
05:34:35.119378 BGP: 10.1.6.3 send UPDATE 204.194.28.0/22
05:34:35.119411 BGP: 10.1.6.3 send UPDATE 209.129.244.0/23
```

Listing 6.3: Log file messages from bgpd, relating to the $\sim$CUSTN and +CUSTN$_{\triangleright R}$ events. Ellipsis denote that a line has been truncated to fit the page width. These logfile messages come from the trial with the minimal overall outage time.

router. We present the resulting data, for the trials with the minimal and maximal outage times, in Listings 6.3 and 6.4, respectively.

Our primary observation from the data of these listings is that, in both cases, the BGP UPDATEs are sent to the remote router after the expiration of the routeadv timer. In the case of the trial with the shortest overall outage time, this timer expires approximately 850ms after bgpd instructs zebra to add a route for CUSTN. For the trial with the longest overall outage time, the routeadv timer expires twice before updates are transmitted to the remote router. The first timer expiration occurs 1ms after bgpd instructs zebra to add the route for CUSTN, while the second expiration occurs an additional 5 seconds later.

Based on correlation of the routeadv timer with the generation of BGP UPDATEs, and the fact that the timer interval matches the 5 seconds suggested for the default value of the MinRouteAdvertisementIntervalTimer in the BGP specification [69], we infer that the routeadv timer in bgpd implements the limitation on routing update frequency described in Section 9.2.1.1 of the RFC defining BGP-4 [69].

6.3.2 Our Patch

Based on our analysis of log file messages, we believe that the route propagation delay we have observed can be attributed to the need to wait for a timer event before transmitting route advertisements to the remote router. This is likely due to a feature of BGP intended to limit the frequency of routing updates. While this is appropriate for autonomous operation, we would like a deliberate action like rehoming to proceed more quickly.

To allow rehoming to proceed more quickly, we developed a patch which transmits routing advertisements without waiting for the routeadv timer. We present the core code of that patch here, as Listing 6.5. This patch alters the receive path for BGP updates, in bgp_update_receive. Specifically, when bgp_update_receive function receives and End-of-RIB marker from the customer

```
05:20:21.837676 BGP: Zebra send: IPv4 route add 128.2.0.0/16 nexthop 10.1.8.3 metric 0
05:20:21.837968 BGP: Zebra send: IPv4 route add 209.129.244.0/23 nexthop 10.1.8.3 ...
05:20:21.838134 BGP: Zebra send: IPv4 route add 204.194.28.0/22 nexthop 10.1.8.3 ...
05:20:21.838473 BGP: Zebra send: IPv4 route add 192.80.210.0/24 nexthop 10.1.8.3 ...
05:20:21.838684 BGP: Zebra send: IPv4 route add 192.58.107.0/24 nexthop 10.1.8.3 ...
05:20:21.838985 BGP: Zebra send: IPv4 route add 192.12.32.0/24 nexthop 10.1.8.3 ...
05:20:21.839174 BGP: Zebra send: IPv4 route add 128.237.0.0/16 nexthop 10.1.8.3 ...
05:20:21.839337 BGP: 10.1.6.3 [FSM] Timer (routeadv timer expire)
05:20:26.843757 BGP: 10.1.6.3 [FSM] Timer (routeadv timer expire)
05:20:26.846585 BGP: 10.1.6.3 send UPDATE 128.2.0.0/16
05:20:26.846620 BGP: 10.1.6.3 send UPDATE 128.237.0.0/16
05:20:26.846652 BGP: 10.1.6.3 send UPDATE 192.12.32.0/24
05:20:26.846685 BGP: 10.1.6.3 send UPDATE 192.58.107.0/24
05:20:26.846717 BGP: 10.1.6.3 send UPDATE 192.80.210.0/24
05:20:26.846750 BGP: 10.1.6.3 send UPDATE 204.194.28.0/22
05:20:26.846782 BGP: 10.1.6.3 send UPDATE 209.129.244.0/23
```

Listing 6.4: Log file messages from `bgpd`, relating to the ∼CUSTN and +CUSTN$_{\triangleright R}$ events. Ellipsis denote that a line has been truncated to fit the page width. These logfile messages come from the trial with the maxmimal overall outage time.

	before patch	with patch
internet to CUST1	7.54	8.26
internet to CUSTN	7.54	8.26
customer to INET1	0.04	0.04
customer to INETN	0.04	0.05
any	7.55	8.27

Table 6.4: Comparison of mean outage times, in seconds, and over ten trials, before, and with, the patch of Listing 6.5. Data for the before case are copied from Table 6.2.

router, our patch checks the peer structure for each peer, to determine if there are any pending updates for that peer. If so, the patch calls `_bgp_write` to transmit those messages immediately.

6.3.3 Evaluation

To evaluate the effectiveness of our patch, we consider the overall outage time, and its component-wise breakdown, which we present in Tables 6.4 and 6.5, respectively. Surprisingly, we find that the mean overall outage time has increased by 720ms, and that the bulk of the increase in outage time occurs in the route processing.

We consider two potential explanations for the increase in outage time. Specifically, the increase might be due to a greater CPU demand by `bgpd` during rehoming, or it might be due to variation in the time that `bgpd` waits for the `routeadv` timer to expire. Because we can only measure timer variation for experiments with debugging enabled, we begin with an examination of CPU demand.

To test if our patch has increased CPU demand, we examine the CPU time used by `bgpd`, `zebra`,

```
if (! attribute_len && ! withdraw_len)
  {
    struct listnode *node, *nnode;
    struct peer *p;

    /* End-of-RIB received */
    SET_FLAG (peer->af_sflags[AFI_IP][SAFI_UNICAST],
              PEER_STATUS_EOR_RECEIVED);

    /* NSF delete stale route */
    if (peer->nsf[AFI_IP][SAFI_UNICAST])
      bgp_clear_stale_route (peer, AFI_IP, SAFI_UNICAST);

    for (ALL_LIST_ELEMENTS (peer->bgp->peer, node, nnode, p))
      {
        struct thread t;
        time_t oldsync;

        oldsync = p->synctime;
        t.arg = p;

        p->synctime = bgp_clock() + 1;
        if (bgp_write_proceed(p))
          {
            /* cancel any pending write thread, since we're taking
               care of writes here. (mukesh.20100819). */
            BGP_WRITE_OFF(p->t_write);
            _bgp_write(&t, 0);
          }
        p->synctime = oldsync;
      }

    if (BGP_DEBUG (normal, NORMAL))
      zlog (peer->log, LOG_DEBUG, "rcvd End-of-RIB for IPv4 Unicast from %s",
            peer->host);
  }
```

Listing 6.5: Core source code for our patch to improve route propagation delay. This patch modifies bgp_update_receive in bgp_packet.c. The complete patch is provided as Listings B.53–B.54.

component	from	to	before patch	with patch
session establishment	outage start	**bgp** $\uparrow_c$	1.69	1.78
route reception	**bgp** $\uparrow_c$	**+custn**$_{\lhd c}$	1.36	1.35
route processing	**+custn**$_{\lhd c}$	$\sim$**custn**	1.28	1.83
route propagation	$\sim$**custn**	**+custn**$_{\rhd r}$	3.14	3.22
other	**+custn**$_{\rhd r}$	outage end	0.08	0.08

Table 6.5: Breakdown of outage time after application of the patch in Listing 6.5. All values are means over ten trials. Because we detect the $\sim$**custn** event by polling once-per-second, we adjust the raw route processing value by subtracting 0.5 seconds, and adjust the raw route propagation value by adding 0.5 seconds, before reporting them here. We repeat the data of Table 6.3, in the "before patch" column, for ease of reference.

	before patch	with patch
`bgpd`	0.51	0.59
`zebra`	0.00	0.10
other processes	0.77	1.13
idle time	0.00	0.02
total	1.28	1.84

Table 6.6: Comparison of CPU time used by routing processes before and after application of the patch in Listing 6.5. Values reported are in seconds, and are mean times over ten trials.

and other processes, between the **+custn**$_{\lhd c}$ and $\sim$**custn** events. We present the data in Table 6.6. From the data in the table, we observe that both `bgpd` and `zebra` exhibit some increase in their CPU utilization during route processing.

The bulk of the increase, however, comes from other processes. Because other processes in the system should not be affected by the transmission of BGP updates from `bgpd`, we proceed under the assumption that the increase we observe in CPU time used during route processing is due primarily to experimental variation.

Even with our assumption that the increase in route processing time is due to experimental variation, an important question remains. Namely, why does the patch fail to accelerate route propagation? To answer this question, we repeat our experiment with all possible debugging options enabled, and then consult the log file data.

When examining the log file data, however, we make one important change. In addition to messages logging communication with `zebra`, and messages containing the IP address of the remote router, we also extract lines matching the message generated by lines 33– 35 of Listing 6.5. Because these lines execute after our new code, seeing the corresponding log file message will assure us that our code was reached.

We present the extracted data in Listing 6.6. Focusing on lines 1 and 2 of this listing, we observe that our new code must have executed before `bgpd` instructed `zebra` to install routes for the prefixes advertised by the customer. This suggests that, at the time our code is reached, `bgpd` has not yet processed the routing updates from the customer router. Accordingly, we set aside our concern

```
10:23:38.198513 BGP: rcvd End-of-RIB for IPv4 Unicast from 10.1.8.3                    1
10:23:38.198858 BGP: Zebra send: IPv4 route add 128.2.0.0/16 nexthop 10.1.8.3 metric 0  2
10:23:38.199121 BGP: Zebra send: IPv4 route add 209.129.244.0/23 nexthop 10.1.8.3 ...   3
10:23:38.199320 BGP: Zebra send: IPv4 route add 204.194.28.0/22 nexthop 10.1.8.3 ...    4
10:23:38.199474 BGP: Zebra send: IPv4 route add 192.80.210.0/24 nexthop 10.1.8.3 ...    5
10:23:38.199613 BGP: Zebra send: IPv4 route add 192.58.107.0/24 nexthop 10.1.8.3 ...    6
10:23:38.199748 BGP: Zebra send: IPv4 route add 192.12.32.0/24 nexthop 10.1.8.3 ...     7
10:23:38.199885 BGP: Zebra send: IPv4 route add 128.237.0.0/16 nexthop 10.1.8.3 ...     8
10:23:38.200036 BGP: 10.1.6.3 [FSM] Timer (routeadv timer expire)                       9
10:23:38.200655 BGP: 10.1.6.3 send UPDATE 128.2.0.0/16                                  10
10:23:38.200726 BGP: 10.1.6.3 send UPDATE 128.237.0.0/16                                11
10:23:38.200758 BGP: 10.1.6.3 send UPDATE 192.12.32.0/24                                12
10:23:38.200790 BGP: 10.1.6.3 send UPDATE 192.58.107.0/24                               13
10:23:38.200822 BGP: 10.1.6.3 send UPDATE 192.80.210.0/24                               14
10:23:38.200854 BGP: 10.1.6.3 send UPDATE 204.194.28.0/22                               15
10:23:38.200885 BGP: 10.1.6.3 send UPDATE 209.129.244.0/23                              16
```

Listing 6.6: Log file messages from bgpd, relating to the $\sim$CUSTN and +CUSTN$_{\text{DR}}$ events, following application of the patch in Listing 6.5. Ellipsis denote that a line has been truncated to fit the page width. These logfile messages come from the trial with the minimal overall outage time.

about route propagation delay for now, and focus instead of route processing delay.

6.4 Optimizing Route Processing, Part I

In the previous section, we attempted to optimize route propagation delays. We analyzed the source of the problem, and then developed and tested a patch to address it. After finding the patch to be ineffective, we again analyzed the system behavior, and concluded that an effective patch for improving route propagation delay depends on first addressing route processing delay.

Accordingly, in this section, we attempt to address route processing delay. We first analyze the source code of bgpd, to locate the source of route processing delays. From this analysis, we conclude that rather than processing updates immediately as they are received, bgpd queues the updates on to a work queue, for background processing. We then develop a patch to address the problem. The key idea of the patch is to process any pending updates from a peer immediately after the peer has sent its End-of-RIB marker.[5]

Empirically, we find that our patch does improve mean overall outage time. Investigating the source of the improvement, we find that, as expected, route propagation time improves after the application of our patch. This validates our hypothesis that the failure of our route propagation patch alone was due to the fact that the routing updates from the customer router had not been processed at the time that the code of Listing 6.5 executed.

Surprisingly, however, we find that our patch fails to improve route processing time. To understand why, we consult log file data, and identify a discrepancy between our external observations, and the events reported by bgpd. Based on this discrepancy, and CPU utilization data from a sys-

[5]As in Section 6.3, we note that this was not feasible prior to the development of Graceful Restart, as the End-of-RIB marker was introduced as part of Graceful Restart [73].

tem chart, we conclude that the route processing delay is due to bgpd performing other tasks before processing the routing updates from its peer. We leave identification of those tasks, however, for the next section.

6.4.1 Finding the problem, and our patch

In order to locate the problem, we studied the code of `bgp_update_receive`, to determine why the routing updates from the customer router were not processed by the time the End-of-RIB message was received. We found that `bgp_update_receive` passes the routing update to `bgp_nlri_parse`, which leads, after a chain of three more function calls, to `bgp_process`.[6]

Unfortunately, although the name might suggest otherwise, `bgp_process` does not immediately process a routing update. Instead, as shown at line 30 of Listing 6.7, this function simply enqueues the routing update on to a work queue. To resolve this problem, our patch simply services the relevant work queue after the customer router has sent its End-of-RIB marker, but before the code from our patch of Listing 6.5. The core code for this patch is provided here as Listing 6.8.

6.4.2 Evaluation

To evaluate the effects of this patch, we again consider the overall outage time, and its component-wise breakdown. We present these measurements in Tables 6.7 and 6.8, respectively. The former table brings us the good news that our patch has reduced outage time by 2.75 seconds over the previous patch. The latter table brings us two pieces of news, as we next elaborate.

The first bit of news from Table 6.8 is that processing routing updates upon the receipt of the End-of-RIB message from the customer router has, in combination with the patch of Listing 6.5, eliminated route propagation delay. This is consistent with our hypothesis in Section 6.3.3, that the reason bgpd did not transmit routing updates sooner was that the updates had not yet been processed.

The second bit of news is surprising, and less pleasant. It appears that our patch to reduce route processing delay has failed to achieve its goal. In fact, route processing delay is longer with this patch, than before the patch was applied. While we could ask why this delay has increased, the more pressing question at hand is why a delay exists at all. Accordingly, we next turn to understanding the cause of this delay.

6.4.3 Diagnosis

To understand why we observe a route processing delay after our patch of Listing 6.8, we examine the full set of log file messages generated by bgpd on the target router, between the time when bgpd reports that the peering session with the customer router is established, and the time when it reports that it has updated the kernel routing table entry for CUSTN.

We present this sequence of log messages, for the trial with the minimal outage time, in Listing 6.9. From these messages, we make two important observations. First, examining lines 1 and 13, we observe a route reception delay of almost 3.2 seconds. This is much longer than the 1.61 second

[6]The full call chain, starting from `bgp_update_receive`, is `bgp_nlri_parse`, `bgp_update`, `bgp_update_main`, and then `bgp_process`.

```
void
bgp_process (struct bgp *bgp, struct bgp_node *rn, afi_t afi, safi_t safi)
{
  struct bgp_process_queue *pqnode;

  /* already scheduled for processing? */
  if (CHECK_FLAG (rn->flags, BGP_NODE_PROCESS_SCHEDULED))
    return;

  if ( (bm->process_main_queue == NULL) ||
       (bm->process_rsclient_queue == NULL) )
    bgp_process_queue_init ();

  pqnode = XCALLOC (MTYPE_BGP_PROCESS_QUEUE,
                    sizeof (struct bgp_process_queue));
  if (!pqnode)
    return;

  /* all unlocked in bgp_processq_del */
  bgp_table_lock (rn->table);
  pqnode->rn = bgp_lock_node (rn);
  pqnode->bgp = bgp;
  bgp_lock (bgp);
  pqnode->afi = afi;
  pqnode->safi = safi;

  switch (rn->table->type)
    {
      case BGP_TABLE_MAIN:
        work_queue_add (bm->process_main_queue, pqnode);
        break;
      case BGP_TABLE_RSCLIENT:
        work_queue_add (bm->process_rsclient_queue, pqnode);
        break;
    }

  return;
}
```

Listing 6.7: Source code of `bgp_process`, from `bgp_route.c`. This function is called during the execution of `bgp_update_receive`.

```
1       if (! attribute_len && ! withdraw_len)
2         {
3           struct listnode *node, *nnode;
4           struct peer *p;
5           struct thread t;
6           int res;
7
8           /* End-of-RIB received */
9           SET_FLAG (peer->af_sflags[AFI_IP][SAFI_UNICAST],
10                    PEER_STATUS_EOR_RECEIVED);
11
12          /* NSF delete stale route */
13          if (peer->nsf[AFI_IP][SAFI_UNICAST])
14            bgp_clear_stale_route (peer, AFI_IP, SAFI_UNICAST);
15
16          res = WQ_SUCCESS;
17          t.arg = bm ? bm->process_main_queue : NULL;
18          while (bm && bm->process_main_queue && bm->process_main_queue->items
19                 && listcount(bm->process_main_queue->items) &&
20                 (res == WQ_SUCCESS))
21            res = work_queue_run(&t);
22
23          for (ALL_LIST_ELEMENTS (peer->bgp->peer, node, nnode, p))
24            {
```

Listing 6.8: Core source code for our first patch to improve route processing delay. This patch modifies bgp_update_receive in bgp_packet.c. The complete patch is provided as Listings B.55–B.57.

	before patch	with patch
internet to CUST1	8.26	5.51
internet to CUSTN	8.26	5.52
customer to INET1	0.04	0.05
customer to INETN	0.05	0.05
any	8.27	5.52

Table 6.7: Comparison of mean outage times, in seconds, and over ten trials, before, and with, the patch of Listing 6.8. Data for the before case are copied from Table 6.4.

component	from	to	before patch	with patch
session establishment	outage start	**bgp** $\uparrow_{c}$	1.78	1.74
route reception	**bgp** $\uparrow_{c}$	**+CUSTN**$_{\triangleleft c}$	1.35	1.61
route processing	**+CUSTN**$_{\triangleleft c}$	$\sim$**CUSTN**	1.83	2.14
route propagation	$\sim$**CUSTN**	**+CUSTN**$_{\triangleright R}$	3.22	-0.05
other	**+CUSTN**$_{\triangleright R}$	outage end	0.08	0.07

Table 6.8: Breakdown of outage time after application of the patch in Listing 6.8. All values are means over ten trials. Because we detect the $\sim$**CUSTN** event by polling once-per-second, we adjust the raw route processing value by subtracting 0.5 seconds, and adjust the raw route propagation value by adding 0.5 seconds, before reporting them here. We repeat the data of Table 6.5, in the "before patch" column, for ease of reference.

component	from	to	external data	`bgpd` log data
session establishment	outage start	**bgp** $\uparrow_{c}$	1.39	1.39
route reception	**bgp** $\uparrow_{c}$	**+CUSTN**$_{\triangleleft c}$	1.32	3.19
route processing	**+CUSTN**$_{\triangleleft c}$	$\sim$**CUSTN**	1.71	0.29
route propagation	$\sim$**CUSTN**	**+CUSTN**$_{\triangleright R}$	0.45	0.00
other	**+CUSTN**$_{\triangleright R}$	outage end	0.06	0.06

Table 6.9: Comparison of breakdown of outage times, using external and internal observations, for our first patch for improving route processing delay, and with full logging enabled. The tables presents data for the trial with the minimal overall outage time. Because we detect the $\sim$**CUSTN** event by polling once-per-second, we adjust the raw route processing value by subtracting 0.5 seconds, and adjust the raw route propagation value by adding 0.5 seconds, before reporting them here.

mean route reception delay of Table 6.8. Second, examining lines 13 and 21, we observe a route processing delay of 290 milliseconds. This is much shorter than the 2.14 second mean route processing delay of Table 6.8. We now investigate these surprises more deeply.

A number of factors might explain the discrepancy between the delays observed in the log file data of Listing 6.9 and the data computed from external observations in Table 6.8. First, because the data in Table 6.8 come from experiments without logging enabled, the difference might be due to perturbation caused by logging. Second, because the table reports mean data over multiple trials, whereas the log file comes from a single trial, the difference might be due to experimental variation.

To eliminate both of these possibilities, we compute the component-wise breakdown of outage time for the trial whose log file data is shown in Listing 6.9. We present this breakdown as Table 6.9. Examining the route reception and route propagation delays reported in this table, we observe that the discrepancies remain. That is, even when comparing external and logfile data for a single trial with debugging enabled, the route reception delay from the log messages is higher than that from the external observations, and the route processing delay is lower than that from the external observations.

The existence of a discrepancy between these two sources of data is not, in itself, cause for

```
12:45:56.714471 BGP: %ADJCHANGE: neighbor 10.1.8.3 Up
12:45:56.714513 BGP: 10.1.8.3 sending KEEPALIVE
12:45:56.714537 BGP: 10.1.8.3 send message type 4, length (incl. header) 19
12:45:59.905761 BGP: Import timer expired.
12:45:59.906387 BGP: 10.1.8.3 rcv message type 4, length (excl. header) 0
12:45:59.906441 BGP: 10.1.8.3 KEEPALIVE rcvd
12:45:59.906777 BGP: 10.1.2.2 [FSM] Timer (routeadv timer expire)
12:45:59.906895 BGP: [AS4SEG] Parse aspath segment: got total byte length 6
12:45:59.906927 BGP: [AS4SEG] Parse aspath segment: got type 2, length 1
12:45:59.906962 BGP: [AS4SEG] Parse aspath segment: Bytes now: 6
12:45:59.907026 BGP: 10.1.8.3 rcvd UPDATE w/ attr: nexthop 10.1.8.3, origin i, path 9
12:45:59.907790 BGP: 10.1.8.3 rcvd 128.2.0.0/16
12:45:59.907919 BGP: 10.1.8.3 rcvd 209.129.244.0/23
12:45:59.907974 BGP: 10.1.8.3 rcvd 204.194.28.0/22
12:45:59.908026 BGP: 10.1.8.3 rcvd 192.80.210.0/24
12:45:59.908077 BGP: 10.1.8.3 rcvd 192.58.107.0/24
12:45:59.908128 BGP: 10.1.8.3 rcvd 192.12.32.0/24
12:45:59.908178 BGP: 10.1.8.3 rcvd 128.237.0.0/16
12:45:59.908369 BGP: 10.1.6.3 [FSM] Timer (routeadv timer expire)
12:46:00.198020 BGP: Zebra send: IPv4 route add 128.2.0.0/16 nexthop 10.1.8.3 metric 0
12:46:00.198532 BGP: Zebra send: IPv4 route add 209.129.244.0/23 nexthop 10.1.8.3 ...
```

Listing 6.9: Log file messages from bgpd, between the $+\textsc{custn}_{\ll c}$ and $\sim\textsc{custn}$ events. Ellipsis denote that a line has been truncated to fit the page width. These logfile messages come from the trial with the minimal overall outage time

concern. But it is important for us to understand the cause of the difference. In particular, given that the TCP stack receives the packet containing the advertisement for CUSTN at some time x, why is it that bgpd does not report receiving the advertisement until 1.87 seconds later?

We consider two possible explanations for the delay between the advertisement being received by the TCP stack, and the processing of the advertisement by bgpd. First, the delay might be due to a bgpd idling while waiting for a timer event. Second, the delay might be due to computation delays. To determine which is more likely, we further consult the log file data.

Examining Listing 6.9 again, we focus on the messages between between session establishment, at line 1, and route reception, at line 13. We observe a gap of 3.19 seconds between lines 3 and 4. This accounts for the essentially the entire route reception delay reported in the log file. Given that line 4 reports a timer expiration event, the log file data suggests that the delay between receipt of the advertisement by the TCP stack, and its processing by bgpd is due to timer delays.

Before attempting to address this problem, however, we check the data from the other trials of this experiment. Rather than examine the log file entries for each of these other nine trials manually, we examine them in aggregate by computing the mean elapsed time between the establishment of the BGP session with the customer, and the expiration of the import timer. We then compute the mean route reception delay for all trials with debugging enabled, and compare this with the mean import timer expiration delay.

We find that the mean import timer expiration delay is 6.57 seconds, and that the mean route reception delay from log file data is 4.19 seconds. Given that the mean route reception delay is lower than the mean import timer delay, our theory that route reception happens only after expiration of the import timer does not hold. We conclude that bgpd can, and does, receive routes independently of the import timer.

To test our alternate theory, that route reception by bgpd is delayed due to computation time, we consult the system chart for this same experiment. We present a partial system chart for the trial with the minimal outage time here as Figure 6.1. Note that in addition to the events depicted in our other system charts, this chart has the event **B+CUSTN**$_{\ll c}$, denoting when the advertisement for CUSTN is received by bgpd.

Examining the system chart, we observe that, during the time between the **bgp** $\uparrow_c$ and **B+CUSTN**$_{\ll c}$ events on this chart, the processor on the target router is fully utilized. We further note that bgpd accounts for one-half or more of the load during this interval. This provides support for our second theory, that the delay between lines 3 and 4 of Listing 6.9 is due to computation time. Accordingly, in the next section, we will locate the source of this computation, and determine whether it can be removed from the critical path.

6.5 Optimizing Route Processing, Part II

In Section 6.4, we introduced a patch to reduce route processing delays during rehoming, but found that it did not improve route processing delays. Thereafter, we diagnosed the problem, finding that there is a delay between the time that the TCP stack receives a route advertisement from the customer router, and the time when bgpd receives that advertisement. We further found that this delay was due to computation time. Accordingly, our goal in this section is to locate this computation, and to remove it from the critical path for rehoming.

In order to achieve our goal, we first consult source code, and log file data to determine where the CPU time is being spent. We then develop a patch which defers this computation until after

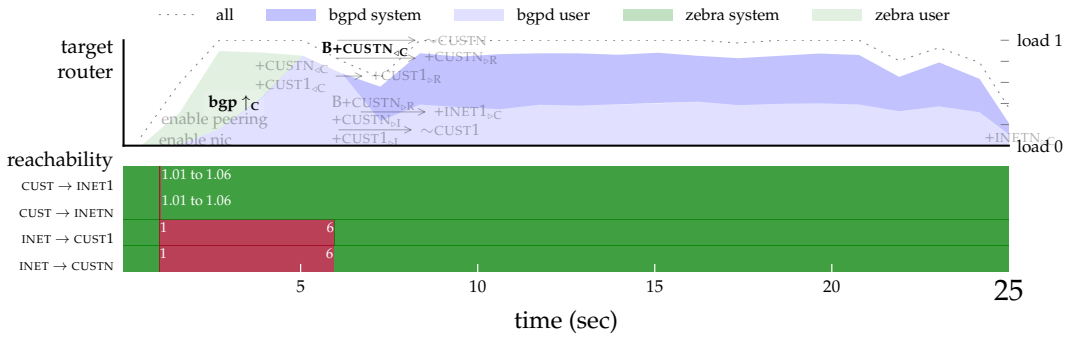


Figure 6.1: Partial system chart following the application of the patch in Listing 6.8. Note that full debugging is enabled for bgpd. This chart presents the the trial with the minimal overall outage time. For the complete system chart, see Figure A.19.

the routing advertisements from the customer router have been processed. After demonstrating that the patch achieves its goal, we discuss alternative approaches, and, finally, consider avenues for further improvement.

Before proceeding, we note that, whether we classify the problem as route reception delay or route processing delay depends on our vantage point. From external observations, the delay is best classified as route processing delay, while from the bgpd log messages, the delay is best classified as route reception delay. For consistency with the rest of this chapter, we classify the delay based on external observations, and refer to it as route processing delay.

6.5.1 Finding the problem, and our patch

In order to locate the source of the delay, we consult the source code for bgpd, using log file messages to guide our search. In particular, we recall, from Section 6.4.3, that much of the delay that we seek to address can be explained by the gap of 3.19 seconds between lines 3 and 4 of Listing 6.9. Having observed, in Section 6.4.3, that line 4 does not well correlate with observed delays, we focus on locating code that generate the message of line 3.

A quick grep of the source code reveals that the message of line 3 might be generated by five different functions. These are: `bgp_keepalive_send`, `bgp_open_send`, `bgp_notify_send_with_data`, `bgp_route_refresh_send`, and `bgp_capability_send`. To narrow our search, we note that the message at line 2 can only be generated by `bgp_keepalive_send`. This function is, in turn, called by three different functions: `bgp_fsm_open`, `bgp_fsm_keepalive_expire`, and `bgp_establish`. Of these three functions, only `bgp_establish` can generate the message of line 1.

Based on the preceding analysis, we hypothesize that the sequence of messages observed between lines 1 and 3 are due to `bgp_establish`. Because this is the last function executed before the 3.19 second gap, we examine its source code, looking for any potentially long-running computations. We present an excerpt of the source code for `bgp_establish` as Listing 6.10. Note that due to the length of the function, we omit the code that generates the log file message of line 1, and focus on the portion from the code that generates the message of line 2, through the end of the function.

Examining Listing 6.10, we observe a call to `bgp_announce_route_all` at line 12. This function calls `bgp_announce_route`, which, in turn, calls `bgp_announce_table`. Accordingly, we inspect the

```
if (peer->v_keepalive)                                                                  1
  bgp_keepalive_send (peer);                                                            2
                                                                                        3
/* First update is deferred until ORF or ROUTE-REFRESH is received */                   4
for (afi = AFI_IP ; afi < AFI_MAX ; afi++)                                              5
  for (safi = SAFI_UNICAST ; safi < SAFI_MAX ; safi++)                                  6
    if (CHECK_FLAG (peer->af_cap[afi][safi], PEER_CAP_ORF_PREFIX_RM_ADV))               7
      if (CHECK_FLAG (peer->af_cap[afi][safi], PEER_CAP_ORF_PREFIX_SM_RCV)              8
          || CHECK_FLAG (peer->af_cap[afi][safi], PEER_CAP_ORF_PREFIX_SM_OLD_RCV))      9
        SET_FLAG (peer->af_sflags[afi][safi], PEER_STATUS_ORF_WAIT_REFRESH);            10
                                                                                        11
bgp_announce_route_all (peer);                                                          12
                                                                                        13
BGP_TIMER_ON (peer->t_routeadv, bgp_routeadv_timer, 1);                                 14
                                                                                        15
return 0;                                                                               16
```

Listing 6.10: Excerpted source code of `bgp_establish`, from `bgp_fsm.c`. The call to the function `bgp_keepalive_send`, at line 2, generates the log messages at lines 2 and 3 of Listing 6.9.

source code for `bgp_announce_table`, to determine whether that function might account for the 3.14 second delay observed in the log file data. We present the source for `bgp_announce_table` as Listing 6.11.

Examining Listing 6.11, we focus on the loop that begins at line 16. We observe that it iterates over `bgp_node` elements, and includes, at line 17, a nested loop over `bgp_info` elements. For each of these `bgp_info` elements, the nested loop checks, at line 18, if the `bgp_info` element has the `BGP_INFO_SELECTED` flag set. If so, and if `bgp_announce_check` returns true, at line 22, the function calls `bgp_adj_out_set`, at line 23.

Assuming that the `bgp_node` elements represent IP address prefixes, that the `bgp_info` elements represent different routes to those prefixes, and that `bgp_adj_out_set` generates an outbound advertisement for a prefix, we conclude that this function identifies the routes that should be advertised to a peer, and generates routing advertisements for each of those routes.

Given the relatively large number of routes that the target router must advertise to the customer router, it is reasonably likely that this function accounts for the 3.14 second delay observed in Listing 6.9. Accordingly, to address the delay, we developed a patch which delays the execution of `bgp_announce_table`. We present the core source code for this patch as Listing 6.12.

The patch of Listing 6.12 prevents `bgp_announce_route` from calling `bgp_announce_table`, unless `bgpd` has received an End-of-RIB marker from its peer router. This patch functions in conjunction with the patches of Listings 6.8 and 6.5. The former ensures that received updates are processed by the time that the End-of-RIB marker is received, while the latter ensures that any updates resulting from the received advertisements are propagated upstream as the End-of-RIB marker is received. Because `bgp_announce_table` will execute only after the End-of-RIB marker is received, it will no longer affect the critical path.

Before proceeding, we note that this patch implements behavior similar to that specified for the role of a restarting router in Graceful Restart. Specifically, as explained in Section 4.1.1, a restarting router will defer transmission of its routing table to its peers, until it has received the End-of-RIB

```
1  static void
2  bgp_announce_table (struct peer *peer, afi_t afi, safi_t safi,
3                      struct bgp_table *table, int rsclient)
4  {
5    struct bgp_node *rn;
6    struct bgp_info *ri;
7    struct attr attr = { 0 };
8
9    if (! table)
10     table = (rsclient) ? peer->rib[afi][safi] : peer->bgp->rib[afi][safi];
11
12   if (safi != SAFI_MPLS_VPN
13       && CHECK_FLAG (peer->af_flags[afi][safi], PEER_FLAG_DEFAULT_ORIGINATE))
14     bgp_default_originate (peer, afi, safi, 0);
15
16   for (rn = bgp_table_top (table); rn; rn = bgp_route_next(rn))
17     for (ri = rn->info; ri; ri = ri->next)
18       if (CHECK_FLAG (ri->flags, BGP_INFO_SELECTED) && ri->peer != peer)
19         {
20          if ( (rsclient) ?
21               (bgp_announce_check_rsclient (ri, peer, &rn->p, &attr, afi, safi))
22               : (bgp_announce_check (ri, peer, &rn->p, &attr, afi, safi)))
23             bgp_adj_out_set (rn, peer, &rn->p, &attr, afi, safi, ri);
24           else
25             bgp_adj_out_unset (rn, peer, &rn->p, afi, safi);
26
27           bgp_attr_extra_free (&attr);
28         }
29 }
```

Listing 6.11: Source code of `bgp_announce_table`, from `bgp_route.c`. This function is reached from the call to `bgp_announce_route_all` at line 12 of Listing 6.10..

```
/* First update is deferred until ORF or ROUTE-REFRESH is received */
if (CHECK_FLAG (peer->af_sflags[afi][safi], PEER_STATUS_ORF_WAIT_REFRESH))
  return;

/* First update is deferred until peer has sent End-of-RIB */
if (CHECK_FLAG (peer->af_sflags[afi][safi], PEER_STATUS_EOR_WAIT))
  return;

if (safi != SAFI_MPLS_VPN)
  bgp_announce_table (peer, afi, safi, NULL, 0);
else
  for (rn = bgp_table_top (peer->bgp->rib[afi][safi]); rn;
       rn = bgp_route_next(rn))
    if ((table = (rn->info)) != NULL)
     bgp_announce_table (peer, afi, safi, table, 0);
```

Listing 6.12: Core source code for our second patch to improve route processing delay. This patch modifies bgp_announce_route in bgp_route.c. The complete patch is provided as Listings B.58–B.69.

	before patch	with patch
internet to CUST1	5.51	3.52
internet to CUSTN	5.52	3.52
customer to INET1	0.05	0.04
customer to INETN	0.05	0.05
any	5.52	3.53

Table 6.10: Comparison of mean outage times, in seconds, and over ten trials, before, and with, the patch of Listing 6.12. Data for the before case are copied from Table 6.7.

marker from all of its peers. Our patch follows a similar strategy, but applies it only to the peer being rehomed.

6.5.2 Evaluation, and Design discussion

As with our other patches in this chapter, we evaluate the effectiveness of this patch by comparing the total downtime before and after the patch, and by examining the component-wise decomposition of down time. We present the total downtime in Table 6.10, and the decomposition of down time in Table 6.11. We observe that the patch has reduced mean overall outage time by approximately 2 seconds. Further, as expected, the improvement comes primarily from a reduction in route processing delay.

Having demonstrated that our patch achieves its goal, we now discuss alternative approaches to achieving the same goal. In particular, instead of removing `bgp_announce_table` from the critical path, we might pursue one of two other alternatives. First, we might simply optimize the function, to reduce its running time. Second, we might modify the function to yield periodically, rather than

component	from	to	before patch	with patch
session establishment	outage start	**bgp** $\uparrow_{c}$	1.74	1.73
route reception	**bgp** $\uparrow_{c}$	**+custn**$_{\triangleleft c}$	1.61	1.38
route processing	**+custn**$_{\triangleleft c}$	$\sim$**custn**	2.14	0.44
route propagation	$\sim$**custn**	**+custn**$_{\triangleright r}$	-0.05	-0.11
other	**+custn**$_{\triangleright r}$	outage end	0.07	0.08

Table 6.11: Breakdown of outage time after application of the patch in Listing 6.12. All values are means over ten trials. Because we detect the $\sim$**custn** event by polling once-per-second, we adjust the raw route processing value by subtracting 0.5 seconds, and adjust the raw route propagation value by adding 0.5 seconds, before reporting them here. We repeat the data of Table 6.8, in the "before patch" column, for ease of reference.

running to completion. We consider these in turn.

If there were straightforward means to significantly reduce the processing time required by `bgp_announce_table`, we might prefer that optimization to our patch. However, a quick inspection of the source code for this function, as provided in Listing 6.11, gives us little hope that such means can be found. There are two reasons for our pessimism, as we next explain.

We first consider optimizing `bgp_announce_table`. Examining the code of its main loop, starting at line 16 of Listing 6.11, we find that the loop simply iterates over the relevant data structures, calling other functions to do the bulk of the work. Thus, there is no obvious optimization to be made in `bgp_announce_table` itself.

We next consider optimizing the functions called by `bgp_announce_table`. Note that these functions, `bgp_route_next`, `bgp_announce_check`, and `bgp_adj_out_set`, are at the core of what `bgpd` does. Because they represent the common-case code paths, it is likely that they are already well optimized, and that any further optimization would be difficult and intrusive.

While it might not be feasible to reduce the running time of `bgp_announce_table`, we might reduce its impact on route processing delay by modifying `bgp_announce_table` to yield during its execution, rather than running to completion. The key challenge to such a modification is guaranteeing that the function correctly generates the appropriate route advertisements, even if other functions modify the routing table.

As it turns out, the BGP software in XORP [39] uses such a strategy. Accordingly, to assess the difficulty of implementing such a change, we consult the design document for the XORP BGP daemon [86]. In this document, the authors of XORP enumerate the complications that arise from this strategy, and note that their code for handling those complications is "perhaps the most complex part of the BGP machinery."

Given that the XORP developers found supporting a concurrent RIB dumping strategy a complex task in a clean slate design, we expect retrofitting such a change on to an existing router to be impractical. Thus, we conclude that neither reducing the running time of `bgp_announce_table`, nor reducing its impact by causing it to yield periodically, is a viable strategy.

δt	sender	description
0.00	target	BGP OPEN
1.26	customer	BGP OPEN
0.00	target	BGP KEEPALIVE
0.00	customer	BGP KEEPALIVE
0.00	target	BGP KEEPALIVE
0.33	customer	BGP KEEPALIVE
1.00	customer	BGP UPDATE

Figure 6.2: Sequence of packets exchanged by the target router and the customer router, between the start of the outage for traffic to CUSTN, and **+CUSTN**$_{\ll}$. The left-most column provides the elapsed time, in seconds, between a message and its predecessor. This figure presents data from the trial with the minimal overall outage time.

6.5.3 Avenues for improvement

Turning our attention to avenues for further improvement, we examine the breakdown of outage time in Table 6.8, and note that the dominant factors are now session establishment delay, and route reception delay. To understand the source of these remaining delays, we consult our packet captures. We begin with the packet capture for the trial with the minimal outage time.

Figure 6.2 presents the sequence of packets transmitted during session establishment and route reception, for the trial with the minimal outage time, and as captured on the target router. Examining the left-most column, we observe that all of the inter-packet delays occur before the transmission of a packet from the customer router. This suggests that much of the remaining session establishment and route reception delays are due to the customer router.

To verify this hypothesis, we sum the δt values, for packets transmitted by the customer router, for each trial. We then compute the mean of this sum, over all trials. This yields a mean customer-side delay of 2.81 seconds. Given that Table 6.11 reports a mean session establishment delay of 1.73 seconds, and a mean route reception delay of 1.38 seconds, we conclude that both of these delays are largely attributable to the customer router.

The finding that much of the remaining outage time is due to the customer router might suggest that we must modify the customer router in order to make further improvements. After all, the customer router is the authoritative source of information about which prefixes are reachable through the customer network. Without advertisements from the customer router, how can the target router know which packets to forward to the customer router?

To answer this question, we revisit our example of Graceful Rehoming, from Chapter 4. In particular, we illustrate the initial state of the initial, customer, and target routers with an excerpt from Figure 4.2. We present the excerpt here as Figure 6.3. We observe that the loc-RIB at the target router initially includes CUST1 and CUSTN, due to advertisements that the target router has received from the initial router. We might, then, be able to use this routing information until the target router has established its own peering with the customer router, and received routes from the customer router.

	initial router (I)		customer router (C)		target router (T)	
step	RIB-in C	loc-RIB	RIB-in ISP	loc-RIB	RIB-in C	loc-RIB
start	CUST1	INET1	INET1	INET1	<down>	INETN
	CUSTN	INETN	INETN	INETN		CUST1-I
		CUST1-I		CUST1		CUSTN-I
		CUSTN-I		CUSTN		

Figure 6.3: Excerpted example of Graceful Rehoming, illustrating initial system state. Suffixes on loc-RIB entries denote the next-hop router for the respective prefixes. For the full example, see Figure 4.2.

6.6 Conclusion

We set out, in this chapter, to further improve the down time caused by the target router. Having exhausted the most promising micro-optimizations in Chapter 5, we turned our attention to scheduling optimizations. We first showed that simply giving the operator control over when the target router initiates a new peering session with the customer router improves outage time by 42%. We then showed that modifying the scheduling of route processing and route propagation further improved outage time by 53%.

With these optimizations, we have improved mean overall outage time during rehoming from 12.91 seconds to just 3.53 seconds. Consequently, assuming that we move customers to a temporary router and back, we can now support historical rates of routing software upgrades using only 11% of our annual outage budget. Applying rehoming to other maintenance events as well, we could support 39 such events without exceeding our annual outage budget.

Based on the numbers above, we believe that the optimizations presented in this chapter yield a viable rehoming solution. The solution greatly reduces the impact of planned maintenance due to software upgrades. And, if our solution were extended to apply to other sources of downtime, it could deliver five-nines availability even if a router experienced a failure, or required maintenance, every 8 days.

Extending our solution to apply to other sources of downtime is beyond the scope of this work, though we do present our thoughts on such extensions in Section 8.3.2. In the next chapter, we turn instead to a more readily addressed issue. In particular, we explore whether we can exploit the observation of Section 6.5.3 to make rehoming transparent to real-time applications. To do so, we will need to reduce the down time of a rehoming event from our current 3.53 seconds, to sub-second levels.

To infinity… and beyond!

Buzz Lightyear

7 ZIRO

From our baseline experiments in Chapter 2, to our scheduling improvements in Chapter 6, we have made great strides in reducing the down time caused by routing software upgrades. In Chapter 2, we established that, with Low Spec hardware, a customer using BGP with dynamic routing would experience a mean down time of approximately 140 seconds. In Chapter 3, we explained the concept of rehoming, and showed that we could rehome a customer with approximately 120 seconds of down time. While this was not a viable solution in itself, it set us on the right path.

In order to make rehoming viable, we pursued a series of optimizations. First, in Chapter 4, we showed how rehoming could be combined with Graceful Restart [73], to reduce overall outage time to approximately 23 seconds. Next, in Chapter 5, we showed how to dramatically reduce the CPU time used by Quagga during rehoming, and thereby reduce down to time to approximately 13 seconds. Finally, in Chapter 6, we demonstrated that scheduling changes could further reduce down time, bringing our mean overall outage time to approximately 3.5 seconds.

Our work in the preceding chapters is valuable not only for the improvements in down time, but for the insight it has given us in to what occurs when a BGP peering is established, or torn down. In Chapter 3, we observed that rehoming imposed tens of seconds of CPU load on the customer router, and hypothesized that this might be improved by eliminating the need to remove and reinstall routes. In Chapter 4, we saw that Graceful Rehoming greatly reduced the CPU time used on the customer router, thereby confirming our hypothesis that route removal and route installation can be expensive operations.

Chapters 5 and 6 yielded valuable insights as well. In Chapter 5, we learned that not all of the CPU load incurred during rehoming is fundamental. In particular, we found that much of the CPU time used by `zebra` on the target router was due to a bug in a portion of the scheduling code. In Chapter 6, we found that the timing of different phases of BGP processing in Quagga is non-intuitive, and that the interactions between these phases is intricate.

In this chapter we build on the improvements of, and use the lessons learned from, previous

chapters, to develop a rehoming scheme that delivers sub-second outage times. We then further leverage our new-found knowledge of router behavior to simplify this rehoming scheme. Subsequently, we investigate the performance of our scheme for a customer with 18 times as many routes, again demonstrating sub-second outage times. Finally, we argue that sub-second outage times allow for practically zero interruption to network traffic, and then conclude.

The remainder of this chapter is structured as follows:

- In Section 7.1, we introduce soft handoff, our scheme for achieving sub-second outage times. We first explain soft handoff in principle, then detail our implementation.
- In Section 7.2, we evaluate soft handoff in combination with the rehoming solution developed through Chapter 6. We call this combination ZIRO. We find that our initial design of soft handoff, as presented in Section 7.1 increases mean overall outage time by nearly an order of magnitude, to 32.85 seconds. However, a modification to the rehoming procedure of Section 7.1.3 resolves the issue, yielding a mean overall outage time of 0.70 seconds.
- In Section 7.3, we first show that our scheduling patch for ZIRO subsumes, or obviates the need for, many of the scheduling patches of Chapter 6. We then show that, because ZIRO moves nearly all control place processing off of the critical path, we can remove the CPU optimizations of Chapter 5. These changes can be made without increase outage times, but they have the effect of increasing completion times.
- In Section 7.4, we evaluate the performance of ZIRO for an autonomous system that originates 18 times as many prefixes as the network evaluated in all previous experiments in this dissertation. We show that outage times do not increase significantly, and speculate on the reason for this surprising result.
- In Section 7.5, we present our argument for the claim that ZIRO allows for practically zero interruption of network traffic. This argument considers three types of network traffic, and analyzes the expected impact based on transport protocol dynamics and human factors.
- In Section 7.6, we summarize and conclude. We offer two primary conclusions. The first is that ZIRO readily achieves the five-nines reliability goal. ZIRO enables an ISP to keep current with routing software releases, while consuming only 1.3% of the annual outage budget. Further, if extended to other use cases, ZIRO could support five-nines availability even in the face of a failure or maintenance event nearly every day. Our second conclusion is that a ZIRO rehoming event will likely be unnoticed by most users.

7.1 The Soft Handoff Concept

We concluded Chapter 6 with a mean outage time of 3.53 seconds, and the observation that the remaining down time was largely attributable to delays on the customer router. We presented the result, and the observation, in Sections 6.5.2, and 6.5.3, respectively. In Section 6.5.3, we also hypothesized that we might be able to avoid the delays caused by the customer, by exploiting routing information from the initial router.

This key insight of Section 6.5.3 derives from our illustration of the state of the routing system during graceful rehoming, first presented in Figure 4.2. We repeat and elaborate the initial state from that figure here, as Figure 7.1. Examining this figure, we observe that, even without a direct peering with the customer router, the target router knows routes for the customer prefixes. These prefixes are learned from the target router's peering with the initial router, as illustrated by the cust1 and custn entries in the **RIB-in I**. Our challenge is to exploit this information to implement

	initial router (I)		customer router (C)		target router (T)		
step	RIB-in C	loc-RIB	RIB-in ISP	loc-RIB	RIB-in I	RIB-in C	loc-RIB
start	CUST1	INET1	INET1	INET1	CUST1	<down>	INETN
	CUSTN	INETN	INETN	INETN	CUSTN		CUST1-I
		CUST1-I		CUST1			CUSTN-I
		CUSTN-I		CUSTN			

Figure 7.1: Excerpted example of Graceful Rehoming, illustrating initial system state. Suffixes on loc-RIB entries denote the next-hop router for the respective prefixes. For the full example, see Figure 4.2.

a soft handoff of customer-bound traffic from the initial router to the target router.

In order to effectively exploit this knowledge of the customer prefixes, the target router must perform three tasks. First, it must identify the subset of the RIB entries that correspond to prefixes routable through the customer router. Second, it must reprocess these RIB entries, as though they had been received directly from the customer router. Third, it must immediately propagate the resulting changes to other BGP peers.

We accomplish the first task through the use of a "route-map." We accomplish the second and third tasks, jointly, through a patch to `bgpd`, and a change to our rehoming procedure. We now explain the route-map, our patch, and the change to the rehoming procedure, in turn. Note that our patch depends crucially on the understanding of scheduling behavior that we developed in Chapter 6.

7.1.1 Our route-map

In Quagga, as in Cisco IOS, a route-map is key building block of router configuration. A route-map specifies criteria for matching some set of routes, and the actions to be taken for the matched routes. For example, a route-map might match prefixes reserved for private use [70], and take the action of filtering those routes. Similarly, a route-map might be used to lengthen the AS PATH on outbound routing advertisements, to implement ingress traffic engineering [35].

For our soft handoff scheme, our route-map must perform three tasks. First, it must match the routes sent by the customer router. Second, because these routes were learned from the initial router, our route-map must override their next-hop information, directing packets using these routes to the customer-facing interface, rather than the interface to the initial router.[1] Third, because default BGP behavior does not allow the target router to propagate routes learned from the initial router, to the remote router, our route-map must override this default behavior.[2]

These tasks are accomplished by lines 3, 5, and 6 of Listing 7.1, respectively. Line 3 identifies routes from the customer router by checking the first entry in the AS path.[3] If the first entry in the

[1] We assume that the routes arrive at the target router with the next-hop attribute set to the address of the initial router. When this is not the case, such as configurations in which the route to the customer interface is propagated through BGP, this step is not required.

[2] In an archetypal BGP deployment, all of the BGP routers within an autonomous system peer directly with each other. Hence, propagating the routes advertised by one internal BGP peer to another internal BGP peer would cause unnecessary processing load.

[3] Note that this method would not be sufficient for a customer with multiple connections to the ISP.

```
1         ip as-path access-list SOFT-HANDOFF-9 permit ^9_
2         route-map SOFT-HANDOFF-9 permit 10
3          match as-path SOFT-HANDOFF-9
4          set local-preference 1000
5          set ip next-hop 10.1.8.3
6          set reflect
```

Listing 7.1: The route-map used to identify customer routes, and mark them for soft handoff. Note that `^9_` is a regular expression that matches strings beginning with the digit 9, followed immediately by a space.

AS path matches the customer's AS number, the route will be modified by the actions of lines 4–6. Line 4 is not specific to our soft handoff scheme, but instead implements the same modification to the LOCAL_PREF attribute as described in Section 4.4. Line 5 directs packets to the customer-facing interface. Finally, line 6 works with the patch of the following subsection, to allow the target router to propagate the matching route to the remote router.

7.1.2 Our patch

While the route-map of the preceding section provides a method for selecting routes, and modifying their attributes, it remains to actually reprocess the received advertisements for these routes, and to propagate the resulting changes to BGP peers. In order to accomplish these goals, we use the soft-reconfiguration feature in Quagga, which allows an operator to modify the routing policy applied to an established peering. To use this feature effectively, however, we must make some modifications to `bgpd`.

The first of our modifications, provided in Listing 7.2, addresses an obstacle to the propagation of routes learned from the peering with the initial router, as discussed in Section 7.1.1. Namely, per Section 9.2 of the BGP-4 specification [69], a router does not ordinarily propagate a route learned from one internal BGP peer, on to other internal BGP peers. Our modification allows a network operator to override this rule by flagging routes in the manner of line 6 of Listing 7.1. When a route is so flagged, `bgpd` skips the IBGP reflection check at lines 11–32 of Listing 7.2.

Our second modification, provided in Listing 7.3, addresses the need to reprocess routes, and to propagate the resulting updates promptly on to peers. This modification is motivated by the lessons of Sections 6.3 and 6.4. Namely, in Section 6.3, we learned that `bgpd` sends updates periodically, based on a timer, rather than transmitting them as they are generated. And in Section 6.4, we learned that rather than processing routing updates immediately, `bgpd` places them in a work queue for background processing

To address these issues, the modifications of Listing 7.3 apply the patches of Listings 6.5 and 6.8 to the soft reconfiguration code in `bgpd`. Lines 9–14 repeat the changes of Listing 6.8, which immediately services the work queue of pending routing updates. Lines 16–27 repeat the changes of Listings 6.5, which immediately transmit any pending outbound updates.

The last of our modifications does not affect soft handoff directly. Instead, it addresses a concern about the broader rehoming procedure. Namely, while routing information from the initial router is valuable substitute when information is not yet available directly from the customer router, the routes from the initial router should not continue to override those from the customer router, after

```
/* Route-Reflect check. */
if (peer_sort (from) == BGP_PEER_IBGP && peer_sort (peer) == BGP_PEER_IBGP)
/* NB: special-case prefixes with LOCAL_FLAG_REFLECT, as though they were
   originated by us, rather than reflected by us. (mukesh.20100823) */
if ((peer_sort (from) == BGP_PEER_IBGP && peer_sort (peer) == BGP_PEER_IBGP)
    && !(ri->attr->local_flags & LOCAL_FLAG_REFLECT))
  reflect = 1;
else
  reflect = 0;

/* IBGP reflection check. */
if (reflect)
  {
    /* A route from a Client peer. */
    if (CHECK_FLAG (from->af_flags[afi][safi], PEER_FLAG_REFLECTOR_CLIENT))
      {
        /* Reflect to all the Non-Client peers and also to the
           Client peers other than the originator.  Originator check
           is already done.  So there is noting to do. */
        /* no bgp client-to-client reflection check. */
        if (bgp_flag_check (bgp, BGP_FLAG_NO_CLIENT_TO_CLIENT))
          if (CHECK_FLAG (peer->af_flags[afi][safi], PEER_FLAG_REFLECTOR_CLIENT))
            return 0;
      }
    else
      {
        /* A route from a Non-client peer. Reflect to all other
           clients. */
        if (! CHECK_FLAG (peer->af_flags[afi][safi], PEER_FLAG_REFLECTOR_CLIENT))
          return 0;
      }
  }
```

Listing 7.2: Core source code, part 1 of 3, for patch to implement soft handoff. This code modifies the function bgp_announce_check, in bgp_route.c. The complete soft handoff patch is provided as Listings B.70–B.77.

```
  if (safi != SAFI_MPLS_VPN)
    bgp_soft_reconfig_table (peer, afi, safi, NULL);
  else
    for (rn = bgp_table_top (peer->bgp->rib[afi][safi]); rn;
         rn = bgp_route_next (rn))
      if ((table = rn->info) != NULL)
        bgp_soft_reconfig_table (peer, afi, safi, table);

  res = WQ_SUCCESS;
  t.arg = bm ? bm->process_main_queue : NULL;
  while (bm && bm->process_main_queue && bm->process_main_queue->items
         && listcount(bm->process_main_queue->items) &&
         (res == WQ_SUCCESS))
    res = work_queue_run(&t);

  for (ALL_LIST_ELEMENTS (peer->bgp->peer, node, nnode, p))
    {
      if (p->status != Established)
        continue;

      t.arg = p;
      p->synctime = bgp_clock() + 1;
      /* cancel any pending write thread, since we're taking
         care of writes here. (mukesh.20100625). */
      BGP_WRITE_OFF(p->t_write);
      bgp_write(&t);
    }
```

Listing 7.3: Core source code, part 2 of 3, for patch to implement soft handoff. This patch modifies the function bgp_soft_reconfig_in, bgp_route.c. The complete soft handoff patch is provided as Listings B.70–B.77.

```
if (! attribute_len && ! withdraw_len)
  {
    ...
    if (BGP_DEBUG (normal, NORMAL))
      zlog (peer->log, LOG_DEBUG, "rcvd End-of-RIB for IPv4 Unicast from %s",
            peer->host);

    if (bgp_flag_check (peer->bgp, BGP_FLAG_LOG_NEIGHBOR_CHANGES))
      zlog_info ("rcvd End-of-RIB for IPv4 Unicast from %s", peer->host);
  }
```

Listing 7.4: Core source code, part 3 of 3, for patch to implement soft handoff. This code modifies the function bgp_update_receive, in bgp_packet.c. The complete soft handoff patch is provided as Listings B.70–B.77. Ellipsis denote where one or more lines of the original source have been omitted, for clarity of presentation.

bridge	initial	target	remote
		start	
move link		enable NIC	new CUSTN in bgpd?
		start soft handoff	
		new route	
		enable BGP	
		received end-of-RIB?	
		stop soft handoff	
		done	

Figure 7.2: Soft handoff rehoming procedure. The column for each node lists the steps taken on that node. Gray horizontal bars indicate barriers used to synchronize actions across nodes. Steps between barriers are unordered with respect to actions of other nodes. Differences from the naïve rehoming procedure of Figure 3.1 are emphasized in italics.

the peering between the target router and the customer router is established.

In order to prevent this situation from arising, our rehoming procedure will remove the route-map after the target router has received routing information from the customer router. To do so, the rehoming script needs to know when bgpd has processed the End-of-RIB marker from the customer router. Accordingly, the code of Listing 7.4 generates a log file message when bgpd receives the End-of-RIB marker.

7.1.3 Our rehoming procedure

In order for our route-map, and our patch, to take effect, we must modify our rehoming procedure. Most obviously, the rehoming procedure must first initiate soft handoff, and, following the receipt of routing advertisements from the customer router, terminate soft handoff. We illustrate these, and the rest of our changes, relative to the naïve rehoming procedure of Figure 3.1, in Figure 7.2.

Focusing on the steps emphasized in italics, we note four changes. First, after enabling the customer-facing interface card on the target router, we initiate the soft handoff process. Second, before instructing the target router to enable its BGP peering with the customer router, we wait for the remote router to receive the routing updates generated by the soft handoff process. Third, after enabling the BGP peering, we wait for bgpd on the target router to report that it has received the End-of-RIB marker from the customer router. Fourth, after the End-of-RIB marker has been received, we terminate the soft handoff process.

The need for the first of these changes is self-evident, and the purpose of the third and fourth changes was explained at the conclusion of Section 7.1.2. Accordingly, we will not belabor them here. The second change, which delays enabling the BGP session with the customer router until after the remote router has received routing updates resulting from soft handoff, however, merits elaboration.

This second change is motivated by the desire to eliminate a race condition, which might expose effects of rehoming to systems outside of the ISP's network. Specifically, in order to keep rehoming transparent to other autonomous systems, we must ensure that the remote router does not generate withdrawal messages for the customer prefixes. This, in turn, requires that the remote router always have a viable route to the customer prefixes during the rehoming process.

We might expect that the remote router would, in fact, always have a viable route to the customer prefixes. Specifically, in Section 4.4, we designed a solution that would establish a route to the customer prefixes through the target router, without explicitly invalidating the routes through the initial router. Our previous rehoming process does, however, implicitly invalidate the routes through the initial router. This implicit invalidation occurs when the initial router processes the advertisements, from the target router, for cust1 and custn. We will explain the cause of this implicit validation shortly.

Whether or not the implicit invalidation causes the remote router to generate withdrawal messages to its external BGP peers, for cust1 and custn, depends on message ordering. We illustrate the possible outcomes in Figures 7.3(a) and 7.3(b). We first explain the sequence of events that occurs in Figure 7.3(a), which illustrates the case where rehoming proceeds transparently. We then explain the key difference in the unsuccessful case, illustrated in Figure 7.3(b). In the successful case, rehoming proceeds as follows:

1. We initiate soft handoff on the target router. This causes the target router to generate and transmit advertisements for the customer routes, such as custn, to the remote router. Note that the target router does not send the same advertisements to the initial router, as the routes triggering the advertisements originated from the initial router.
2. The target router establishes a new peering with the customer router, and receives the customer router's advertisement for custn. Because this advertisement originated from the customer router, the target router propagates the advertisement to both, the initial router, and the remote router.
3. The initial router processes the advertisement, for custn, from the target router. Because this route has a higher LOCAL_PREF value than the route learned from the customer router, the initial router selects this route. The initial router then sends withdrawal messages for custn, to both the target router, and the remote router.

That the initial router generates a withdrawal message in step 3 might run counter to intuition. Specifically, we might expect the initial router to advertise the selected route to the remote router. However, as discussed in Section 7.1.2, rules for internal BGP peerings prohibit the initial router from propagating an advertisement received from the target router, on to the remote router. Thus,

the initial router generates a withdrawal message for CUSTN.[4]

The timing of this withdrawal message is, in turn, the key to understanding the unsuccessful case. In Figure 7.3(b), we execute the same sequence of steps as in Figure 7.3(a), and the routers within the ISP generate the same set of messages to each other. However, the messages from the target router to the remote router are delayed, such that they arrive after the withdrawal of CUSTN from the initial router. In this case, the remote router will send a withdrawal for CUSTN to its external peers.

It is to eliminate the possibility of this occurrence that our rehoming procedure waits for the remote router to receive the new routes, through the target router, before enabling the BGP peering between the target router and the customer router. We claim that with this barrier in place, the remote router cannot process the withdrawal from the initial router before processing the advertisement from the target router.

Our argument proceeds as follows. First, note that the initial router's withdrawal, at step 3, is triggered by the events of step 2. Second, note that the barrier ensures that step 2 occurs only after the remote router has processed the advertisements generated at step 1. Now, as step 3 depends on step 2, and step 2 depends on the remote router processing the advertisements of step 1, it follows that the remote router cannot process the withdrawal of step 3 before the advertisement of step 1.

7.2 ZIRO Results

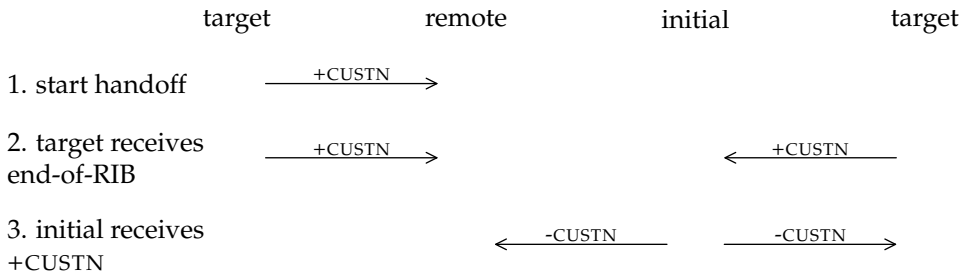
Having explained the soft handoff concept, and its implementation, we turn now to evaluating our idea in practice. We first show that the scheme, as explained in Section 7.1, actually increases mean overall outage time, from 3.53 seconds to 32.85 seconds. We then diagnose the problem through the use of log file data and code inspection. Based on this diagnosis, we revise the rehoming procedure of Section 7.1.3, and demonstrate that this revised procedure achieves a mean overall outage time of 0.70 seconds. Finally, we conclude the section with a brief design discussion.

7.2.1 Evaluation

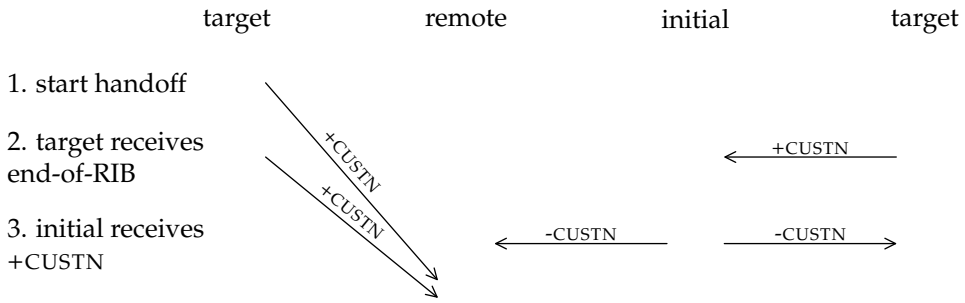
We present mean overall times for ZIRO in Table 7.1. We note that the mean outage times reported in this table are troubling. Instead of improving down time, our soft handoff scheme has increased it from 3.53 seconds to 32.85 seconds. To determine the cause for this highly unexpected increase, of nearly an order of magnitude, we consult the system charts for the trials with the shortest and longest overall outage times, We present these charts as Figures 7.4(a), and 7.4(b), respectively.

Examining these charts, we make three observations. First, the outage time is highly variable. In the trial with the minimal outage time, the overall outage time is approximately 12 seconds. In contrast, the trial with the maximal overall outage time sees an outage of approximately 63 seconds. Second, we observe that the CPU on the target router is largely idle during these outages. Third, we note that the $\sim$**CUSTN** and **+CUSTN**$_{\triangleright R}$ events both occur much later than the **start handoff** event.

[4] We observed this behavior while examining the full set of system charts for the experiments of Section 6.2. Specifically, in some trials, we observed a brief restoration of connectivity to CUST1 and CUSTN, as the remote router continued to route through the initial router, after the initial router had selected the route through the target router. Shortly thereafter, the remote router would process the withdrawal from the initial router. Connectivity would then be lost, until the remote router processed the advertisements, from the target router, for CUST1 and CUSTN. We provide the system chart for one such trial as Figure A.25.



(a) successful case, transparent to external peers

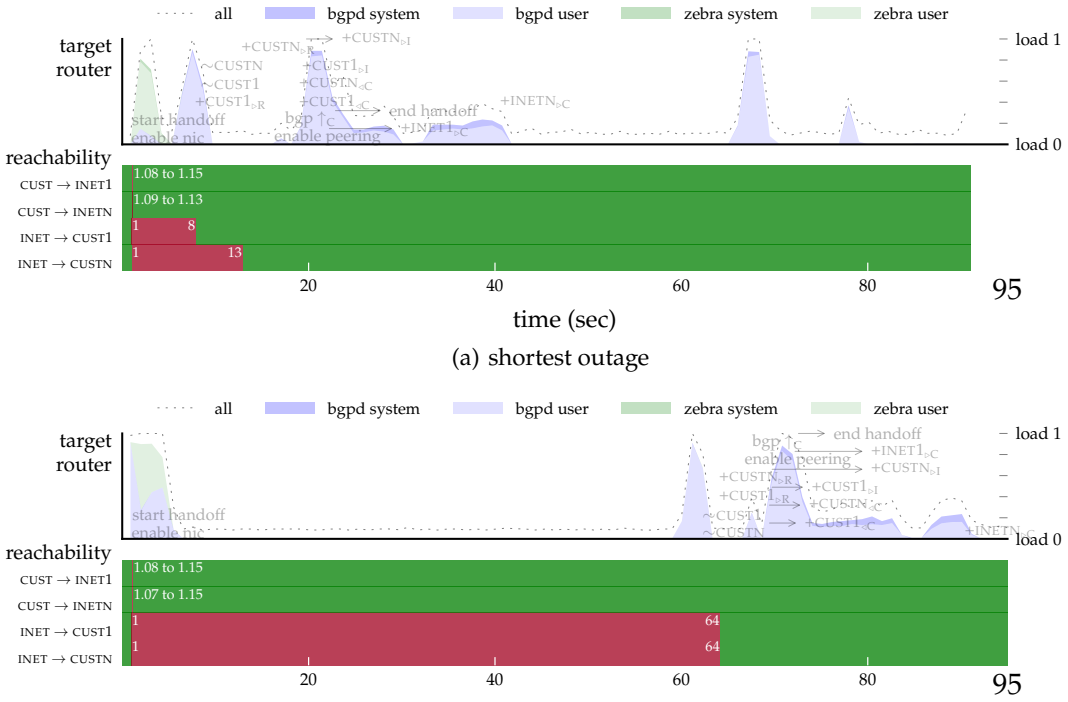


(b) unsuccessful case, where external peers would receive a withdrawal for CUSTN

Figure 7.3: Illustration of race condition that would exist without the "new route" barrier in the soft handoff rehoming procedure of Figure 7.2.

	before soft handoff	with soft handoff
internet to CUST1	3.52	31.35
internet to CUSTN	3.52	32.85
customer to INET1	0.04	0.06
customer to INETN	0.05	0.05
any	3.53	32.85

Table 7.1: Comparison of mean outage times, in seconds, and over ten trials, before, and with soft handoff.



(a) shortest outage

(b) longest outage

Figure 7.4: Partial system charts comparing trials of ZIRO, using the rehoming procedure of Figure 7.2. The figure illustrates the trials with the shortest and longest outages. For the complete system charts, see Figures A.20 and A.21.

message	delay
Performing BGP general scanning	1.80
scanning IPv4 Unicast routing tables	0.49
routeadv timer expire	1.86
BGP connection from host 10.1.8.3	n/a
BGP connection IP address 10.1.8.3 is Idle state	n/a

Table 7.2: Mean delay, over ten trials, between various log file messages and the FIB update for cust1. For messages that did not occur in one or more trials, the delay is listed as "n/a".

Because the ∼**custn** event should be a direct consequence of our soft handoff code, and not depend on other routers in the system, we hypothesize that the cause of the delay is local to the target router. That the outages end soon after the **+custn**$_{\triangleright R}$ event provides further support for the hypothesis that the problem is localized to the target router, rather than, for example, being due to the remote router. That the delay is highly variable, even though the CPU is idle, suggests that a timer may be involved. We next investigate this hypothesis in depth.

7.2.2 Diagnosis

To test our hypothesis that the delay is due to a timer, we repeat the experiment, with all possible debugging options enabled for bgpd. We then examine the log file data for the trial with the maximal overall outage time, focusing on the five seconds prior to the **enable peering** event.[5] We focus on the messages immediately prior to this event, because this event occurs soon after the remote router has received the target router's advertisements for cust1 and custn. Thus, this event occurs soon after soft handoff has completed its work.

We present the relevant log file messages in Listing 7.5, noting that the last line in the listing corresponds to the **enable peering** event. Examining the rest of the listing, we observe that bgpd updates the local FIB entries for the customer routes at lines 7–13. Approximately 1.59 seconds later, at lines 15–21, bgpd transmits advertisements for the prefixes to the remote router. In order to explain the 60 seconds outage, then, we must determine why the local FIB entries are not updated earlier.

The simplest explanation, given the data before us, is that the FIB updates are triggered by the import timer expiration that immediately precedes them, at line 6. To test this hypothesis, we consult the log files for the other nine trials. Specifically, we compute the mean delay between the last expiration of the import timer before the update to the FIB entry for cust1, and the time of the update to the FIB entry for cust1 itself. This analysis yields a mean delay of 10.80 seconds, casting considerable doubt on our hypothesis.

Given this rather large mean delay, we repeat our analysis for the the other messages of lines 1–6 in Listing 7.5, in search of a more likely explanation. We present the results of this analysis in Table 7.2. Examining that table, we note that the message that occurs most closely before the FIB updates, over all ten trials is the message regarding scanning of IPv4 Unicast routing tables. The mean delay between that message, and the beginning of FIB updates, is 0.49 seconds.

Based on the mean delay data, we now hypothesize that the FIB updates are triggered by the IPv4 Unicast scan. We note that our hypothesis is not entirely satisfying, as the log messages

[5]The overall outage time for this trial was 60.13 seconds.

```
10:07:39.035483 BGP: Performing BGP general scanning                                    1
10:07:40.196898 BGP: scanning IPv4 Unicast routing tables                               2
10:07:40.197175 BGP: 10.1.2.2 [FSM] Timer (routeadv timer expire)                       3
10:07:40.197320 BGP: [Event] BGP connection from host 10.1.8.3                          4
10:07:40.197359 BGP: [Event] BGP connection IP address 10.1.8.3 is Idle state          5
10:07:40.526530 BGP: Import timer expired.                                              6
10:07:40.611136 BGP: Zebra send: IPv4 route add 128.2.0.0/16 nexthop 10.1.8.3 metric 0  7
10:07:40.612871 BGP: Zebra send: IPv4 route add 128.237.0.0/16 nexthop 10.1.8.3 ...     8
10:07:40.753265 BGP: Zebra send: IPv4 route add 192.12.32.0/24 nexthop 10.1.8.3 ...     9
10:07:40.758005 BGP: Zebra send: IPv4 route add 192.58.107.0/24 nexthop 10.1.8.3 ...    10
10:07:40.760130 BGP: Zebra send: IPv4 route add 192.80.210.0/24 nexthop 10.1.8.3 ...    11
10:07:40.980994 BGP: Zebra send: IPv4 route add 204.194.28.0/22 nexthop 10.1.8.3 ...    12
10:07:41.086341 BGP: Zebra send: IPv4 route add 209.129.244.0/23 nexthop 10.1.8.3 ...   13
10:07:42.672505 BGP: 10.1.6.3 [FSM] Timer (routeadv timer expire)                       14
10:07:42.672764 BGP: 10.1.6.3 send UPDATE 128.2.0.0/16                                  15
10:07:42.672810 BGP: 10.1.6.3 send UPDATE 209.129.244.0/23                              16
10:07:42.672843 BGP: 10.1.6.3 send UPDATE 204.194.28.0/22                               17
10:07:42.672876 BGP: 10.1.6.3 send UPDATE 192.80.210.0/24                               18
10:07:42.672908 BGP: 10.1.6.3 send UPDATE 192.58.107.0/24                               19
10:07:42.672940 BGP: 10.1.6.3 send UPDATE 192.12.32.0/24                                20
10:07:42.672973 BGP: 10.1.6.3 send UPDATE 128.237.0.0/16                                21
10:07:43.741184 BGP: Vty connection from 127.0.0.1                                      22
```

Listing 7.5: Log file messages from `bgpd`, for the five seconds prior to the **enable peering** event. Ellipsis denote that a line has been truncated to fit the page width. These logfile messages come from the trial with the maxmimal overall outage time.

at lines 3–6 of Listing 7.5 show other activities between the scan message, and the FIB updates. Nonetheless, we proceed with this hypothesis, as it is the most promising at hand. We will return to the other log messages before concluding the subsection.

Examining the source code for bgpd, we find that the message regarding the IPv4 Unicast scan is generated by the function bgp_scan, in bgp_nexthop.c. We present the core loop of that function here, as Listing 7.6. We shall now proceed to explain this loop. We will assume, as we did in Section 6.5.1, that the rn variable loops over IP address prefixes, and that the bi variable iterates over different routes to a given prefix.

We first observe that the code checks each potential route to each prefix, at lines 13–17, to determine if the next hop IP address for that prefix is valid. Later, at line 20, the code checks if the validity of the nexthop differs from its previous value.[6] If the validity state has changed, the code modifies the BGP_INFO_VALID flag for the prefix and route, at line 26, or 30. Finally, at line 37, the code calls bgp_process, to effect any necessary changes.

Based on the above explanation of the code, we hypothesize that the FIB updates occur after the bgp_scan code has set the BGP_INFO_VALID flag to true, and bgp_process has effected the relevant changes. This would be consistent with Section 9.2.1.1 of the BGP-4 specification [69], which states that BGP implementations should not select routes for which the next hop IP address can not be resolved through the current routing table. Further, our maximal outage time of 63 seconds agrees well with the default bgp_scan interval of 60 seconds in bgpd.

This explanation assumes that the next hop IP addresses of cust1 and custn are unreachable at the time that our soft handoff code executes. We hypothesize that this is true, and that this occurs because of a race condition between the execution of our soft handoff code in bgpd, and the execution of the code in zebra that informs bgpd that the interface to the customer router is ready for use.

In the next subsection, we consider and evaluate a solution to the extended outages, under the assumption that the above hypothesis holds. Before proceeding, however, we note that line line 37 of Listing 7.6 provides an explanation for the messages we observed between the start of the IPv4 Unicast scan, and the FIB updates. Namely, as noted in Section 6.4.1, bgp_scan does not process changes immediately. Instead, it queues the change for processing at background priority. Thus, timers and other events may interrupt this processing.

7.2.3 Revision and Re-evaluation

In the previous subsection, we hypothesized that the extended outage that we observe with soft handoff are due to two factors. First, at the time that our soft handoff code executes, the next hop IP address of the routes to cust1 and custn are not reachable. Second, because bgpd evaluates next hop changes every 60 seconds, there can be a significant delay between the time that an IP address becomes reachable, and the time when routes using that next hop are selected for use.

In order to resolve these problems, we modify our rehoming procedure once again. We make three changes, as we illustrate in Figure 7.5. This includes two changes on the target router, and one change on the bridge. On the target router, we insert two steps between enabling the customer-facing interface card, and initiating soft handoff. In the first new step, we wait for the CPU to become idle. This gives zebra to complete the processing that occurs after an interface state change.

[6] Note that the most current validity state is in the variable valid, while the prior validity state is in the unfortunately named variable current.

```
for (rn = bgp_table_top (bgp->rib[afi][SAFI_UNICAST]); rn;
     rn = bgp_route_next (rn))
  {
    for (bi = rn->info; bi; bi = next)
      {
        next = bi->next;

        if (bi->type == ZEBRA_ROUTE_BGP && bi->sub_type == BGP_ROUTE_NORMAL)
          {
            changed = 0;
            metricchanged = 0;

            if (peer_sort (bi->peer) == BGP_PEER_EBGP && bi->peer->ttl == 1)
              valid = bgp_nexthop_check_ebgp (afi, bi->attr);
            else
              valid = bgp_nexthop_lookup (afi, bi->peer, bi,
                                          &changed, &metricchanged);

            current = CHECK_FLAG (bi->flags, BGP_INFO_VALID) ? 1 : 0; ...
            if (valid != current)
              {
                if (CHECK_FLAG (bi->flags, BGP_INFO_VALID))
                  {
                    bgp_aggregate_decrement (bgp, &rn->p, bi,
                                             afi, SAFI_UNICAST);
                    bgp_info_unset_flag (rn, bi, BGP_INFO_VALID);
                  }
                else
                  {
                    bgp_info_set_flag (rn, bi, BGP_INFO_VALID);
                    bgp_aggregate_increment (bgp, &rn->p, bi,
                                             afi, SAFI_UNICAST);
                  }
              } ...
          }
      }
    bgp_process (bgp, rn, afi, SAFI_UNICAST);
  }
```

Listing 7.6: Source code for the core loop of `bgp_scan`, in `bgp_nexthop.c`. Ellipsis denote omitted lines. Note, however, that the omitted lines do not affect the control flow of the function.

bridge	initial	target	remote
		start	
		enable NIC	new CUSTN in bgpd?
		CPU idle?	
NIC up		*NIC up*	
move link		start soft handoff	
		new route	
		enable BGP	
		received end-of-RIB?	
		stop soft handoff	
		done	

Figure 7.5: Revised soft handoff rehoming procedure. The column for each node lists the steps taken on that node. Gray horizontal bars indicate barriers used to synchronize actions across nodes. Steps between barriers are unordered with respect to actions of other nodes. Differences from the original soft handoff rehoming procedure of Figure 7.2 are emphasized in italics.

	before soft handoff	with soft handoff	with revised soft handoff
internet to CUST1	3.52	31.35	0.64
internet to CUSTN	3.52	32.85	0.69
customer to INET1	0.04	0.06	0.04
customer to INETN	0.05	0.05	0.05
any	3.53	32.85	0.70

Table 7.3: Comparison of mean outage times, in seconds, and over ten trials, before soft handoff, with soft handoff, and with revised soft handoff.

In the second new step, we signal to the bridge that we have completed interface bring-up. The bridge, in turn, waits for this signal before moving the customer link to the target router.

We present the mean outage times for this rehoming procedure, as compared to previous experiments in Table 7.3. We observe that this change resolves the extended outage times previously observed with soft handoff. Moreover, this change enables soft handoff to reduce down time, as compared to our best previous result. Specifically, mean overall outage time drops from 3.53 seconds, to just 0.70 seconds. We will investigate the source of the remaining down time in Section 7.4, after first simplifying our solution in Section 7.3.

7.2.4 Design Discussion

Having demonstrated the effectiveness of our solution for the nexthop validity problem, we pause to consider an alternative solution. Namely, we might configure `bgpd` to run `bgp_scan` more frequently. Based on log file messages that indicate when `bgp_scan` starts, and when it completes, however, we believe doing so would be impractical. Specifically, we find that, `bgp_scan` takes, on average, 1.36 seconds to complete. In order to simply equal our mean outage time to below our pre-

vious best, of 3.52 seconds, we would need to run `bgp_scan` every 4.32 seconds. Doing so would, however, consume over 30% of the CPU time available.

To make the above strategy more viable, we might optimize next hop validity scanning by maintaining an index from next hop IP addresses, to routes using those next hops. Then, when examining routes to determine if their next hops were valid, we could analyze only those routes for which the next hop status had changed since the last call to `bgp_scan`. In our situation, doing so would reduce the number of routes to be analyzed from over 300,000, to just 7. This change would be similar to the next-hop address tracking feature introduced by Cisco, in IOS version 12.0(29)S [24]. The change, would, however, be significantly larger, and more complex, than our solution.

Before concluding our design discussion, we note a limitation of our solution. Specifically, for certain network architectures, our solution might disrupt traffic to the customer, while waiting for the CPU on the target router to idle. This will occur in architectures where a route to the IP address of the customer-facing interface is propagated in to the ISP backbone. In such cases, the initial router will re-route customer-bound traffic to the target router, before the target router is ready to forward such traffic to the customer. Due to the fact that such architectures increase the size of the routing table for ISP routers, however, we expect them to be uncommon.

7.3 Simplifying ZIRO

While we have demonstrated a rehoming solution that can achieve sub-second outage times, it includes a rather intricate set of changes to improve the scheduling policy of `bgpd`, as well as some smaller changes to improve the CPU consumption of `zebra`, `bgpd`. While we believe that none of the changes are overly burdensome, we would like to minimize the changes that our solution requires.

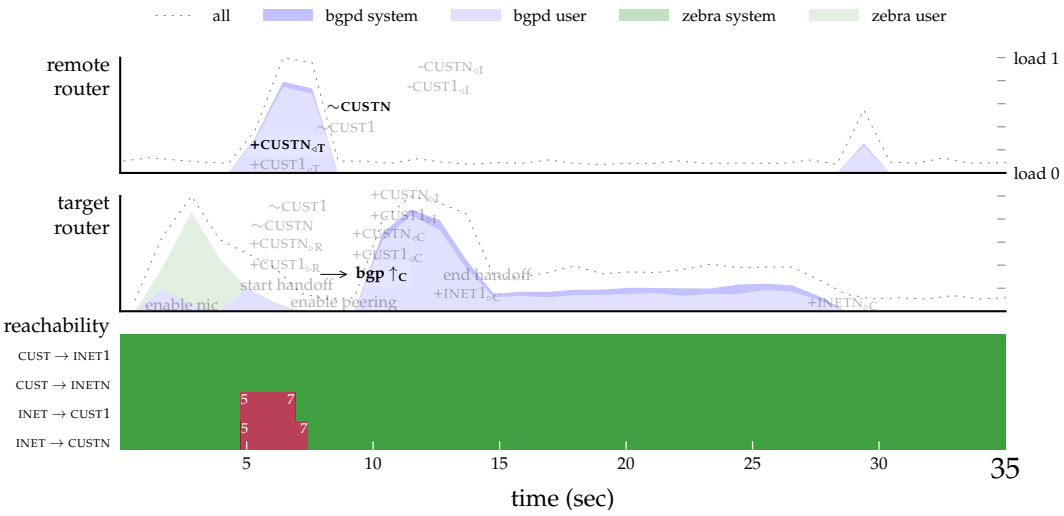
Accordingly, in this section, we evaluate the performance of soft handoff when these changes are omitted. We first remove the changes to improve the scheduling behavior of `bgpd`, and show that the outage times do not increase. We then repeat the exercise for our changes to the improve the CPU consumption of `zebra` and `bgpd`. We again show that outage times do not suffer, though the total time required to complete rehoming does increase.

7.3.1 Removing changes to scheduling policy

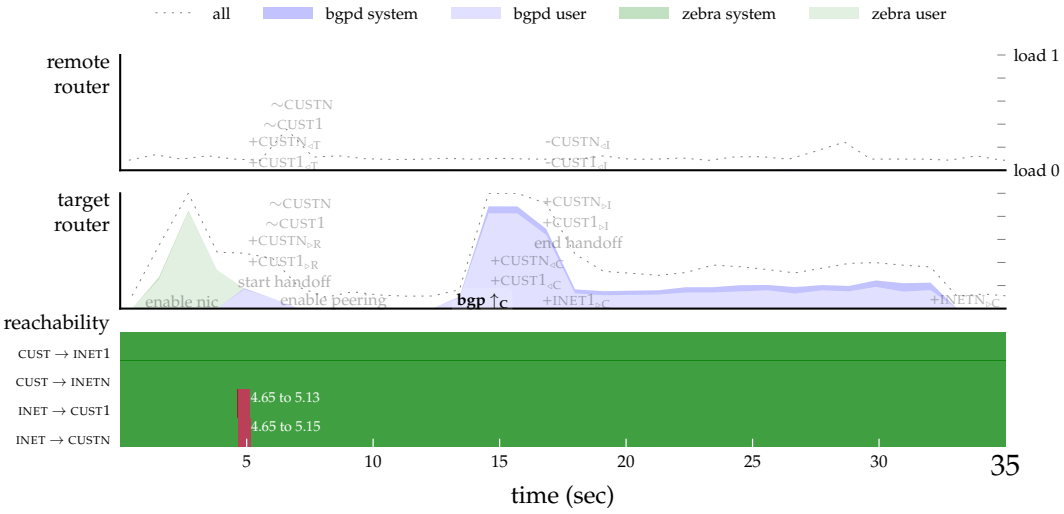
In our attempt to simplify our solution, we first remove the scheduling changes to `bgpd`. This reverts all of the patches introduced in Chapter 6. We present the mean outage times that result from reverting these patches in Table 7.4. We find, rather surprisingly, that the mean overall outage time has decreased, from 0.70 seconds to 0.43 seconds.

To understand the source of this apparent improvement, we consult the system charts for rehoming, before and after the removal of the patches of Chapter 6. We present the system charts for the trials with the maximal overall outage times as Figure 7.6. We note that the maximal outage time with the scheduling changes present is approximately 2 seconds, while the maximal outage time with the changes removed is approximately 500 ms.

This significant difference in outage times for the trials with the maximal outage times suggests that unexpected improvement may, in fact, be due to the out-lier observed in the experiment with the scheduling patches present. To verify this hypothesis, we examined the outage time for the remaining nine trials. Amongst these trials, we found a maximal outage time of 660 ms, and a



(a) with changes to bgpd scheduling policy



(b) without changes to bgpd scheduling policy

Figure 7.6: Partial system charts comparing ZIRO with and without changes to bgpd scheduling policies. For the complete system charts, see Figures A.22 and A.23.

	before removal	after removal
internet to CUST1	0.64	0.41
internet to CUSTN	0.69	0.42
customer to INET1	0.04	0.05
customer to INETN	0.05	0.05
any	0.70	0.43

Table 7.4: Comparison of mean outage times, in seconds, and over ten trials, before and after removal of scheduling policy changes.

	before removal	after removal
internet to CUST1	0.41	0.42
internet to CUSTN	0.42	0.41
customer to INET1	0.05	0.05
customer to INETN	0.05	0.05
any	0.43	0.43

Table 7.5: Comparison of mean outage times, in seconds, and over ten trials, before and after removal of CPU optimizations.

mean outage time of 470 ms. We thus conclude that the apparent improvement is, in fact, due to experimental variation.

As for the source of this variation, we examine sub-figure (a), and note that the increased outage time can be attributed to the delay between the **+CUSTN**$_{\pi}$ and $\sim$**CUSTN** events on the remote router. Noting that the CPU on the remote router is active during much of this time, we hypothesize that the delay observed in this trial is due to periodic processing. This might, for example, be due to the `bgp_scan` function discussed in Section 7.2.2.

Though we have argued that removing our scheduling policy changes for `bgpd` has not actually improved outage times, a question remains. That is, why do outage times not increase upon the removal of those optimizations? The answer is simply that, with soft handoff in place, the **bgp** $\uparrow_c$ event, and all subsequent processing by `bgpd` on the target router, are no longer on the critical path.

7.3.2 Removing CPU optimizations

Having shown that our changes to the scheduling policies of `bgpd` can be removed without increasing outage times, we now turn to the CPU optimizations we introduced in Chapter 5. We remove the patches introduced therein, and repeat our experiment. We present the mean outage times for this condition in Table 7.5. Comparing the outage times before and after this change, we observe that outage times do not increase due to the removal of our CPU optimizations.

To explain why the CPU optimizations do not affect outage times, we compare the system charts for the experiments with and without the CPU optimizations, for the trials with the maximal overall outage times. We present these charts in Figure 7.7. We first observe that, although `zebra` requires significantly more CPU time without the CPU optimizations, such time elapses between the **enable**

	CMU	Google
internet to CUST1	0.42	0.43
internet to CUSTN	0.41	0.43
customer to INET1	0.05	0.05
customer to INETN	0.05	0.04
any	0.43	0.44

Table 7.6: Comparison of mean outage times, in seconds, and over ten trials, for CMU (AS 9) and Google (AS 15169). Google originates 129 prefixes, whereas CMU originates just 7.

nic event on the target router, and the **move link** event on the bridge. Accordingly, it is not on the critical path. Similarly, the additional CPU time used due to sub-optimal hash table sizing in `bgpd` occurs after the **bgp** $\uparrow_c$ event, which itself occurs after soft handoff has completed.

Our examination of system charts explains why our CPU optimizations do not affect outage times, with the soft handoff scheme. We note however, that the system chart of Figure 7.7(b) illustrates the cost of eliminating the CPU optimizations. Specifically, the time to complete rehoming increases significantly. We briefly note that reintroducing only the `zebra` optimization patch of Listing 5.11, and the session establishment patch of Listing 6.2, would reduce the mean time between the **enable nic** and **+INETN**$_{\triangleright c}$ events from 48.95 seconds to 29.78 seconds, while still avoiding the most intricate changes. We leave a fuller examination of completion times for future work.

7.4 ZIRO at Scale

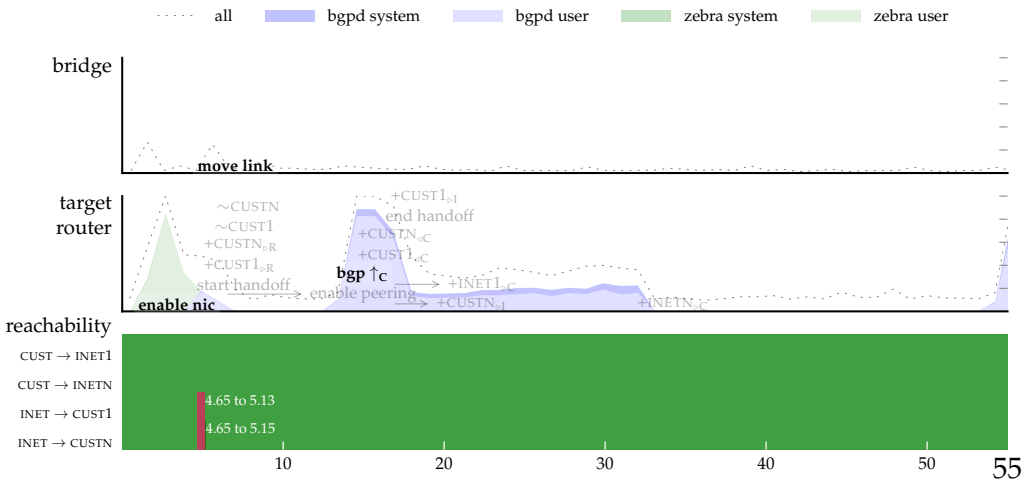
Thus far, we have conducted all of the experiments in this dissertation with a customer trace containing 7 prefixes. As noted in Section 2.1.3, this represents an autonomous system at the 82^{nd} percentile, in terms of the number of prefixes originated. Given the results we have achieved to this point, we now consider the performance we can achieve for larger autonomous systems. In particular, we use a customer trace based on routes announced by AS 15169 (Google). This trace contains 129 prefixes, and ranks at the 99^{th} percentile.

We present the mean outage times with this trace in Table 7.6. We find, perhaps surprisingly, that the 17-fold increase in the number of prefixes originated does not dramatically increase mean outage times. Specifically, we observe an increase of only 10 ms. To understand why this is the case, we repeat the experiment, profiling the `bgpd` process using OProfile.[7] We present the top five functions, in terms of CPU time used during soft handoff, in Table 7.7.

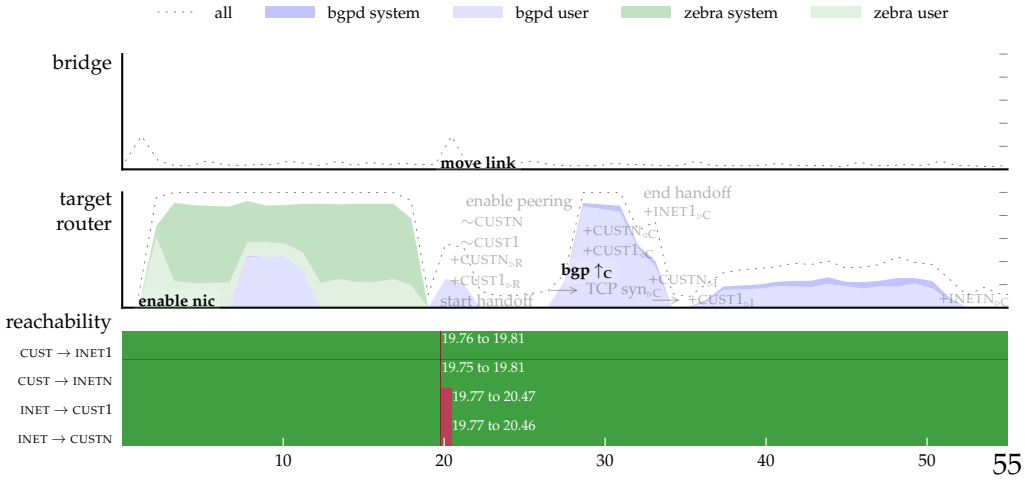
Examining the CPU time reported in Table 7.7, we observe that the function `bgp_route_next` accounts for approximately one-third of the down time observed. Based on our reading of the source code, we hypothesize that this function is a hot spot due to the frequency with which it is called. Specifically, during the soft handoff process, the main loop of `bgp_soft_reconfig_table` iterates through the entire routing table, of over 300,000 entries.

We leave validation of this hypothesis, and optimization of this bottleneck for future work. Before proceeding, however, we note two possible optimizations. First, and most obviously, we might optimize the running time of `bgp_route_next` itself. If this were to prove infeasible, however, we might improve performance by splitting soft handoff into two phases. The first phase, to be

[7] See Section 5.1 for further details on OProfile.



(a) with CPU optimizations



(b) without CPU optimizations

Figure 7.7: Partial system charts comparing ZIRO with and without CPU optimizations in `zebra` and `bgpd`. Note that subfigure (a) repeats Figure 7.6(b), but with the time scale adjusted to match subfigure (b). For the complete system charts, see Figures A.23 and A.24.

function	object file	CPU seconds
bgp_route_next	bgpd	0.15
bgp_scan_timer	bgpd	0.09
bgp_soft_reconfig_table	bgpd	0.07
work_queue_add	libzebra.so.0.0.0	0.02
bgp_unlock_node	bgpd	0.01

Table 7.7: Top five functions called by bgpd during soft handoff, when rehoming Google (AS 15169). Values reported are means over ten trials.

executed prior to link migration, would scan the routing table, identifying the routes that need to be processed during handoff. The second phase, to be executed as the link is moved, would execute the actual handoff.

7.5 ZIRO Interruption

Thus far, we have evaluated our rehoming solutions in terms of the five-nines reliability goal. After demonstrating a solution that met that goal, in Chapter 6, we argued for the value of further improvements. In particular, we contended that sub-second outage times would make rehoming transparent to even demanding real-time applications. Having achieved sub-second outage times, we now substantiate our claim.

Herein, we consider the disruption caused by ZIRO, for three types of network traffic: web browsing, streaming video-on-demand, and real-time video conferencing. As web browsing and video-on-demand both employ TCP as their transport protocol, we first examine empirical TCP behavior during a ZIRO event. We then evaluate the end-user impact of this behavior, arguing that it will not significantly degrade the web-browsing, or video-on-demand, experience. Finally, we consider video conferencing, again concluding that ZIRO will not significantly degrade the end-user experience.

7.5.1 Impact of ZIRO on TCP streams

To evaluate the impact of ZIRO in TCP streams, we repeat the experiment of Section 7.3.2, with some additional instrumentation. Specifically, on the "customer sink" and the "sink" nodes, which previously measured end-to-end connectivity via ICMP messages, we add a pair of nuttcp processes, and another tcpdump process. The nuttcp process on the sink generates a constant bitrate TCP stream which is received by the corresponding process on the customer sink. The tcpdump process, on the customer sink, captures the packet headers for this stream.

During a ZIRO event, our TCP stream will experience lost packets, employ standard loss detection and recovery mechanisms to retransmit the lost packets, and then employ standard congestion avoidance and control mechanisms to resume transmission at the maximal rate permitted by the application, end-hosts, and network.

Given this behavior, we quantify the disruption caused by a ZIRO event by computing the elapsed time between the first lost ICMP packet from the Internet to custn, and the first 100 ms interval during which the customer sink receives stream data at a rate greater than, or equal to, the

	bitrate (Kbps)						
round-trip time	1500	3000	6000	12000	24000	48000	96000
20 ms	0.67	0.90	0.94	1.20	1.32	1.32	1.33
50 ms	1.19	1.13	1.25	1.30	1.31	1.63	1.76
100 ms	1.34	1.55	2.04	1.71	1.49	5.58	3.61

Table 7.8: Disruption experienced, in seconds, and over ten trials, for a constant bitrate stream delivered over TCP. Each row provides data for a given round-trip latency, while each column provides data for a given bitrate. For example, the mean latency for a 1500 Kbps stream, delivered over a connection with 20 ms round-trip latency, was 0.67 seconds.

average data rate from the start of the stream until the loss of connectivity. Note that this overstates actual disruption, as it ignores the data transmitted as TCP ramps up to the full bitrate.

We evaluate this disruption for three different round-trip latencies, and seven different bitrates, presenting the results in Table 7.8. Overall, we find that the mean disruption time varies between 0.67 and 5.58 seconds. With respect to trends, we find that disruption time increases with both round-trip time, and stream bitrate. These trends are as expected, given TCP's algorithms for congestion avoidance and control.

7.5.2 Impact of ZIRO on TCP applications

Given our empirical results, above, on the transport-layer disruption impact of rehoming, we now consider the application-layer impact of rehoming. We consider two common interactive applications, both of which employ TCP as their transport protocol: web browsing, and video-on-demand. We first consider the impact for these applications in today's networks, and then speculate on the impact in future networks, with greater link capacities.

For today's networks, we assume the 5 Mbps average throughput reported by Akamai Technologies for client requests in the United States [5], and the 34.2 ms mean latency reported by AT&T, for its US backbone [9]. Given these parameters, our empirical results in Table 7.8 predict a mean disruption of 2 seconds or less.

For web browsing, we acknowledge that a delay of 2 seconds is significant. However, much prior work demonstrates that users will tolerate delays of between 8.5 and 15 seconds before experiencing dissatisfaction or losing focus, with higher tolerances when the system provides indication of progress [12, 13, 41, 63]. Thus, the additional delay experienced during a rehoming event will be tolerable for all but those web pages already near the threshold of user dissatisfaction.

For video-on-demand, we note that applications typically maintain a buffer of received data, specifically to avoid disruption from network effects such as loss and jitter. Thus, the disruption caused by ZIRO depends on the amount of video buffered by the client. In particular, if the client buffers more than 2 seconds of video, the user will experience no disruption at all. Noting that the Adobe Flash video player, which dominates the online video market [2], defaults to buffering 9 seconds of video [1], we conclude that the outage time observed during ZIRO rehoming will not interrupt most video-on-demand streams.

Having argued that ZIRO rehoming will not significantly degrade the end-user experience for web browsing or video on demand, in today's networks, we now consider the impact of ZIRO rehoming in future networks. Specifically, we note that our empirical results, in Table 7.8, indicate

mean disruptions of up to 5.58 seconds, for very high bitrate streams over high latency connections. Accordingly, techniques to minimize latency, such as the use of Content Delivery Networks, may be required to mask the disruption of rehoming.

7.5.3 Impact on video conferencing

Turning to video conferencing applications, we note an important difference between video conferencing, and other network applications. Namely, for video conferencing, timeliness is more important than reliable delivery. Accordingly, video conferencing applications are likely to use UDP for their transport protocol, rather than TCP.

With UDP, the data lost due to rehoming will not be retransmitted. Instead, the end-user will observe 0.43 seconds of frozen video, and silent audio. While this is certainly long enough to be perceptible, we expect the glitch to be short enough that affected participants can simply ask the other parties to repeat themselves.

7.6 Conclusion

In this chapter, we set out to achieve sub-second outage times during rehoming. Our key insight towards this goal, presented in Section 6.5.3, was that much of the remaining down time, following all of our previous optimizations, was due to the customer router. In particular, the target router could not propagate the customer routes upstream more quickly, simply because the target router had to wait to receive the routes from the customer router.

To resolve this problem, we designed an implemented a soft handoff procedure. This procedure leverages the fact that, by virtue of the redundancy inherent in network routing, the target router can determine the customer routes before the establishment of a peering session with the customer router. To do so, the target router leverages the fact that these routes will have been advertised to the target router by the initial router.

Empirically, we demonstrated that soft handoff could reduce mean outage times to sub-second levels. Specifically, for an autonomous system originating 7 prefixes, we observed a mean overall outage time of 0.70 seconds. Simplifying our solution, by removing the scheduling policy changes of Chapter 6, and then the CPU optimizations of Chapter 5, we observed a mean overall outage time of 0.43 seconds. We argued that this was not an improvement due to the simplification, but simply that the difference was due to experimental variation.

We then investigated the performance of our solution for an autonomous system originating a larger number of routes. To do so, we repeated our experiments using a trace of AS 15169. This system originates 129 prefixes, a 17-fold increase over the 7 prefixes originated in all previous experiments. We found, however, that the outage times did not increase significantly. We hypothesized that the outage time was dominated by other factors, such as the time taken to scan the full routing table, to identify the routes associated with the customer.

ZIRO readily achieves the five-nines availability goal. Assuming that customers are moved to a temporary router and back, and assuming historical rates of routing software upgrades, ZIRO can enable an IP network operator to keep its network up to date, using only 1.3% of the annual outage budget. Additionally, if ZIRO were extended to apply to other sources of down time, it could deliver five-nines availability even if a router failed, or required maintenance, nearly every day.

In addition to achieving the five-nines availability goal, ZIRO causes only minimal interruption of service during rehoming. This is true for a variety of activities, such as web browsing, video-on-demand, and even real-time video conferencing. Web browsing activities will experience some delay. However, unless server response time is already near the user tolerance threshold, the additional delay does not impact user satisfaction. Video-on-demand experiences a similar delay, which can be readily accommodated with standard buffering techniques. Video conferencing applications experience data loss. These losses will, however, be brief enough that participants can simply ask their peers to repeated the missed portion of the conversation.

Begin at the beginning and go on till you come to the end; then stop.

Lewis Carroll

This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

Winston Churchill

8 Dénouement

In this dissertation, we have made significant progress towards understanding, and resolving, one of the key obstacles to a highly reliable Internet. From Chapter 2, where we established baseline performance, through Chapter 7, where we presented a rehoming solution with sub-second outage times, we have exploited rigorous experimentation, and the study of source code, to understand and reduce the down time caused by routing software upgrades. The net result of our efforts is a practical and economical method to mitigate the down time caused by such upgrades, and thereby greatly improve Internet reliability.

Having achieved this significant advance, we now consider our work in a broader context. We begin by abstracting our rehoming solution, to identify design principles for rapid recovery of routing sessions. We then discuss related work, including alternative solutions, and other studies of BGP behavior. Next, we offer suggestions for future work, both closely related to our rehoming solution, and farther afield. Finally, we offer our concluding remarks.

8.1 Design Principles

In order to identify design principles for rapid recovery from the failure of routing sessions, we abstract from the design of ZIRO, as presented in Chapter 7. We focus on the simplified solution of Section 7.3.2, as this minimal solution captures the essential elements of our technique. We enumerate the resulting design principles, along with the solution elements which they generalize, in Table 8.1.

We observe that our solution exhibits three key design principles: manual control, synchronous processing, and the aggressive exploitation of best-available data. We next elaborate each of these principles. Note that, in some cases, our application of these principles contradicts normal-case router behavior. Accordingly, our elaborations include arguments for such choices, as appropriate.

design principle	example	rationale
provide manual control	BGP LOCAL_PREF attribute	Sections 4.4 and 4.5
	reflection policy override	Section 7.1.2
	router-id spoofing	Sections 3.2.5 and 4.3
	end-of-RIB log message	Section 7.1.2
allow synchronous processing	synchronous update processing	Sections 6.4 and 7.1.2
	synchronous update propagation	Sections 6.3 and 7.1.2
exploit best-available data	retained routes in Graceful Restart	Section 3.4, Chapter 4
	IBGP routes in Soft-Handoff	Sections 6.5.3 and 7.1

Table 8.1: Design principles for rapid recovery of routing sessions

Manual Control

The first design principle of Table 8.1 is manual control. We recommend manual control for recovery scenarios because autonomous controls, lacking cross-layer knowledge of system state, are often designed to behave conservatively. For example, following the failure and recovery of a link, a router might delay reestablishing a BGP peering over the restored link. Doing so would limit route flapping, in the case the link failed again.

In operator-initiated recovery scenarios, however, the operator may have broader system knowledge that renders such safety measures unnecessary. For example, if a peering failure is caused by a maintenance activity, the new peering can be established as soon as the maintenance is complete. Or, in our rehoming scenarios, the peering can be established as soon as the customer router has been rehomed to the target router.

Synchronous Processing

The second design principle of Table 8.1 is synchronous processing. Synchronous processing runs counter to the default behavior of Quagga, and some versions of IOS, which queue updates for batched, asynchronous processing. Such processing may make more efficient use of CPU cycles, by reducing overheads and collapsing multiple inbound updates into a single outbound update.

While efficient use of CPU resources is important, rapid recovery, requires that other factors be considered. First, as observed in Chapter 6, asynchronous processing can complicate the proper prioritization of critical messages over less urgent messages. Second, timer-driven batched processing introduces undesirable latency into the recovery process.

Best-Available Data

The second design principle of Table 8.1 is exploitation of best-available data. In general, to avoid problems such as persistent black-holes, routing systems should rely only on data that is known to be current. In soft-state protocols, this is ensured by timing out routing state that has not been refreshed. In hard-state protocols, such as BGP, this is ensured by removing routes when a peering fails.

As noted in Table 8.1, both Graceful Restart and Soft Handoff exploit best-available, though possibly stale, data. Accordingly, they may cause the black-holing of some traffic during recovery. However, both techniques limit the time during which they rely on the potentially stale data. Thus, they will not induce persistent black-holes.

8.2 Related Work

Given the importance of the Internet to contemporary society, it should come as no surprise that our work is far from the first to tackle the problems of understanding and improving Internet reliability. Herein, we discuss the most salient of the prior work. We begin our discussion with work on methods to improve Internet reliability. We then discuss work towards understanding the behavior of BGP, as it significantly influences Internet reliability. We conclude our discussion of related work with pointers to information on the design and implementation of BGP routers, and a comparison of our system charts to previous visualizations.

8.2.1 Internet Reliability

Successful end-to-end communication on the Internet depends on the correct, or sufficiently correct, operation of many components. These components include the networking software on end hosts, physical links, routers on the path between the end hosts, and ancillary services such as domain name servers. Based on studies which indicate that IP routers are a significant source of overall down time [43, 68], this dissertation focused on reducing the down time caused by failures and maintenance of IP routers. Accordingly, we focus herein on work which addresses the down time caused by router, or routing, failures.

General Reliability

A natural method for mitigating the impact of failures is to employ redundancy. In the context of IP networking, this can be accomplished with various means, depending on the location of the fault. These means include multi-homing, redundant backbone links, and overlay networks. Multi-homing addresses failures of access links and access routers, redundant backbone links address the corresponding link and router failure, and overlay networks address a broad range of routing failures, including the preceding failures as well as peering link and peering router failure. We elaborate each method in turn.

As the name suggests, a multi-homed network maintains multiple connections to the Internet at large. This might, for example, mean subscribing to service from two ISPs, such as AT&T and Sprint. When the link to one ISP fails, the customer router re-routes traffic through the other ISP. This provides immediate recovery for outbound traffic. Inbound traffic, will however, be disrupted, as BGP propagates the routing change through the Internet.

Backbone link redundancy masks the failure of ISP-internal links from external networks, such as customers and peers. When a backbone link fails, and an alternative path is available, the interior gateway protocol, such as OSPF [64], routes around the failed link. Additionally, network operators can manually exploit backbone link redundancy, to reduce downtime during maintenance, by "costing-out" a link. This method simply increases the cost metric for links incident to the router subject to maintenance, causing other routers to avoid paths through that router.

Overlay networks, such as RON [6] and MONET [7], address a broader class of faults, subject to the existence of alternate routes through overlay members. Whereas multi-homing and backbone link redundancy improve reliability by changes to IP routers, overlay networks construct a logical network using end-hosts. These Internet end-hosts operate as overlay routers, cooperating to identify overlay routes that offer better availability or performance than direct Internet routes.

While all of the above schemes improve Internet reliability, none of them address outages caused by access link failure, or access router failure, for singly-homed customers. Solutions to address this class of Internet reliability problems fall in to two broad groups: those that improve the reliability of a single router, and those that redirect around failed access routers. We first detail related work on router reliability, and then detail work on avoiding failed access routers.

Router Reliability

Router vendors have proposed a number of solutions for improving IP router reliability, generally through the use of 1:1 hardware redundancy. We focus here on the offerings of Cisco, the dominant vendor of IP routers.[1] High reliability offerings from Cisco include Automatic Protection Switching [22], Route Processor Redundancy [19], Router Processor Redundancy Plus [20], Stateful Switchover [11, 21], and Nonstop Forwarding [11, 18, 23]. We next elaborate each, in turn.

Automatic Protection Switching [22] employs redundant line cards, to improve reliability in the face of line card, or link, failures. Both line cards are connected to an add-drop multiplexer, which duplicates inbound data. In the event that the primary ("working") card, or link, fails, the router detects the failure, and switches to the backup ("protect") link. APS is expensive to implement, as it requires a dedicated backup network port for every primary network port to be protected. Moreover, it does not address down time due to software upgrades, software failures, or failures of other hardware components.

Router Processor Redundancy [19] employs redundant route processors to improve reliability in the face of control plane failures. In the event that the hardware or software on the primary route processor fails, the secondary route processor assumes control of the system. The secondary route processor then completes the system bootstrap process. This includes reading configuration data, initializing line cards, and establishing BGP peerings. Because most resources are reset, RPR can provide only modest improvements in reliability. In particular, it eliminates the time that would have been required to boot the primary route processor to the equivalent of the standby state.

Route Processor Redundancy Plus [20] improves on RPR, by synchronizing configuration state across the primary and secondary route processors. This enables the secondary processor to proceed farther in the boot process, thereby reducing the switchover delay. Additionally, on some routers, RPR+ eliminates the line card reboot. RPR+ does, however, still require the clearing of FIB tables, and the reset of BGP peerings. Given our experimental results in Chapter 2, regarding the time required to establish and quiesce BGP peerings, it is unlikely that RPR+ could enable five-nines reliability.

Stateful Switchover [11, 21] improves on RPR+, by synchronizing some line card and link layer protocol state between the primary and secondary route processors. This synchronization avoids the need to re-establish layer-2 links with peers. Additionally, on the 7500 series and 12000 series routers, SSO eliminates the clearing of FIB tables during route processor switchover. In [21], Cisco reports that the enhancements enable the router to resume packet forwarding 7 seconds after a switchover. However, as with RPR+, BGP peerings must be re-established after the switchover.

[1] For details on the offerings of other vendors, we recommend Reardon's article on IP reliability [68].

Nonstop Forwarding [11, 18, 23] improves Stateful Switchover with the addition of Graceful Restart support. As we have shown, Graceful Restart dramatically reduces the impact of BGP peering loss. Thus, NSF could be used to minimize the outage caused by routing software upgrades. Indeed, testing by the European Advanced Networking Test Center has shown that some Cisco routers can support software upgrades with down times of 45 nanoseconds, or less [27, 72].

With outages of 45 nanoseconds, NSF clearly provides less disruptive failover than our rehoming solution. However, NSF suffers from two important limitations. First, NSF requires redundant hardware in each router. Second, NSF can not be readily extended for use in other scenarios, such as chassis failure, or disaster recovery.

In contrast, our rehoming solution improves reliability through the use of a separate router. This provides two benefits. First, it enables a single router to serve as a backup for multiple routers. Such multiplexing reduces capital costs over solutions requiring redundant components in each router. Second, because our recovery process does not depend on any components of the failed router, our solution can be extended for use in scenarios such as chassis failure, and disaster recovery.

Avoiding Failed Access Routers

In order to reduce the cost of providing high availability, as well as to provide a more general solution, a number of researchers have proposed solutions based on link migration. These solutions first leverage the reconfigurability of the transport network, which provides the physical layer link between the customer router and the ISP access router, to migrate the physical layer link to an alternate access router.[2] The solutions then employ a variety of techniques to minimize the impact of higher layer reconfiguration. We discuss the most closely related work herein.

Early Work The earliest work on IP link migration, of which we are aware, is a proposal by Sebos et al. [74] for reducing the outage time caused by access router failures. The experimental evaluation therein quantifies the outage experienced as an access router fails, and a customer is migrated to a backup router. The data in the paper indicate an overall outage time of approximately 60 seconds in a simple implementation, or approximately 20 seconds with the use of Graceful Restart.

A direct comparison to our results is difficult, due to the difference in scenarios, and the fact that the paper does not report important parameters, such as the routing table size, the routing software used, and the CPU speed. Nonetheless, we note that the 20 second outage time observed with Graceful Restart is similar to our result in Section 4.5.2. Our end result in Chapter 7 achieves a much lower overall outage time, of approximately 0.43 seconds.

In addition to improving outage times, our work presents another important benefit over that of Sebos et al. In particular, the solution proposed in [74] requires modification of the customer router, to avoid the router-id conflict we discuss in Section 3.2.5. In our work, we resolve the conflict by introducing router-id spoofing, which only requires modifications on the target router. Thus, it is possible for customers to enjoy the benefit of our solution, without having to first upgrade their own routers. Or, viewed from the ISP perspective, it enables an ISP to offer an improvement that requires no work on the part of the customer.

RouterFarm In our own previous work [4], called RouterFarm, we evaluated link migration with commercial routers. Using Cisco routers, and a SONET physical layer, we reported mean overall

[2]Note that we refer to the links provided by the transport network as physical layer links, reflecting the perspective of the IP routers.

outage times of approximately 60 seconds with 10 customer routes, and 10 ISP routes. With 5000 customer routes, and 5000 ISP routes, this outage time increased to approximately 80 seconds. While we made efforts to understand the cause of the observed outage times, based on external observations, the closed nature of the commercial routers made it difficult to reach a deep understanding of underlying causes.

Our work here both deepens our understanding of system behavior, and improves overall outage times. Exploiting the open nature of the Linux operating system, and the Quagga routing software suite, we captured extensive data on system behavior. By examining this data, and the source code for Linux and Quagga, we were able to explain and improve CPU consumption, and scheduling behavior. Furthermore, the insights from those investigations enabled us to design a solution that forgoes the CPU optimizations, and scheduling changes, while achieving mean overall outage times of less than one second.[3]

VROOM In [83], Wang et al. propose the VROOM scheme for router migration. This scheme achieves migration by decoupling logical routing functionality from the hardware on which the functionality is realized. VROOM implements router migration through a combination of transport network link reconfiguration, virtual machine migration, and the use of a data plane hypervisor. The data plane hypervisor abstracts the interface between control and data planes, enabling data plane migration across different data plane architectures. Their experimental evaluation demonstrates that their prototype implementation can achieve zero packet loss.

Unfortunately, however, VROOM imposes a burdensome requirement on the transport network, hinders the evolution of network services, and introduces a non-trivial amount of new complexity. Specifically, because VROOM migrates at the router granularity, migration requires transport network capacity in proportion to the total traffic flowing through the router. Additionally, new services can not be deployed until hypervisor support for these services is standardized.[4] Finally, VROOM introduces significant complexity, due to the need to maintain synchronization of data plane state, across the initial and target routers, during the migration process.

Rather than take the extreme position that migration be fully transparent to network traffic, we leverage the "best-effort" IP service model, and construct a solution that is practically transparent. Our solution supports link-by-link migration, which reduces transport network capacity requirements. Additionally, our solution introduces no new synchronization requirement across routers. We note that our solution does require ongoing support, in terms of Graceful Restart support for new BGP address families. However, such support is already required for existing vendor offerings, such as Cisco's Nonstop Forwarding.

Router Grafting In [50], Keller et al. propose a BGP session migration scheme called Router Grafting. This technique support link-by-link migration through the use of reconfigurable transport networks, and control plane state migration. The physical layer migration technique is similar to other schemes. Control plane state is migrated by serializing TCP and BGP state at the initial router, and deserializing this state at the target router. Router Grafting uses tunneling to minimize the outage caused by the transient inconsistencies between physical layer links, and higher layer

[3] The outage time achieved is not directly comparable to our results in [4], due to differences in hardware, software, and routing table sizes. Note, however, that our unoptimized solution, of Chapter 3, experienced a mean overall outage time greater than that in our previous work, while our end result achieves overall outage times substantially less than our previous work.

[4] Alternatively, routers using new services can not be migrated until hypervisor support for these services is available.

state. The paper quantifies state serialization and deserialization time, but provides no end-to-end measurements of packet loss.

A key challenge for Router Grafting is that, while BGP state is reasonably well-defined, TCP state is not. This arises because while only a single version of BGP is widely used, TCP has many variants. These variants differ in their congestion control algorithms [14, 28, 34], data acknowledgement behavior [28], security features [40], or other respects. Accordingly, the widespread deployment of Router Grafting depends on the standardization and debugging of techniques for migration of TCP sessions across systems that differ in their support for these features.

In addition to facing this important deployment obstacle, Router Grafting introduces changes that may compromise network reliability and performance. Reliability is threatened, because BGP is a hard state protocol. Thus, any control plane state corruption caused by errors in the design or implementation of the migration code may persist for an unbounded period of time. Performance is challenged by the use of tunneling to mask transient cross-layer inconsistencies. This change in data plane configuration might cause packets to be processed in software, rather than using optimized hardware paths.

Our approach, which simply initializes fresh control plane state, avoids the deployment and reliability challenges posed by state migration. Our use of Graceful Restart mechanism does introduce the possibility of transient inconsistencies. However, because Graceful Restart retransmits all routing data, and subsequently clears previous state, the system converges to the correct state after the exchange of routing data between the target router and the customer router. Additionally, because our approach minimizes the duration of routing inconsistencies to levels that are practically transparent, it does not require data plane reconfiguration that might compromise performance.

8.2.2 BGP Behavior and Performance

One of the keys to developing our rehoming solution was understanding BGP behavior and performance. Of course, given the importance of BGP to the Internet, our work is far from the first to investigate these aspects of BGP in practice. Previous work has included both, studies of large-scale behavior, and studies of detailed, small-scale behavior. We first discuss the studies of large-scale behavior, to provide broad context. We then discuss prior work on small-scale behavior, for its direct relevance to our work.

Large-scale behavior and performance

Amongst the earliest studies of large-scale BGP behavior is work by Labovitz et al., which characterized routing updates observed at public Internet exchange points between 1996 and 1998 [55, 56]. Their work found that most routing updates were pathological, in the sense that they did not reflect a change in a route's availability or path attributes. Based on discussions with router vendors, their work identified routing software changes to reduce the frequency of pathological updates.

Later work by Labovitz and colleagues quantified Internet route convergence behavior through a fault injection study [52]. This study found that changes announcing a new, or improved, route generally converged in 90 seconds; changes withdrawing a route generally required 2 minutes or more. The paper also reports that BGP can, in the worst case, require the exchange of $n!$ messages before reaching convergence, where n is the number of autonomous systems in a network. The paper explains that use of a MinRouteAdvertisementIntervalTimer can reduce the number of messages exchanged during reconvergence, but may extend the time required for reconvergence.

In [38], Griffin and Premore studied the effects of the MinRouteAdvertisementIntervalTimer through simulation. Analyzing four different network topologies, and several different network sizes, they showed that each network had an optimal setting for the timer interval. Below these optimal values, more messages are exchanged, and convergence is significantly delayed. Above these optimal values, the number of messages does not change significantly, but convergence time increases.

In [76], Teixeira et al. examined the interaction between Interior Gateway Protocol changes, and BGP changes. Their work included both, a large-scale field study, and small-scale testbed experimentation. In the field study, of 176 days of routing data from the AT&T backbone, they found that BGP generally reacted to IGP changes within 80 seconds. In the testbed setting, they observed similar delays, of up to 70 seconds, and hypothesized that this delay was due to timers in the routing software.

Small-scale behavior and performance

Amongst studies of small-scale behavior, the work most relevant to our won is that of Wu et al. [85]. In a benchmark scenario which exercised routers similarly to the way Graceful Rehoming exercises the customer router, they observed update processing rates of 10–3400 updates per second, depending on router hardware and software. For XORP running on an 800 MHz Pentium III processor, they reported a processing rate of approximately 1100 updates per second. Our results in Figure 4.7(b), using Quagga and an 850 MHz Pentium III processor, indicate a processing rate of over 22,000 updates per second. This suggests that significant differences exist in the performance of different BGP implementations.

Other measurements on BGP processing speed include a test of the Cisco CRS-1 router by the European Advanced Networking Test Center [72], and a field study of BGP table transfer time by Zhang et al. [87]. The former reports that the CRS-1 router was able to process 500,000 routing updates in 12 seconds, for a rate of approximately 42,000 updates per second. The latter analyzes data from peering sessions between RouteViews [81] and a number of ISPs, finding that most table transfers take either 20 to 70 seconds, or 200 to 500 seconds. Direct comparison with these results is not possible, due to differences in, or uncertainty about, router hardware.

Other work on BGP performance, more broadly, includes Maennel et al. [59], which described the design, implementation, and validation, of a tool for generating benchmark BGP workloads; Feldmann et al. [30], which experimentally measured the time required for a Cisco 12000-series router to propagate a single BGP routing update; Agarwal et al. [3], which studied the CPU load due to BGP processing, in routers across the Sprint backbone; and Chang et al. [17], which quantified the capacity, in number of prefixes, of different BGP routers.

8.2.3 Other Related Work

Much of the technical work in this dissertation, particularly in Chapter 6, has dealt with understanding the design and implementation of the Quagga routing software suite. While we are not aware of any similarly detailed publication on Quagga, some information on its multi-process architecture, as well as its user/kernel interface, can be found in the source code distribution for Quagga itself [47]. For information on other BGP routing software, we recommend the XORP BGP design document [86], slides from the authors of OpenBGPD [15], and the programmer's documentation for BIRD [33]. For case studies on the impact of implementation choices, we recommend the

studies on real-world BGP behavior by Labovitz et al. [52, 55, 56].

Much of our ability to understand the behavior of Quagga's BGP implementation comes from our system charts. These charts bear resemblance to the graphs of per-process CPU activity in Wu et al.'s work on BGP performance [85], and benefit greatly from the ideas and recommendations of Tufte [77, 78, 79, 80]. As compared to the graphs in Wu et al. [85], our system charts graph multiple nodes in the system, thereby providing a broader view of system behavior. Our charts also include annotations of key events across layers in the system, which enable deeper insights into the root causes of the observed behaviors.

8.3 Future Work

Having discussed prior work, we now turn to future work. We group future work in to five areas: additional experimental validation, support for additional usage scenarios, coping with configuration issues in rehoming, generalizing to other BGP routers, and improving BGP implementations. The first four areas are specific to our rehoming solution, while the last area considers broader implications of our work.

8.3.1 Experimental Validation

While we have conducted an extensive experimental evaluation of our ideas, our experiments omit some potentially important real-world phenomena. These include transport network reconfiguration delays, the effect of rehoming on EBGP peers, BGP route reflectors, background BGP traffic, and network latency between ISP routers. We discuss each of the first two phenomena individually, and the remaining phenomena collectively.

With regard to transport network reconfiguration delay, we have simplified our experiment topology by using a programmable Ethernet bridge to emulate physical layer link migration. In practice, this link migration might be accomplished using a number of techniques, depending on access network technologies. We expect that the choice of migration technique will not affect our results, unless the link migration time is greater than the time required for the target router to process its RIB, and advertise routes for the customer prefixes.

Concerning the effect of rehoming on EBGP peers, we note that without due care, a rehoming solution might cause the remote router to withdraw its advertisements for customer prefixes. As explained in Section 7.1.3, our solution takes great care to avoid this possibility. While Section 7.1.3 argued for the correctness of our solution, and the experiments of Chapter 7 do not exhibit the problematic withdrawals, a more direct verification of correctness is possible. To do so, we would add an EBGP peering between the remote router and another router.[5] We would then verify that no withdrawals were sent from the remote router to this new router, during rehoming.

We now turn to the remaining phenomena: route reflectors, background BGP traffic, and network latency between ISP routers. Each of these phenomena influence route propagation delay. Route reflectors and background BGP traffic may increase propagation delay, due to the MinRouteAdvertisementInterval timer. Wide-area network latency will increase outage times, as that latency directly increases route propagation delay over the values observed in our testbed envi-

[5]While the remote router does maintain an EBGP peering with the sink node in our topology, this peering is used only to inject Internet routes into the routing system. The remote router is configured not to advertise routes to the sink node.

ronment. We do not expect these factors to compromise the ability to achieve five-nines reliability. They may, however, limit the transparency of our solution for some applications.

8.3.2 Usage Scenarios

Throughout this dissertation, we have focused on rehoming as a technique to reduce the down time caused by planned maintenance, for singly-homed customers. Given the results we have achieved, it may be beneficial to apply rehoming in other scenarios. We now consider three dimensions in which we might extend our rehoming techniques. These are multihomed customers, failure recovery, and peering link protection. We elaborate each, in turn.

Multihomed Customers

With respect to multihomed customers, we expect that our solution will work for these customers, but that it may be more disruptive than necessary. Specifically, our use of LOCAL_PREF, to direct the remote router to prefer routes through the target router, may override existing multihoming arrangements.[6] In particular, if the customer maintains connections to both, the initial router, and the remote router, the use of LOCAL_PREF may cause the remote router to forward customer-bound traffic through the target router, rather than through its own connection to the customer. This complication with rehoming and multihomed customers might be resolved through the use of a Custom Decision Process [71], as we next explain.

The Custom Decision Process of [71] offers operators finer control over path selection than offered by the LOCAL_PREF attribute. In particular, the default BGP Decision Process [69] compares LOCAL_PREF values before any other path attribute, causing LOCAL_PREF to override other considerations. In contrast, the Custom Decision Process allows an operator to specify both, a cost metric, and a "point of insertion". All steps in the default decision process up to, and including, the point of insertion are considered before the cost metric. With the ability to specify that the cost metric be considered immediately before the router-id comparison, our rehoming solution could avoid the router-id complications of Section 4.4, without affecting multihomed customers.

Failure Recovery

Turning our attention to failure recovery, we note that failure scenarios, such as a software crash, present challenges not present in planned maintenance scenarios, such as software upgrades. The fundamental challenge posed by failure scenarios is that we cannot presume that the initial router is functional. This challenge has two important consequences. First, the routing information about customer prefixes may no longer be available on the initial router. Second, delays in migrating one customer may increase the down time for all subsequent customers.

Concerning the first issue, possible strategies for coping with the loss of routing information on the initial router will depend on the nature of the failure. In the case of a fail-stop condition, and assuming that the failure is detected before other routers time out their peering with the initial router, we can continue to exploit the observation of Section 7.1, that the necessary routing data exists on other routers in the network. In the case of Byzantine failure, or if failure detection time exceeds BGP peering time-out thresholds, alternative solutions will be required.

[6] See Section 4.5 for details on our use of LOCAL_PREF.

With regard to the second issue, the downtime experienced by latter customers, when migrating multiple customers off of a failed router, our concerns arise due to two factors. First, a single router may serve many customers. While there is little public data on the number of customers typically homed on a single access router, even a small number of BGP customers per router could limit the viability of our rehoming solution for failure recovery. In particular, given our results of Chapter 7, which show that rehoming completes in approximately 30 to 50 seconds, serially rehoming 10 customers would yield an outage time of greater than 4½ minutes for the last customer.

It may be possible to avoid these extended outages by optimizing scheduling. For example, because Graceful Restart moves the transmission of the Internet routes to the customer routers off the critical path, a rehoming solution for failure recovery might transmit all customer routes to upstream routers, before transmitting any Internet routes to customer routers. Additionally, as the profiling of Section 7.4 suggest that much of the delay for transmitting customer routes to upstream routers is due to the time to scan the RIB, it may be beneficial to transmit all customer routes with a single RIB scan. This would improve on the naïve serial implementation, which requires a RIB scan per customer.

Peering Router Protection

As compared to access routers, peering routers pose a significant performance challenge. The key difference between access routers and peering routers is that access routers receive only a small number of routes from their BGP neighbors. As noted in Section 2.1.3, even Google, which ranks at the 99th percentile in terms of the number of prefixes it originates, advertises only 129 routes. In contrast, peer ISPs will advertise routes to the complete set of Internet prefixes. Thus, peering routers will receive over 300,000 routes from each of their BGP neighbors.

We have conducted some preliminary experimentation with peering router protection, which indicates that our solution can achieve down times of approximately 20 seconds. This is significantly lower than the approximately 140 seconds of down time experienced using the router restart technique, for the much less demanding case of access routers. Nonetheless, given that the failure of a peering router may affect large amounts of network traffic, further performance improvements may be required to make rehoming an attractive option. Further study is required to determine the most promising optimizations.

8.3.3 Configuration Management

As part of the re-homing process, the target router must be configured to serve the customer being rehomed. This configuration includes physical link parameters, IP addresses, and BGP configuration. The configuration process is complicated by the diversity of link types, and the fact that common router configuration languages lack any formal grammar or underlying model [61]. Accordingly, we expect initial use of our techniques will depend on the creation of procedures to handle these tasks for a small set of widely used common services and options, such as described by Gottlieb et al. [36]. Further improvements might be made by applying clustering techniques to router configuration files, to identify common configuration patterns.

8.3.4 Generalizing to Other BGP Implementations

Throughout this dissertation, we have presented empirical results from experiments using the Quagga software routing suite. While focusing on a single implementation enables to develop a deep understanding of its behavior, this focus leaves open to question the generalizability of our results. In particular, would our techniques be necessary, and applicable, for other routers? Though experimental study of these questions is well beyond the scope of our work here, we argue that existing data indicates that our techniques are both necessary, and applicable.

Concerning the necessity of our techniques, we note that one of the key challenges addressed by our solution is that the time taken to process BGP updates, following a session reset, is far too long to afford five-nines reliability. Thus, whether or not our methods are necessary for other routers depends on the speed at which they process BGP updates. Any router that does not process updates significantly faster than Quagga will require some techniques to avoid the downtime caused by BGP processing.

As noted in Section 8.2.2, we have observed that Quagga, running on an 850 MHz Pentium III processor, handles approximately 22,000 BGP updates per second. In a similar scenario, Wu et al. [85] reported that XORP, running on an 800 MHz Pentium III processor, handled approximately 1100 updates per second. Wu et al. [85] also reported that a Cisco 3620 router, with an 80 MHz MIPS R4700 processor, handled about 10 updates per second. Thus, we conclude that other routers will require some method of minimizing the down time caused by BGP processing.

Having argued for the necessity of some optimization on other routers, we now consider the feasibility of implementing our changes on other routes. We note that our changes consist of three sets of patches. The first set resolves issues in Quagga's Graceful Restart support. The second set implements router-id spoofing. The third, and final, set provides support for selectively reflecting routes, and expedites processing of routing policy changes.

We believe the first set of changes will be unnecessary for routers with more mature Graceful Restart support, and that the second set is straightforward. The third set is more challenging, as it requires a deep understanding of potentially intricate scheduling mechanisms. However, given our success in developing a solution, despite the complexity of Quagga's scheduling mechanisms, and without prior knowledge of Quagga's BGP implementation, we believe that porting our changes to other routing implementations will be quite feasible for their maintainers.

8.3.5 Improving BGP Implementations

While important work remains towards the goal of achieving five-nines reliability in the Internet, it would be short-sighted for us not to consider other opportunities to improve the Internet. Thus, we now broaden our perspective, and discuss future work orthogonal to five-nines reliability. Specifically, based on the lessons we have learnt from our work, we discuss three directions for improving BGP routers.

Tracing Improvements

A key challenge in our work has been developing a sufficiently rich understanding of router behavior, to optimize the rehoming process. To that end, we have employed a number of techniques, such as monitoring CPU utilization, monitoring inter-process communication, analyzing hardware performance counter data, and analyzing log file messages generated by the routing software. The last of these techniques was complicated by the sheer volume of data generated. In particular, the

generation of this log file data introduced significant new CPU load, and the analysis of this data required the development of data reduction techniques.

Based on our experience, we believe the data generation and data reduction issues could be eliminated by adding filtering functionality to the logging facility of the routing software. Specifically, many parts of the routing software generate a message for each prefix processed. However, when many prefixes are given the same treatment, detailing the processing of each prefix is unnecessary. Instead, we might choose just to log the processing of the first and last prefixes. Doing so would reveal all of the processing steps, as well as the time required to complete processing, while reducing the quantity of log file data by up to a factor of 150,000.

Scheduling Control

Our work in Chapters 6 and 7 demonstrated that careful scheduling is a key element of optimizing the rehoming process. Specifically, in order to facilitate low impact rehoming, we have made a number of changes to the scheduling behavior of Quagga's BGP implementation. While our changes have been designed to address the scheduling issues specific to rehoming, it might be desirable to provide more general control of scheduling behaviors. Such control might allow a network operator to suspend the processing of individual threads, or queues.

A key challenge in providing such control will be to design a control interface that is safe to use, without requiring in-depth knowledge of the default scheduling strategy. While we have not yet explored the design space for scheduling control interfaces, we note that one possibility might be to limit the duration of time for which a thread can be suspended. After this time limit, the system would revert to known safe, autonomous behavior.

Processing Speed

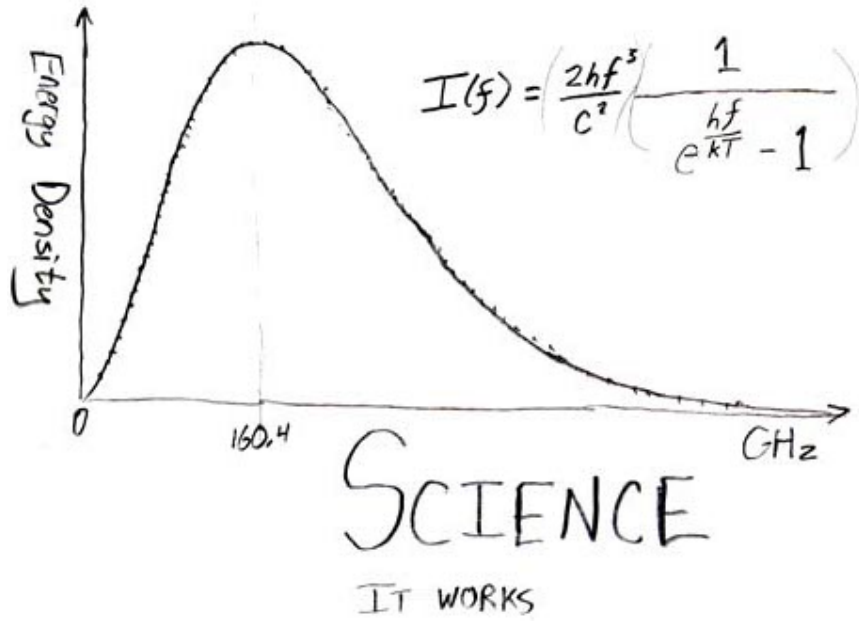
As noted in Sections 8.2.2 and 8.3.4, comparison of our empirical results with those of Wu et al. [85] suggests that there are order of magnitude differences in processing speed between different BGP implementations running on similar hardware. This leads us to wonder about the source of the performance differences, and about the fundamental limit of BGP processing speed. A deeper understanding of BGP performance could help improve existing implementations. Such improvements could yield a number of benefits, such as helping cope with the growth of the number of Internet prefixes [42, 62], eliminating the need for complicated scaling band-aids, such as route-reflectors and confederations [10, 26], or enabling radically new, and simpler, routing architectures [29, 37, 82].

8.4 Concluding Remarks

We have, in this dissertation, made significant progress towards understanding, and resolving, one of the key obstacles to a highly reliable Internet. In particular, we have shown that there exists a set of modest, software-only changes, which can reduce the down time caused by routing software upgrades from minutes, to hundreds of milliseconds. As software upgrades are one of the leading causes of outages in the Internet, our changes could significantly improve Internet reliability. Moreover, we believe our solution for mitigating the outages resulting from routing software upgrades can be extended to address other causes of router outages.

Looking beyond the issue of router outages, we observed that our empirical results, when contrasted with benchmarking of BGP performance by others, suggest that there exist order of magnitude differences in route processing speeds. This, in turn, suggests that, despite much work on BGP performance in the large, there is much yet to be understood about the fundamental limits of BGP processing speeds. We believe this is a promising area of research, which could profoundly simplify intra-AS routing. These simplifications might include eliminating the need for complicated scaling band-aids such as route reflectors and confederations, and enabling radically simpler routing architectures.

Afterword



after Randall Munroe, xkcd.com

A Supplemental System Charts

Throughout this dissertation, we have presented partial system charts. These charts focus omit some nodes, in order to focus attention on the nodes most relevant to the salient aspects of an experiment. In addition, in order to facilitate comparison between experiments, the partial system charts for pairs or groups of experiments are often set on a common time scale.

While those choices are appropriate for supporting the conclusions drawn from the experiments, they limit the reader's ability to explore a broader picture of an experiment. To remedy this deficiency, we present, herein, full system charts corresponding to all of the partial system charts provided in the body of the dissertation. The charts presented here include all nodes except the sink and customer sink. Further, each chart is set on the smallest time scale that spans the experiment duration.

Reading System Charts

A system chart provides a wealth of information, including detailed resource utilization data for each node, significant process and protocol events on each node, and measurements of end-to-end reachability. We detail the presentation of each type of information in turn.

The top five graphs present the CPU load on all of the nodes in the experiment, except the sink and the customer sink. The shaded areas indicate CPU load for the `bgpd` and `zebra` processes, broken down by user mode and system mode time, as denoted in the legend. The light dotted line indicates total CPU load. In cases where CPU load is due to some process other than `bgpd` or `zebra`, the dotted line exceeds the sum of the blue and green shaded areas.

In most cases, the timeseries chart for the customer router also present measurements of the link load, at that router, due to BGP messages.[1] These measurements are provided by the blue and green timeseries, for messages inbound and outbound to the customer router, respectively. Note however, that the link load is generally insignificant.

In addition to the illustrating the CPU load, each of the top five graphs also illustrates the timing of important events during the rehoming process. For each such event, we place an annotation on the graph for the node where the event occurred, and at the x-axis position for the time at which the event occurred. For example, **zebra** $\downarrow$, the leftmost label and marker on the graph labeled "initial router" in Figure A.1, indicates that the experiment framework signalled the `zebra` process to terminate one second after the start of the experiment.[2] We describe the complete set of events illustrated in our system charts, including how they are captured, and the precision of their timestamps, in Table A.1.

Finally, the bottommost graph indicates the reachability of different IP address prefixes, as measured from two vantage points. Green areas denote times during which packets are delivered successfully, and red areas denote times during which packets are lost. For example, in Figure A.1, the large red region in the series labeled CUST $\rightarrow$ INETN, indicates that from time 14 seconds to time 107 seconds, packets sent from the customer sink towards INETN were not received at "sink".

Note that the scale of the x-axis often differs between charts. Accordingly, the time at the end of the outage is emphasized with larger text.

[1] The exception is those experiments involving customers with static routing.

[2] The ordering of closely spaced events can be determined by their vertical positioning. The later event will appear above the earlier event, except in the case where the earlier event occupies the topmost position in the graph. In such cases, the later event occupies the bottommost position.

	description	capture method	precision
bgpd ↻	the framework restarted the `bgpd` process	experiment logfile	nanosecond
bgp $\downarrow_c$	the BGP peering with the customer went down (also initial router, remote router, and ISP router.)	`bgpd` log file	microsecond
bgp $\uparrow_c$	the BGP peering with the customer came up (also initial router, remote router, and ISP router.)	`bgpd` log file	microsecond
∼CUST1	the nexthop IP address of the kernel FIB entry for CUST1 has changed (also CUSTN, INET1, and INETN)	polling kernel FIB (once per second)	nanosecond
−CUST1	the kernel FIB entry for CUST1 has been removed (also CUSTN, INET1, and INETN)	polling kernel FIB (once per second)	nanosecond
+CUST1	a kernel FIB entry for CUST1 has been added (also CUSTN, INET1, and INETN)	polling kernel FIB (once per second)	nanosecond
+CUST1$_{\triangleleft C}$	a BGP advertisement for CUST1 was received from the customer router (with variations for different prefixes and BGP peers)	packet capture	microsecond
+CUST1$_{\triangleright ISP}$	a BGP advertisement for CUST1 was sent to the ISP router (with variations for different prefixes and BGP peers)	packet capture	microsecond
-CUST1$_{\triangleleft I}$	a BGP withdrawal for CUST1 was received from the initial router (also CUSTN)	packet capture	microsecond
-CUST1$_{\triangleright R}$	a BGP withdrawal for CUST1 was sent to the remote router (also CUSTN)	packet capture	microsecond
disable nic	the framework disabled the customer-facing network card	experiment logfile	nanosecond
enable nic	the framework enabled the customer-facing network card	experiment logfile	nanosecond
enable peering	the framework enabled a BGP peering between the ISP and the customer	experiment logfile	nanosecond
move link	the framework reconfigured the layer-2 link between the ISP and the customer	experiment logfile	nanosecond
open$_{\triangleright C}$	a BGP OPEN message was received from the customer router	packet capture	microsecond
reject peer	the customer router rejected a peering request from the ISP	`bgpd` log file	microsecond
zebra ↓	the framework signalled the `zebra` process to terminate	experiment logfile	nanosecond
zebra ↑	the framework started a new `zebra` process	experiment logfile	nanosecond

Table A.1: Key to events depicted on system charts.

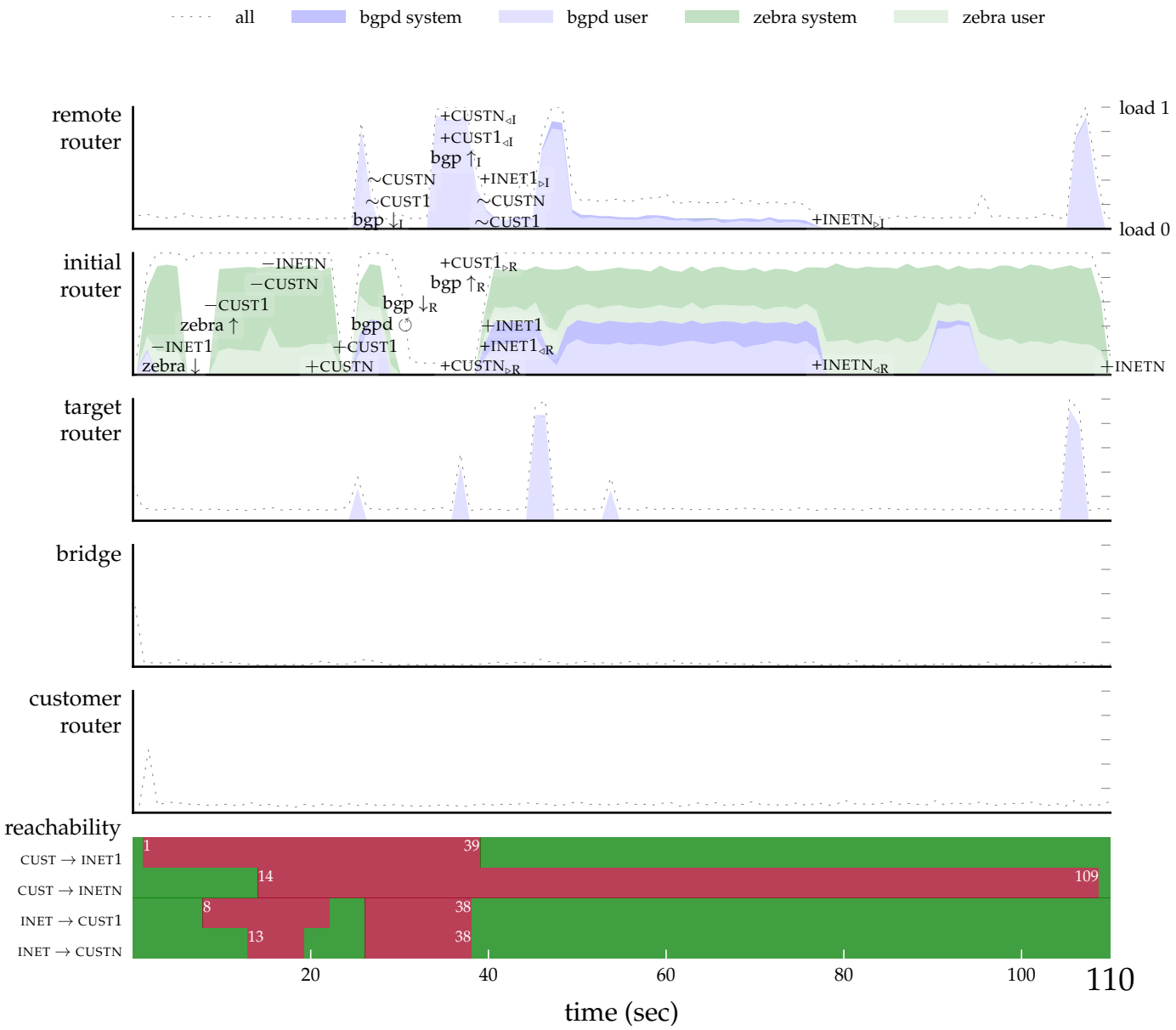


Figure A.1: Full system chart for router restart with a single statically routed customer. This chart illustrates the trial with the shortest overall outage time.

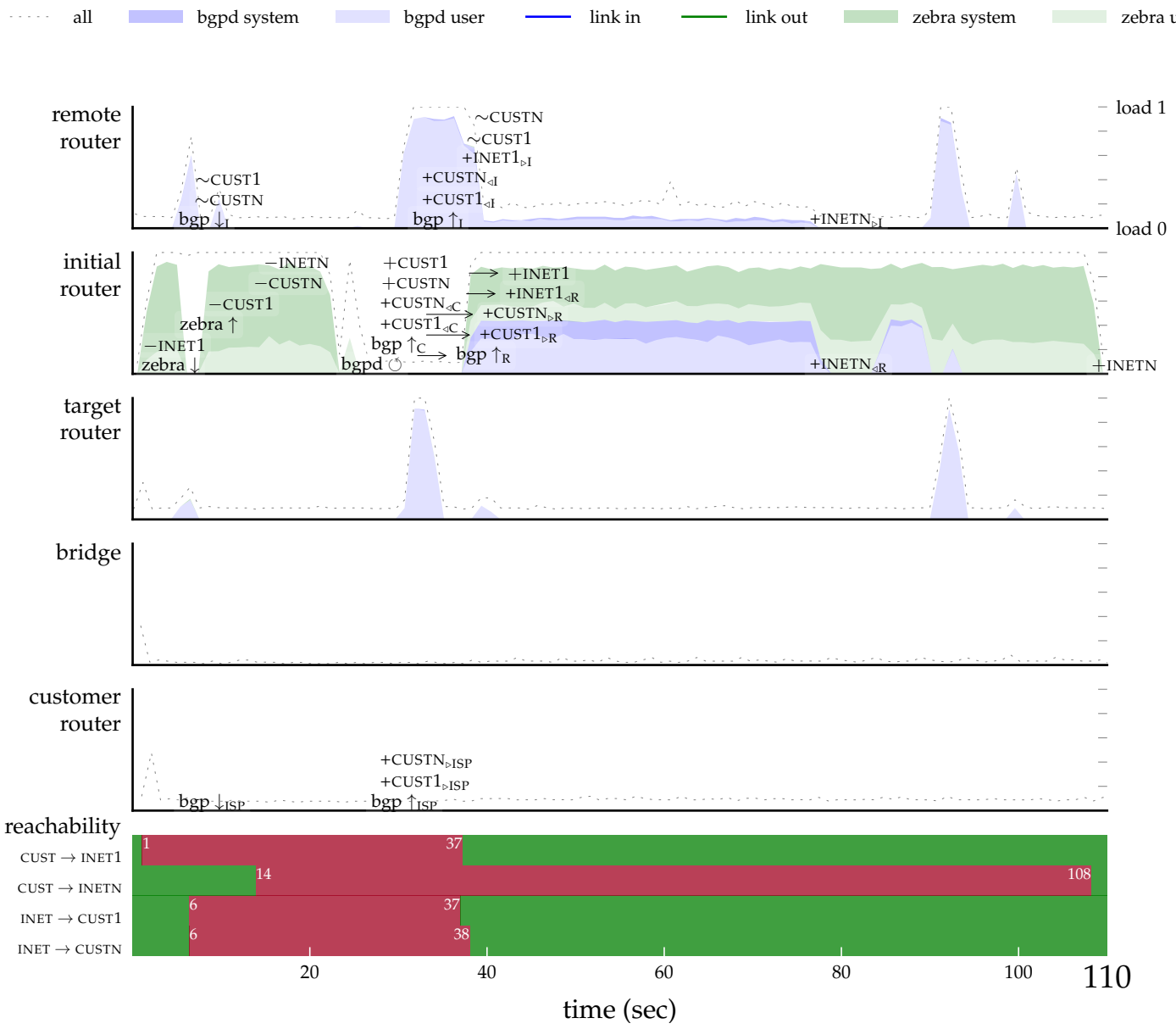


Figure A.2: Full system chart for router restart with a single BGP customer, using default routing for outbound traffic. This chart illustrates the trial with the shortest overall outage time.

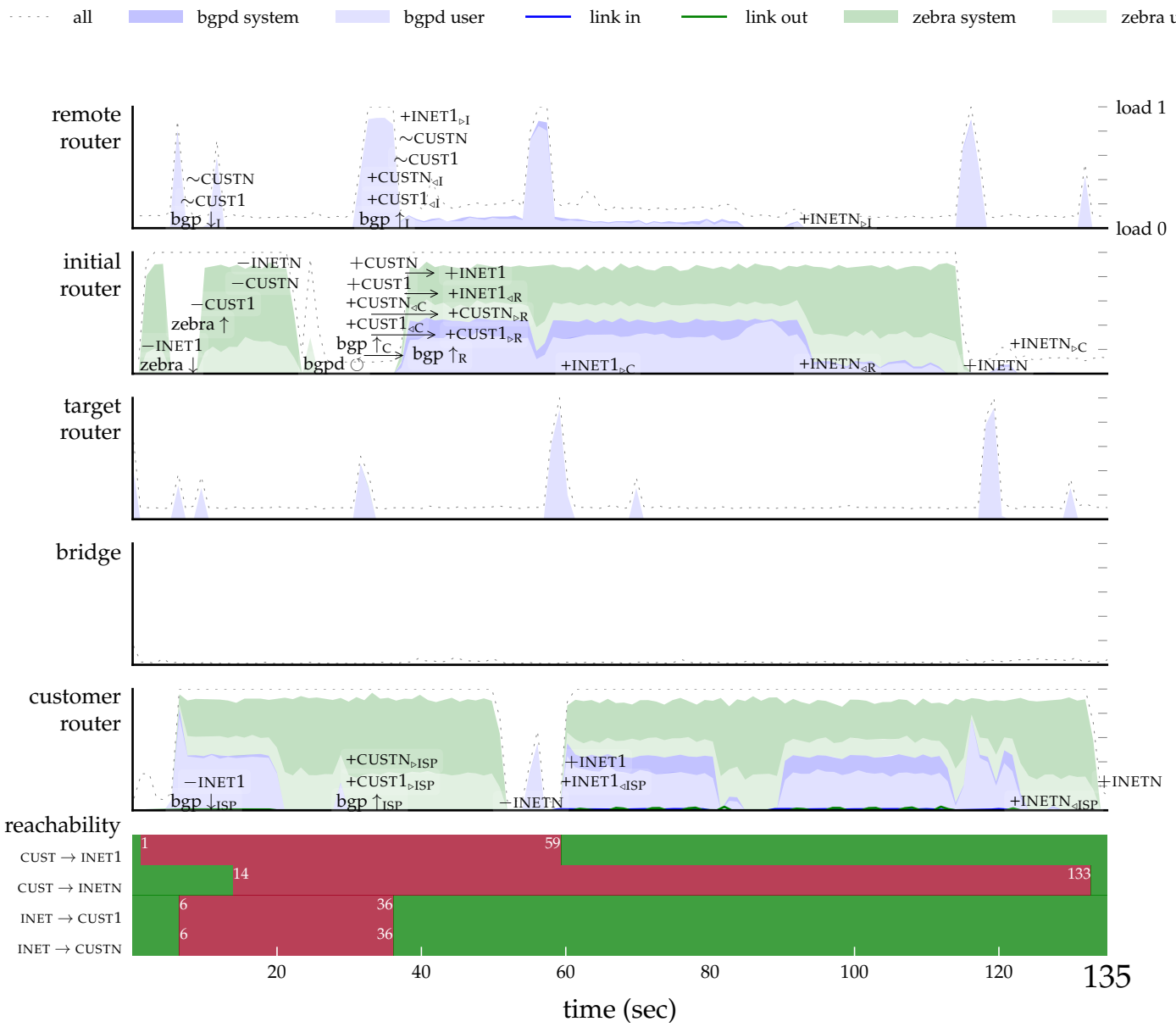


Figure A.3: Full system chart for router restart with a single BGP customer, using dynamic routing for outbound traffic. This chart illustrates the trial with the shortest overall outage time.

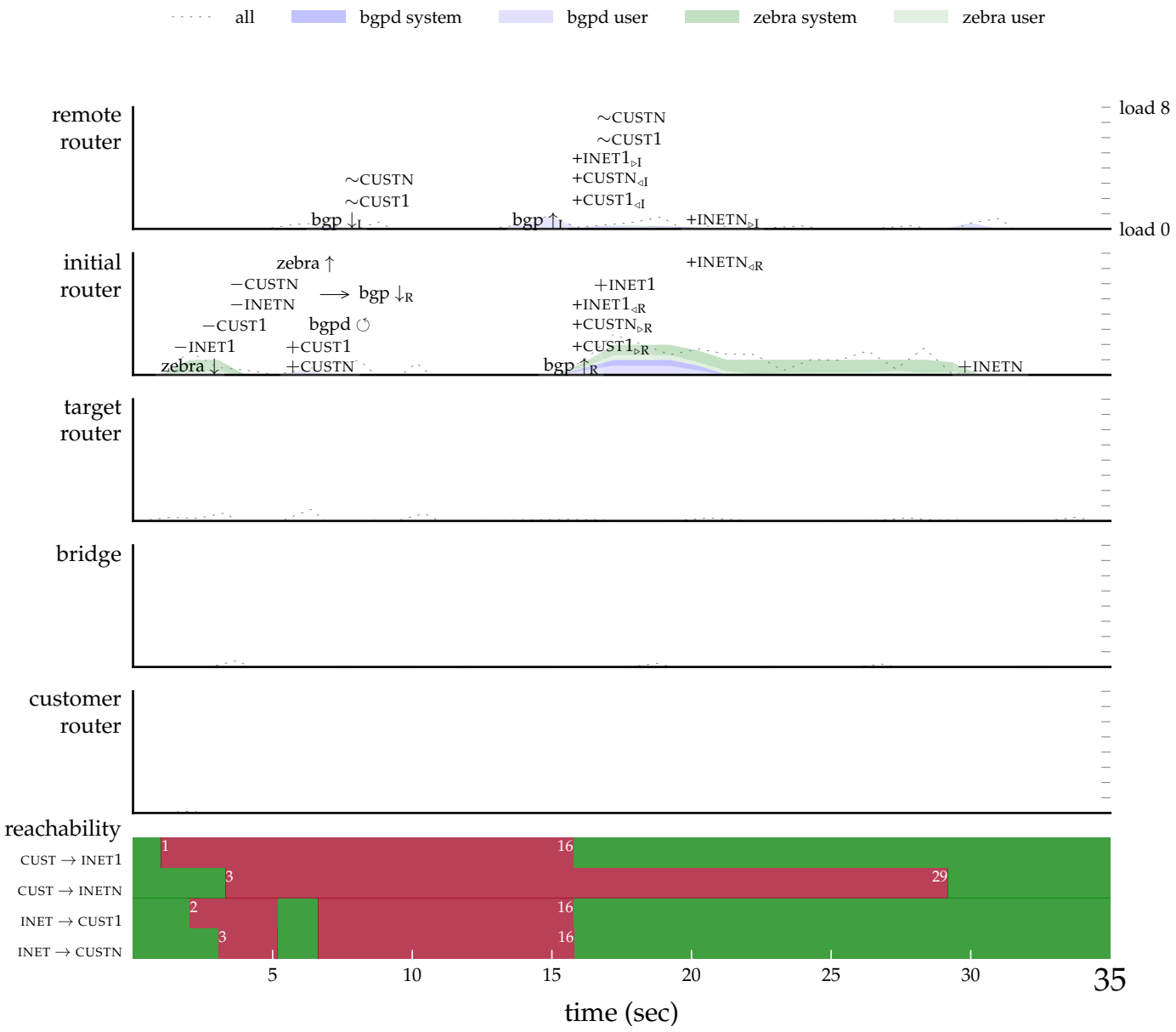


Figure A.4: Full system chart for router restart with a single static customer, on High Spec hardware. This chart illustrates the trial with the shortest overall outage time.

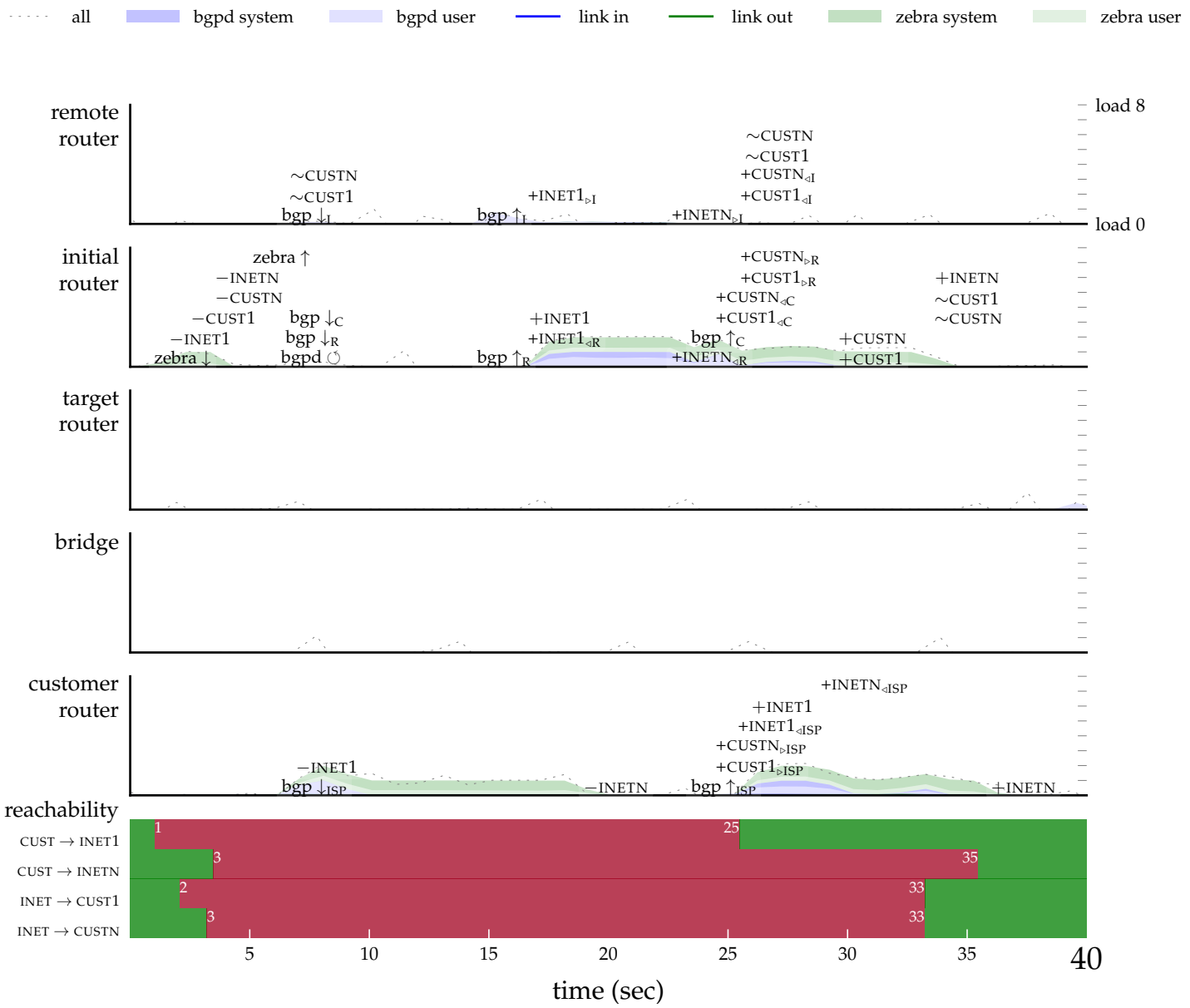


Figure A.5: Full system chart for router restart with a single BGP customer, using dynamic routing for outbound traffic, on High Spec hardware. This chart illustrates the trial with the shortest overall outage time.

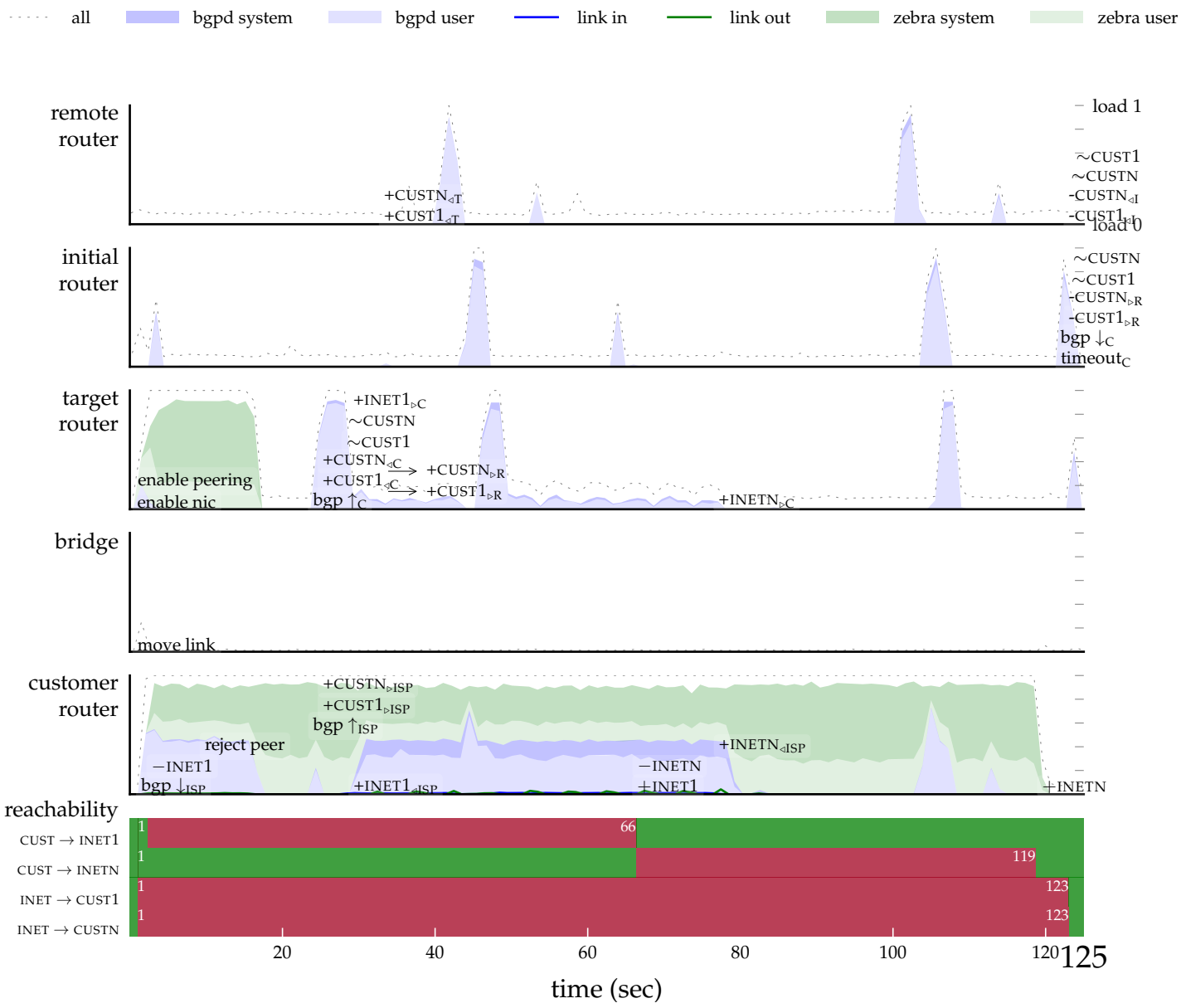


Figure A.6: Full system chart for naïve rehoming, for the trial with the minimum overall outage time.

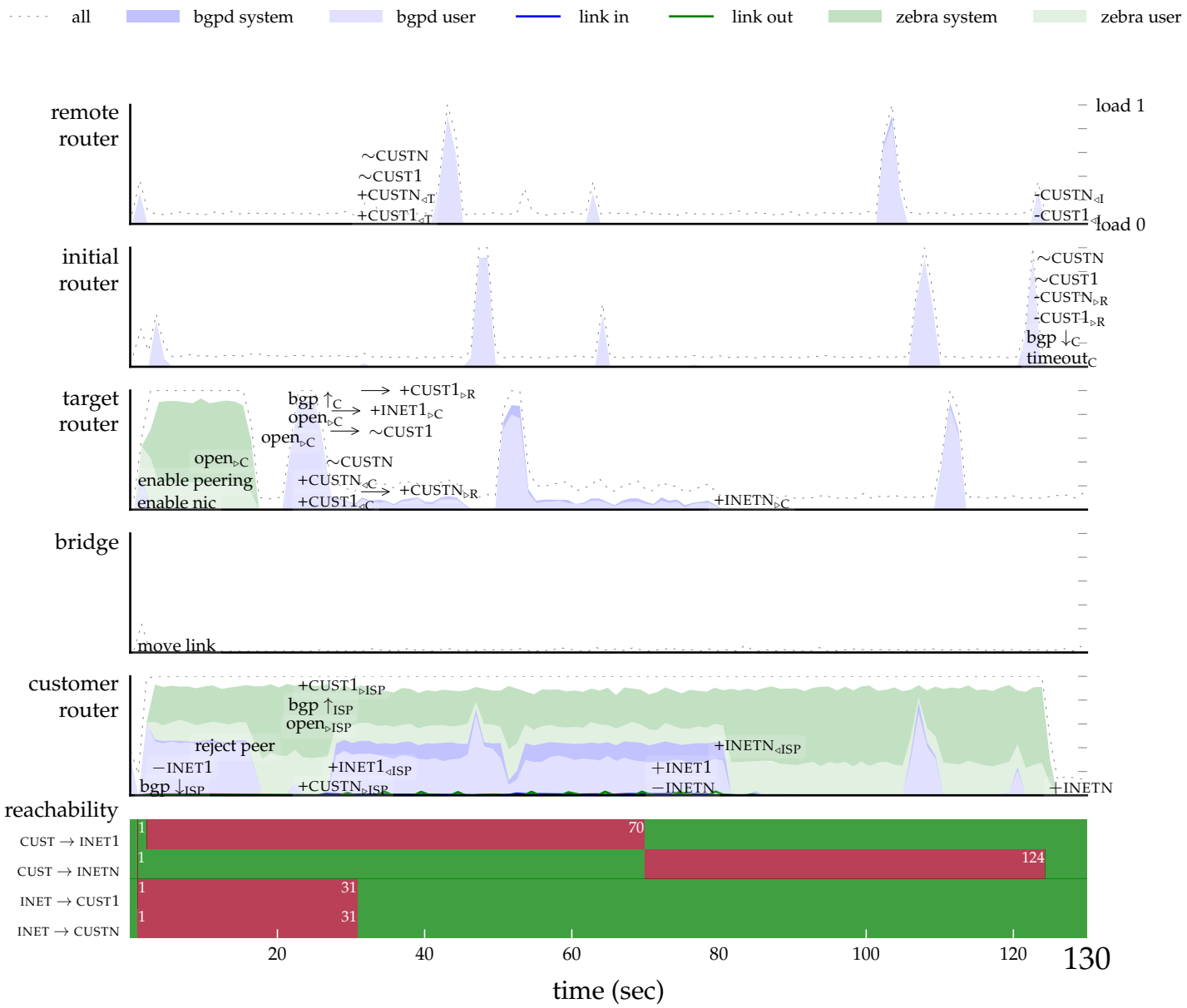


Figure A.7: Full system chart for naïve rehoming, with the initial router configured to have a higher router-id than the target router. This chart illustrates the trial with the shortest overall outage time.

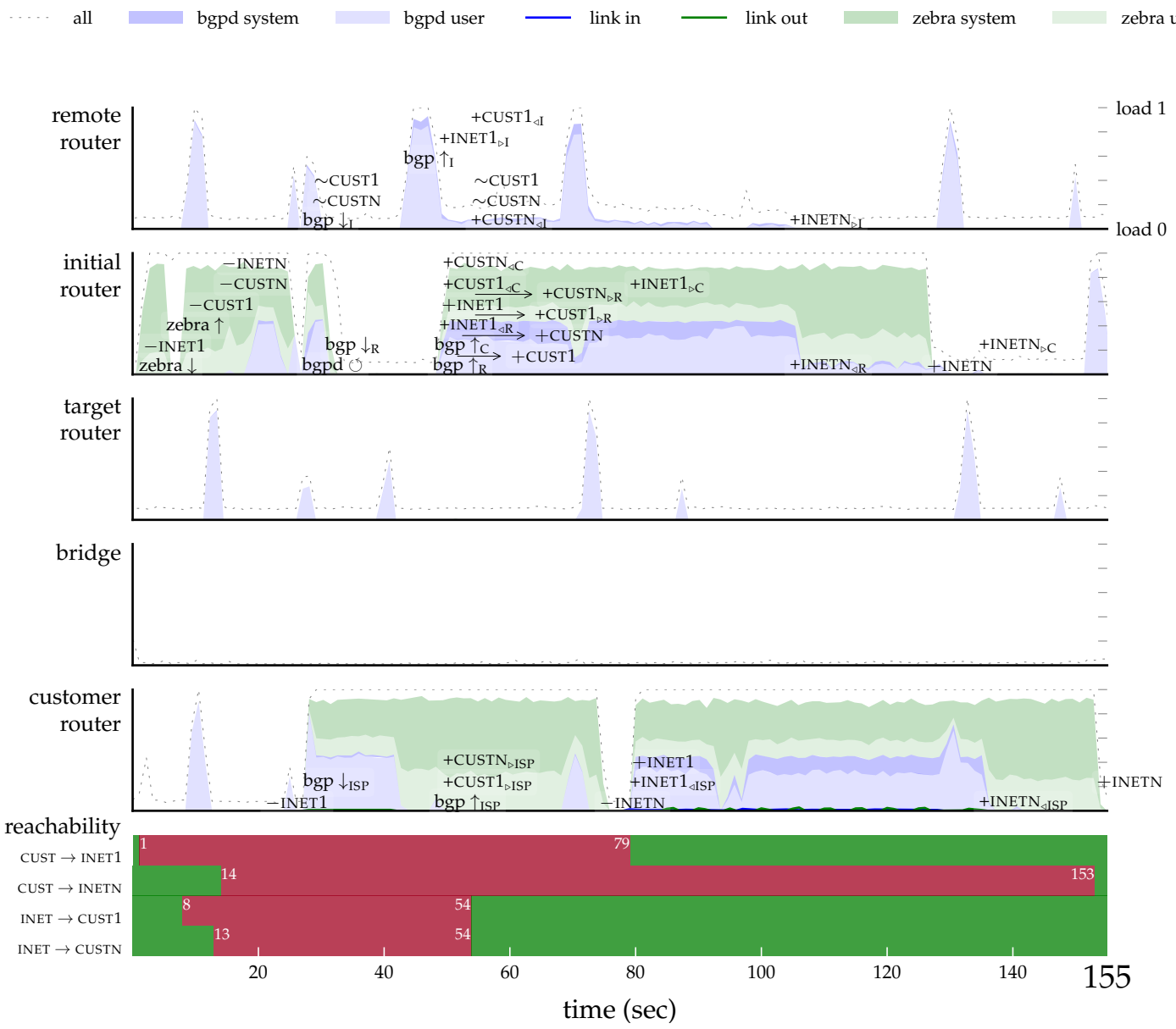


Figure A.8: Full system chart for router restart, with a single BGP customer, using dynamic routing for outbound traffic, for the trial with the longest overall outage time.

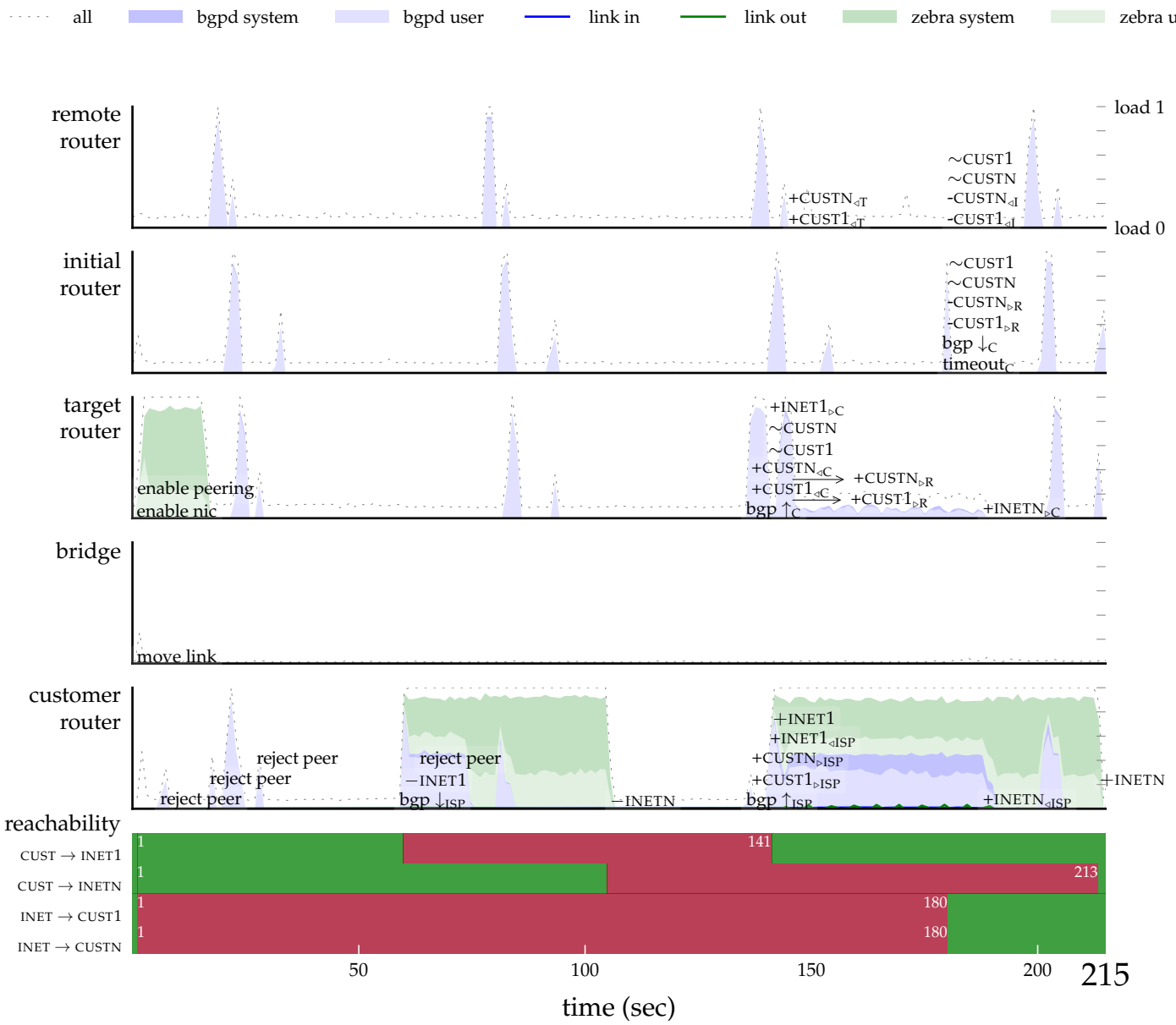


Figure A.9: Full system chart for naïve rehoming, for the trial with the longest overall outage time.

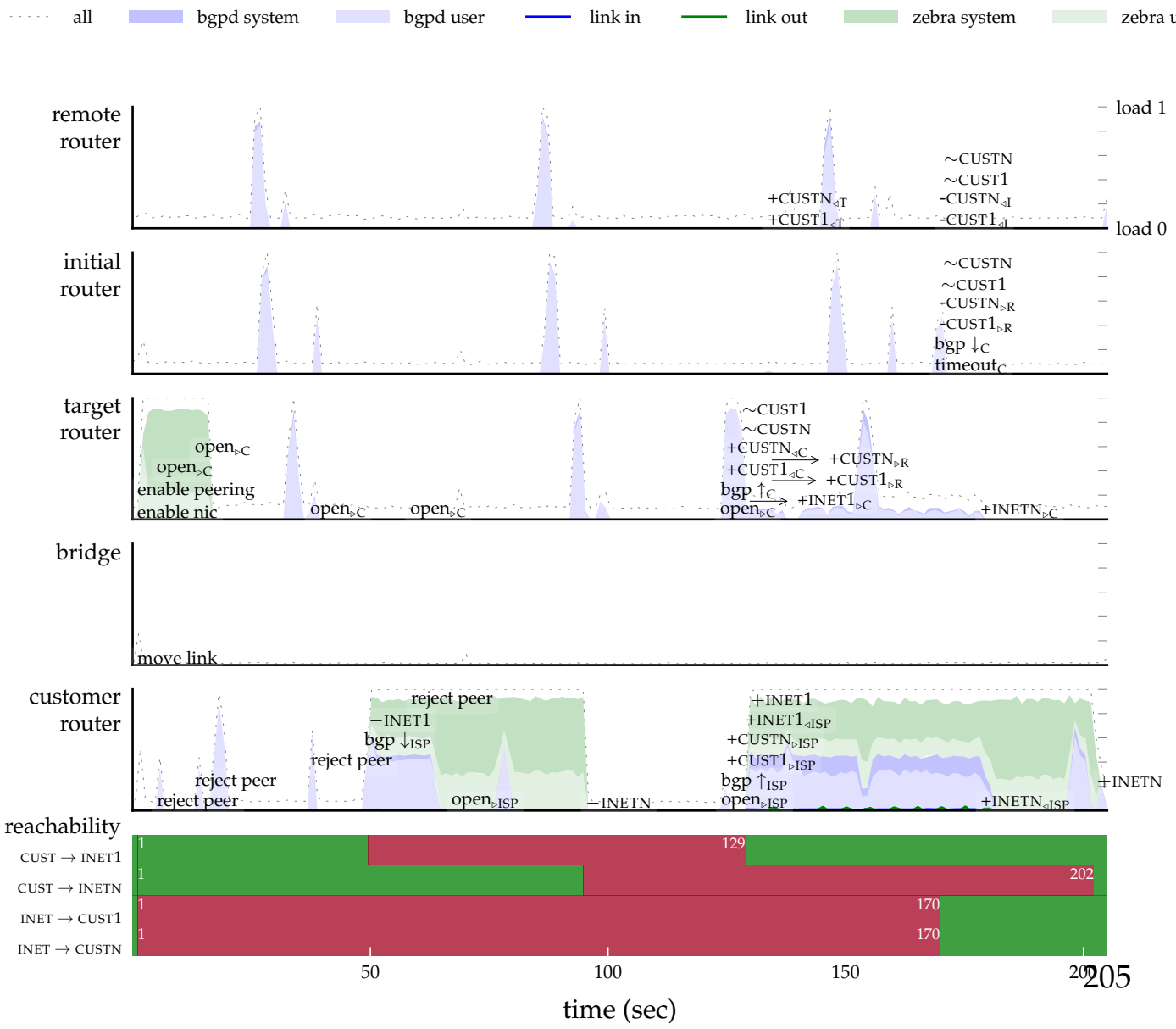


Figure A.10: Full system chart for naïve rehoming with router-id spoofing, for the trial with the longest overall outage time.

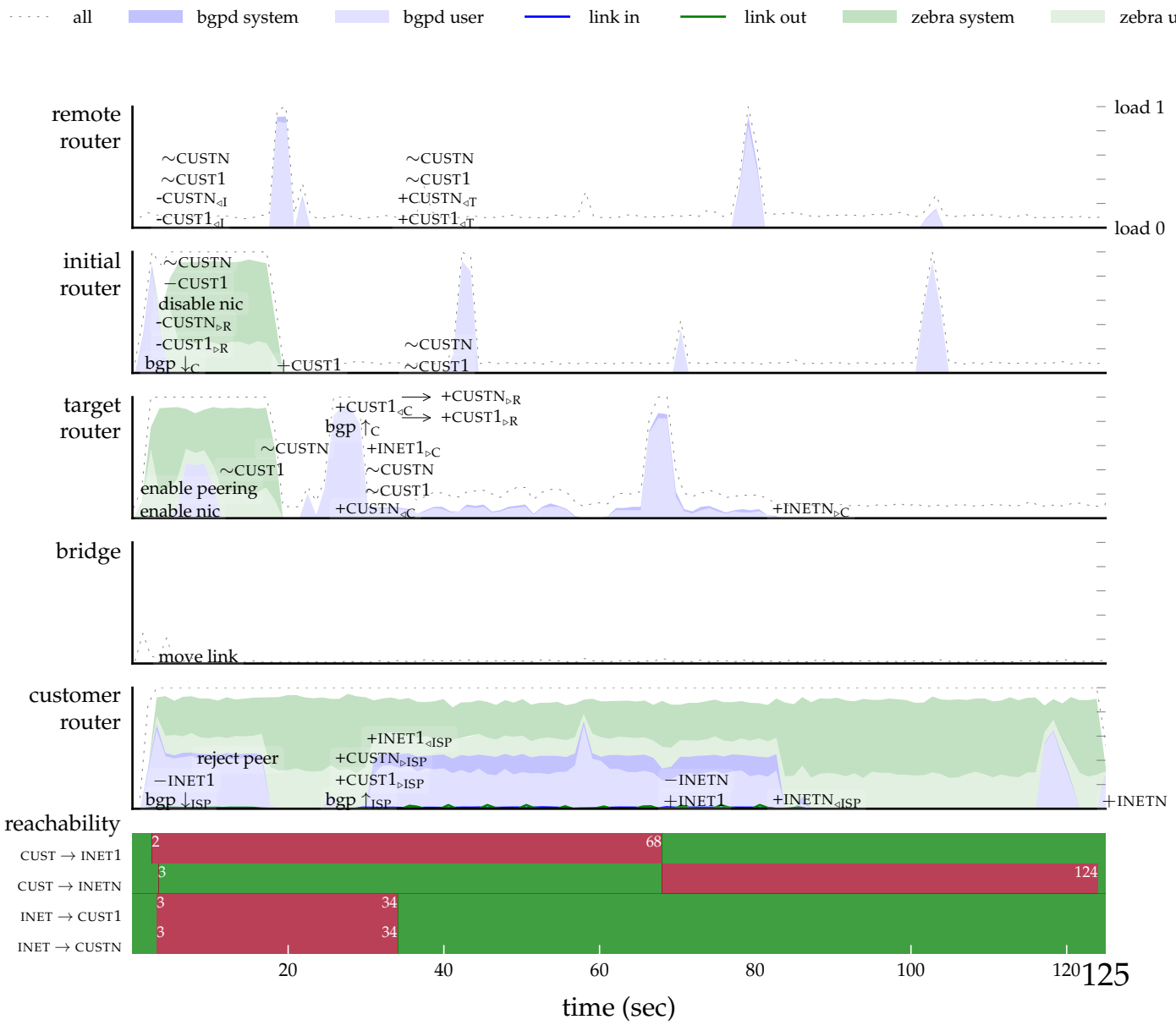


Figure A.11: Full system chart for clean shutdown rehoming, for the trial with the longest outage time.

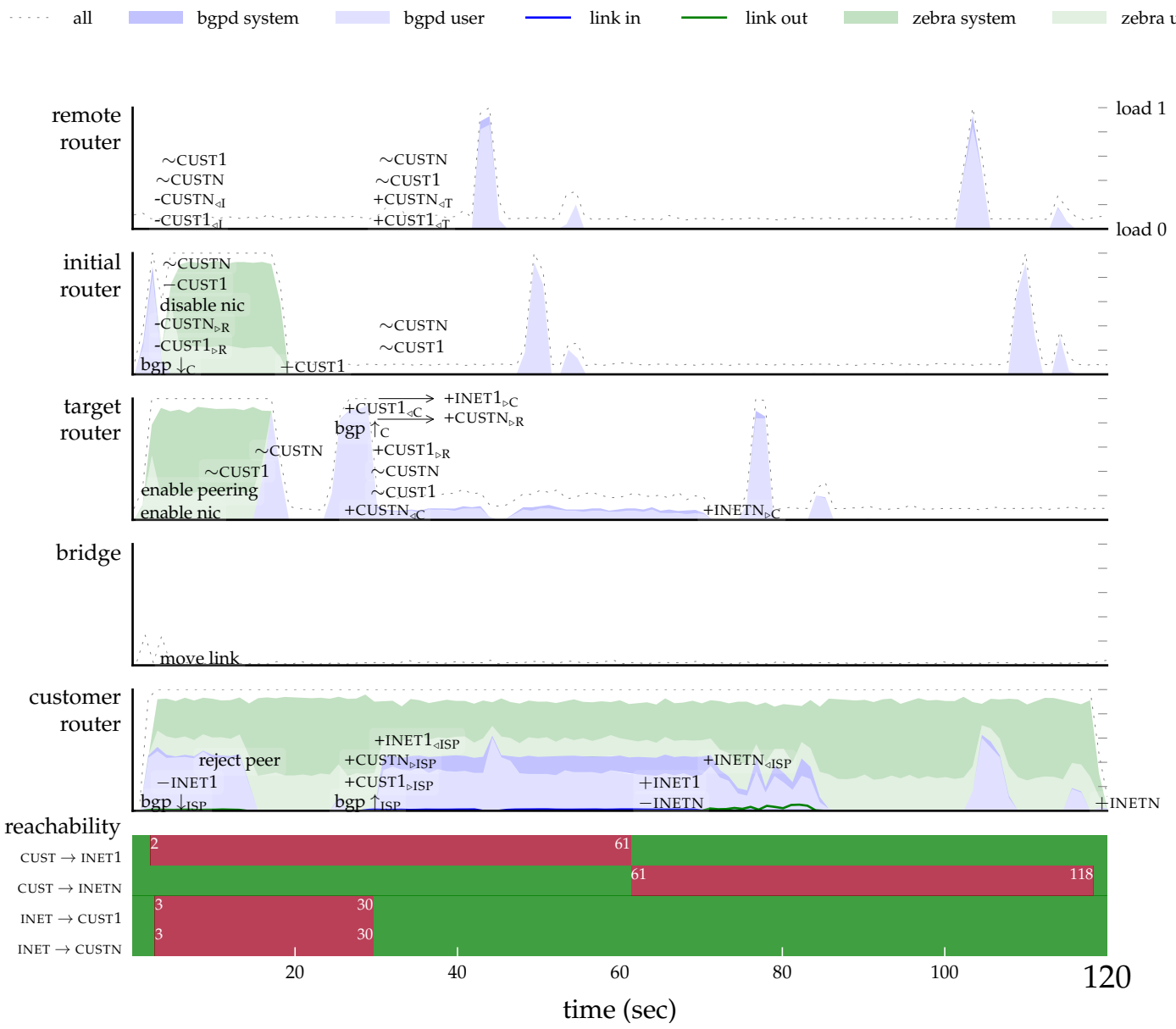


Figure A.12: Full system chart for rehoming with Graceful Restart and clean shutdown, for the trial with the minimal outage time.

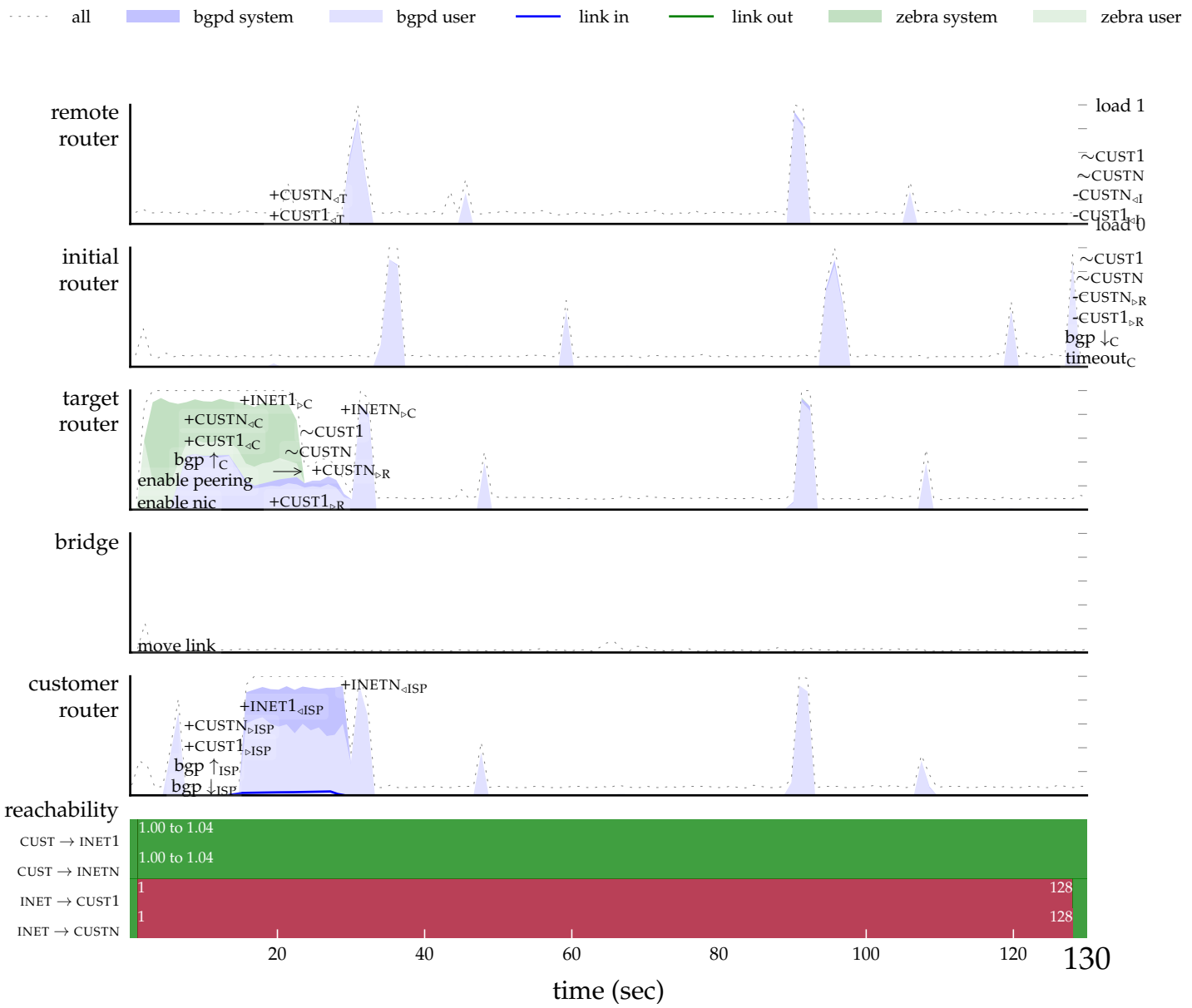


Figure A.13: Full system chart for naïve rehoming with Graceful Restart. This chart illustrates the trial with the shortest overall outage time.

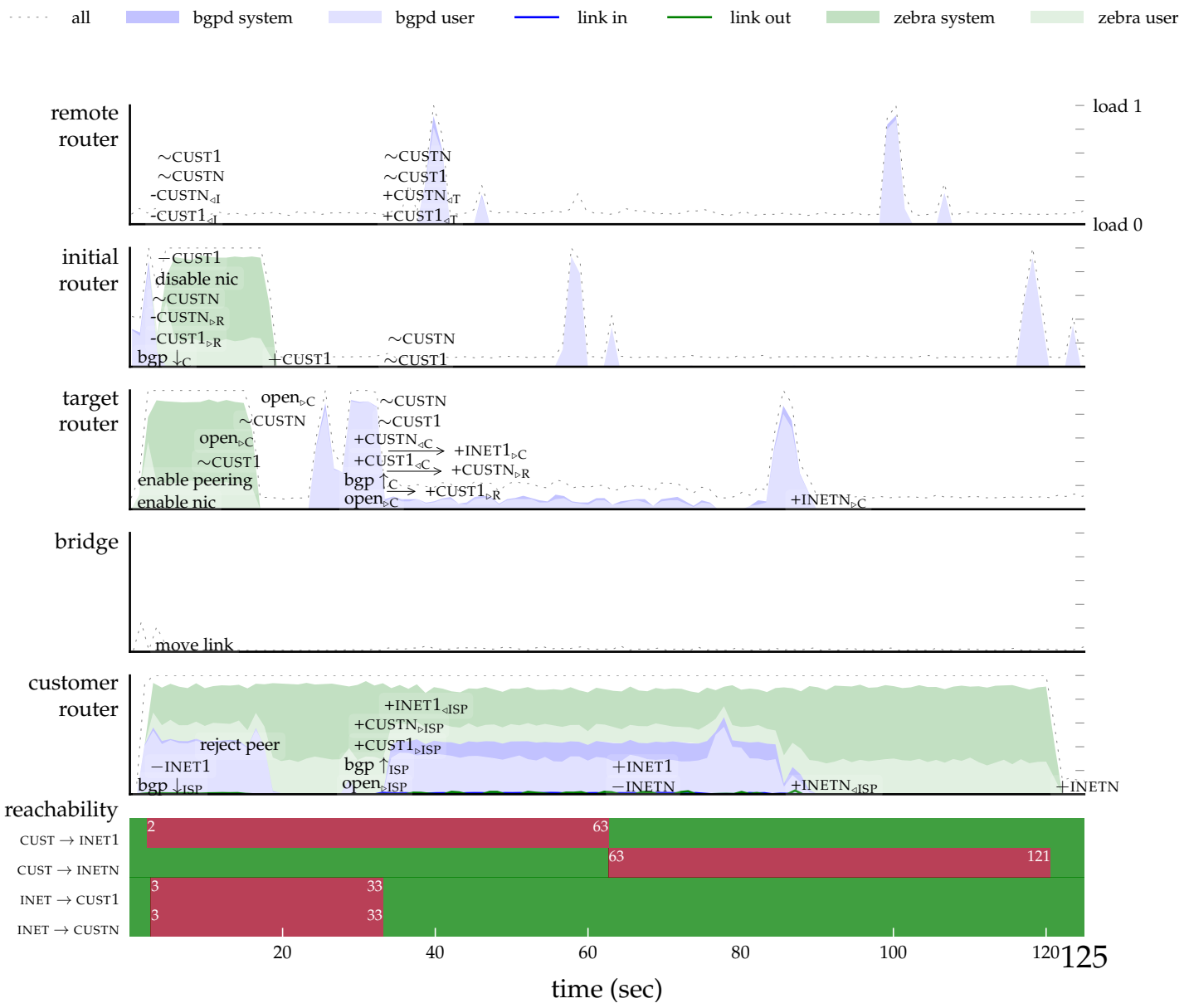


Figure A.14: Full system chart for clean shutdown with router-id spoofing. This chart illustrates the trial with the shortest overall outage time.

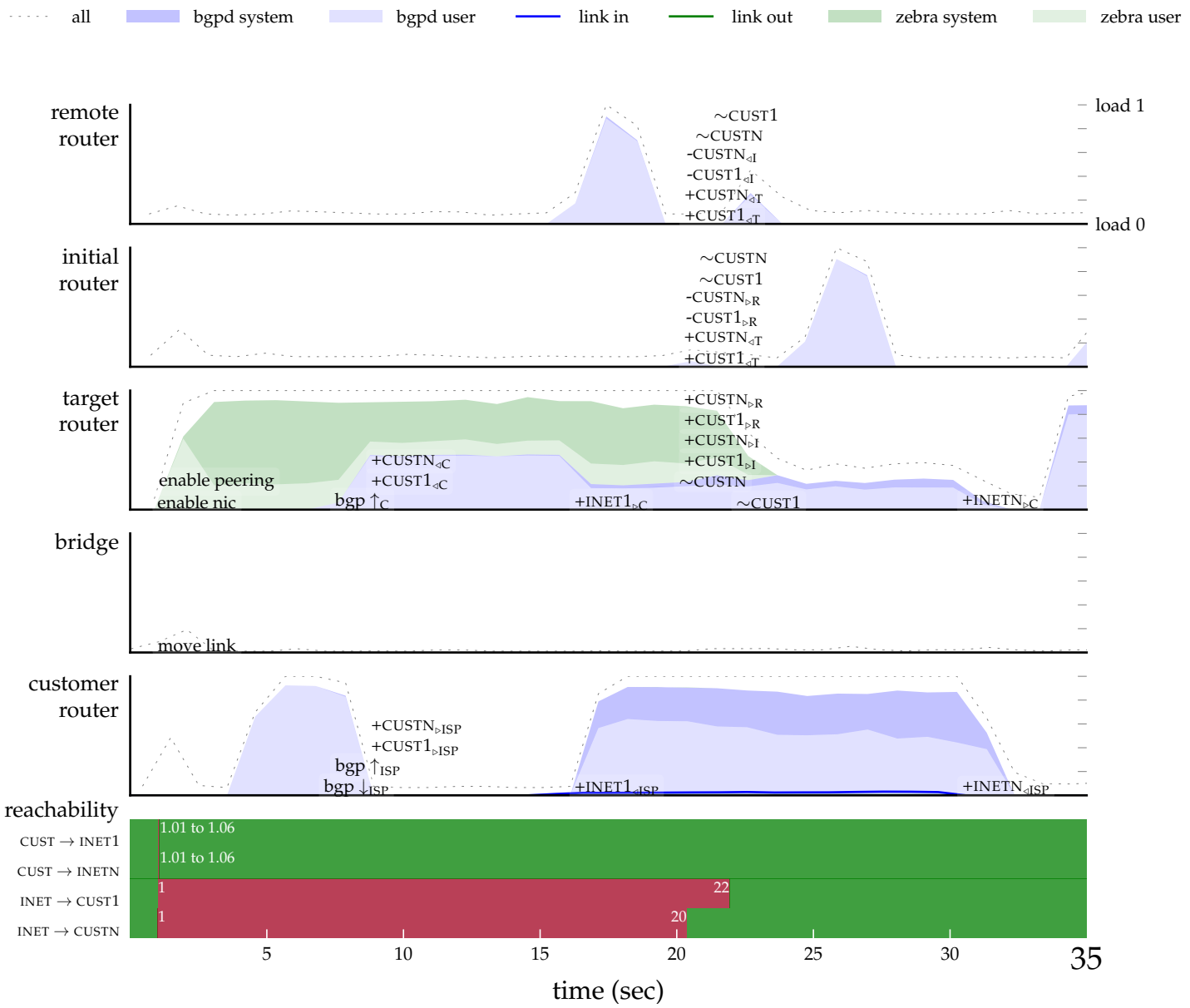


Figure A.15: Full system chart for naïve rehoming with router-id spoofing, Graceful Restart and LOCAL_PREF, for the trial with the minimal overall outage time.

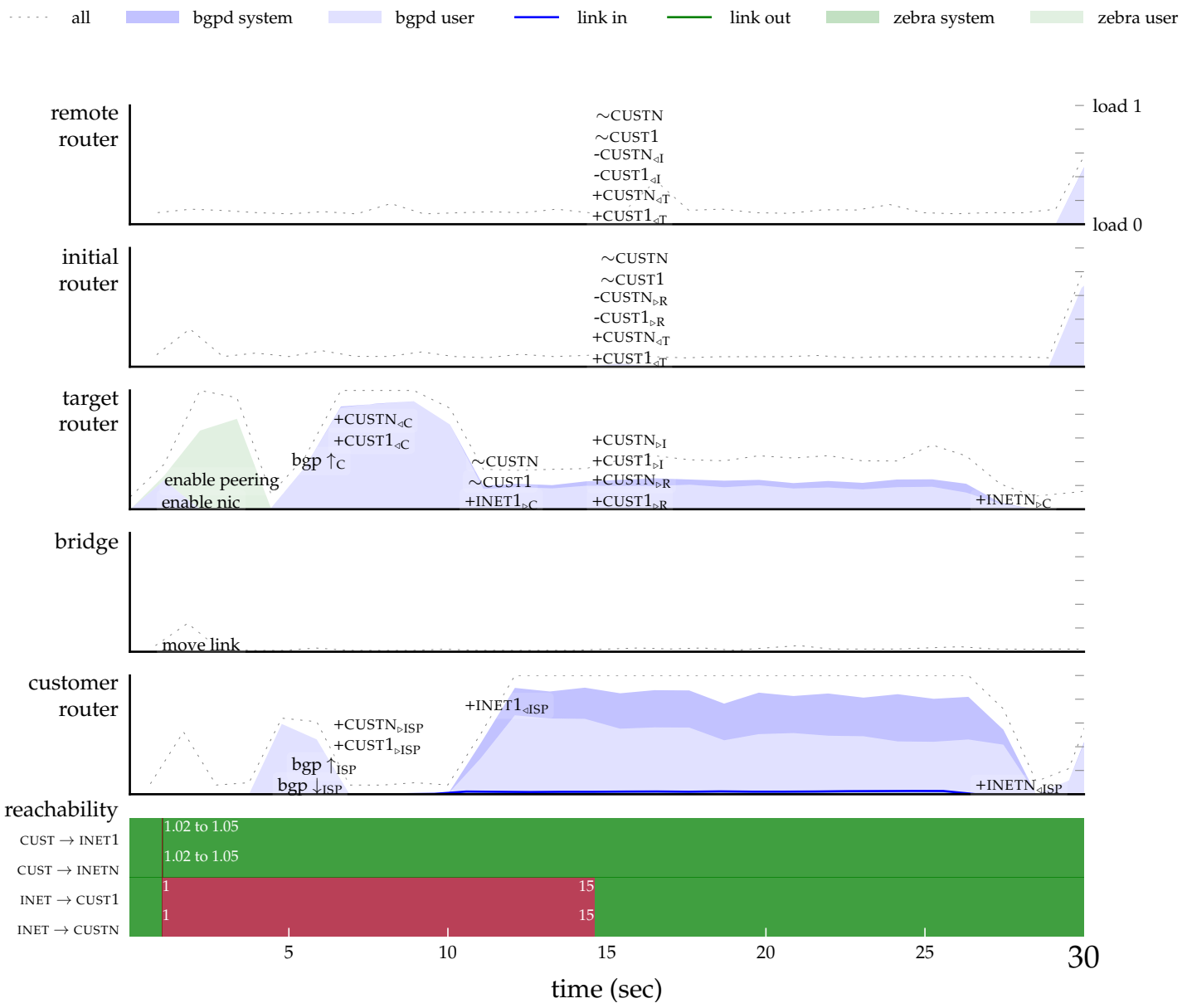


Figure A.16: Full system chart for naïve rehoming with router-id spoofing, Graceful Restart, LOCAL_PREF, and the scheduling patch of Listing 5.11. This chart illustrates the trial with the minimal overall outage time.

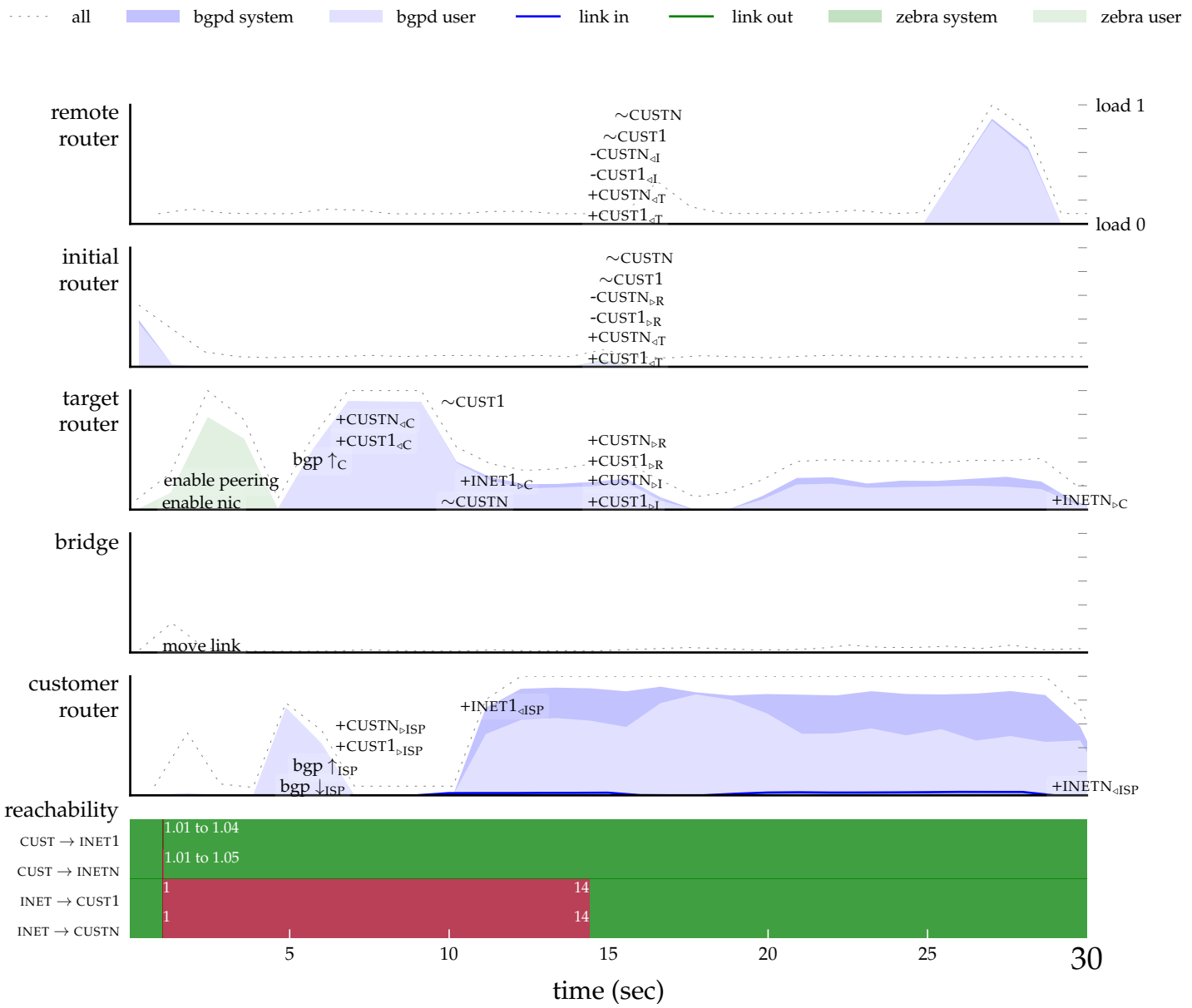


Figure A.17: Full system chart for rehoming with default hash table sizing in bgpd. This chart illustrates the trial with the minimal overall outage time.

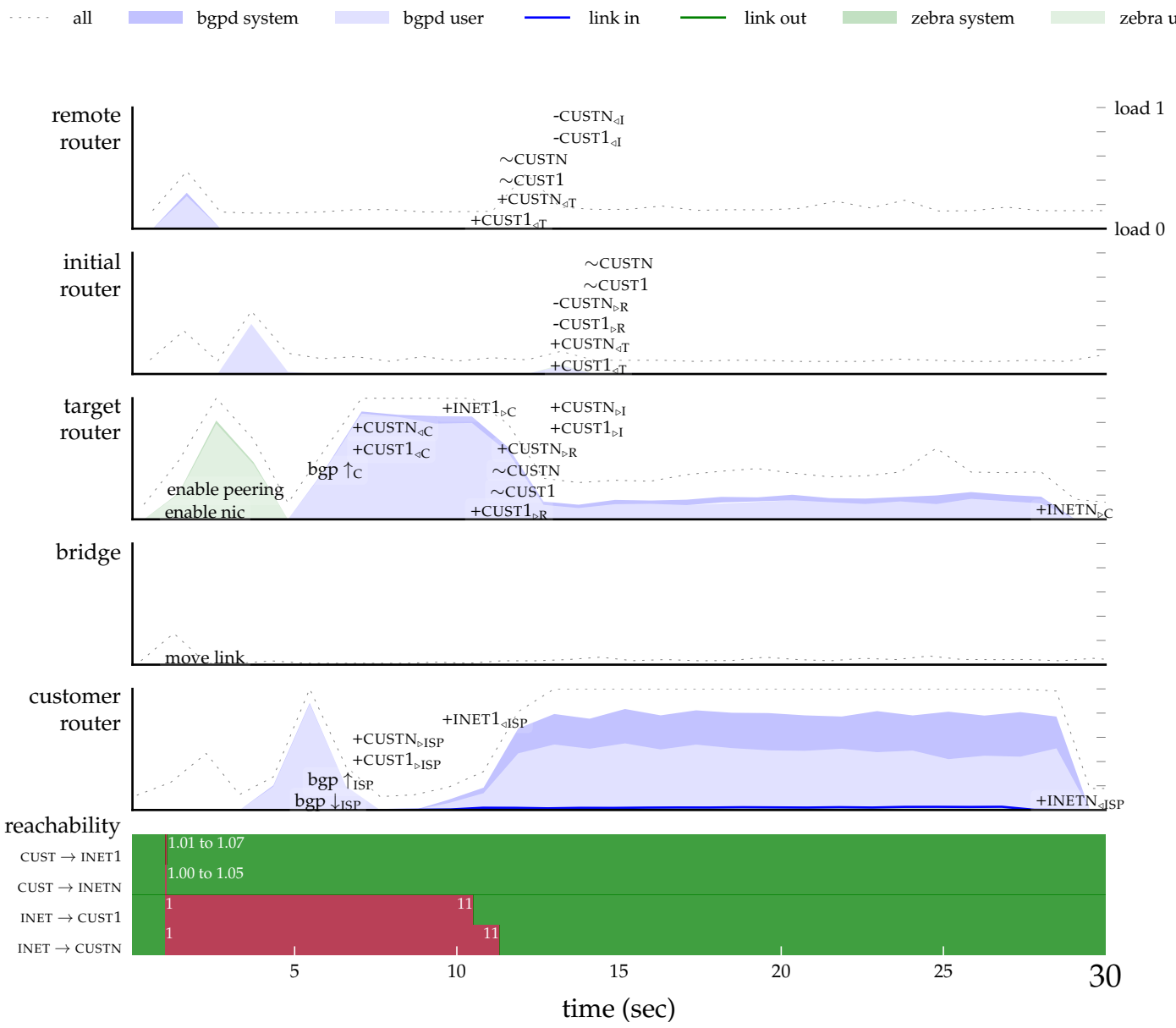


Figure A.18: Full system chart for rehoming with increased hash table sizing in bgpd. This chart illustrates the trial with the minimal overall outage time.

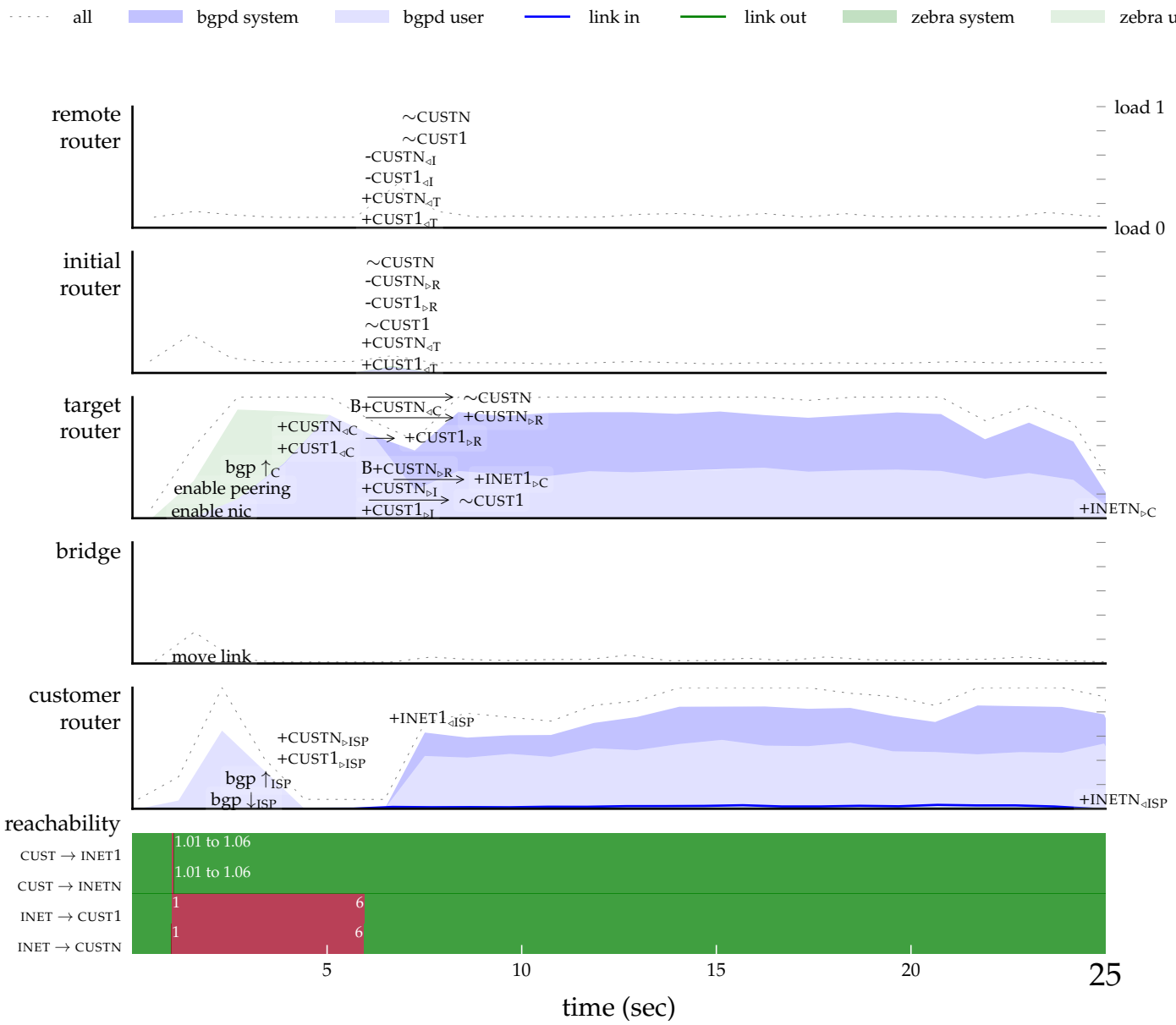


Figure A.19: Full system chart for rehoming following the patch of Listing 6.8, our first patch to reduce route processing delay. Note that debugging is enabled for bgpd on the target router. The chart illustrates the trial with the minimal overall outage time.

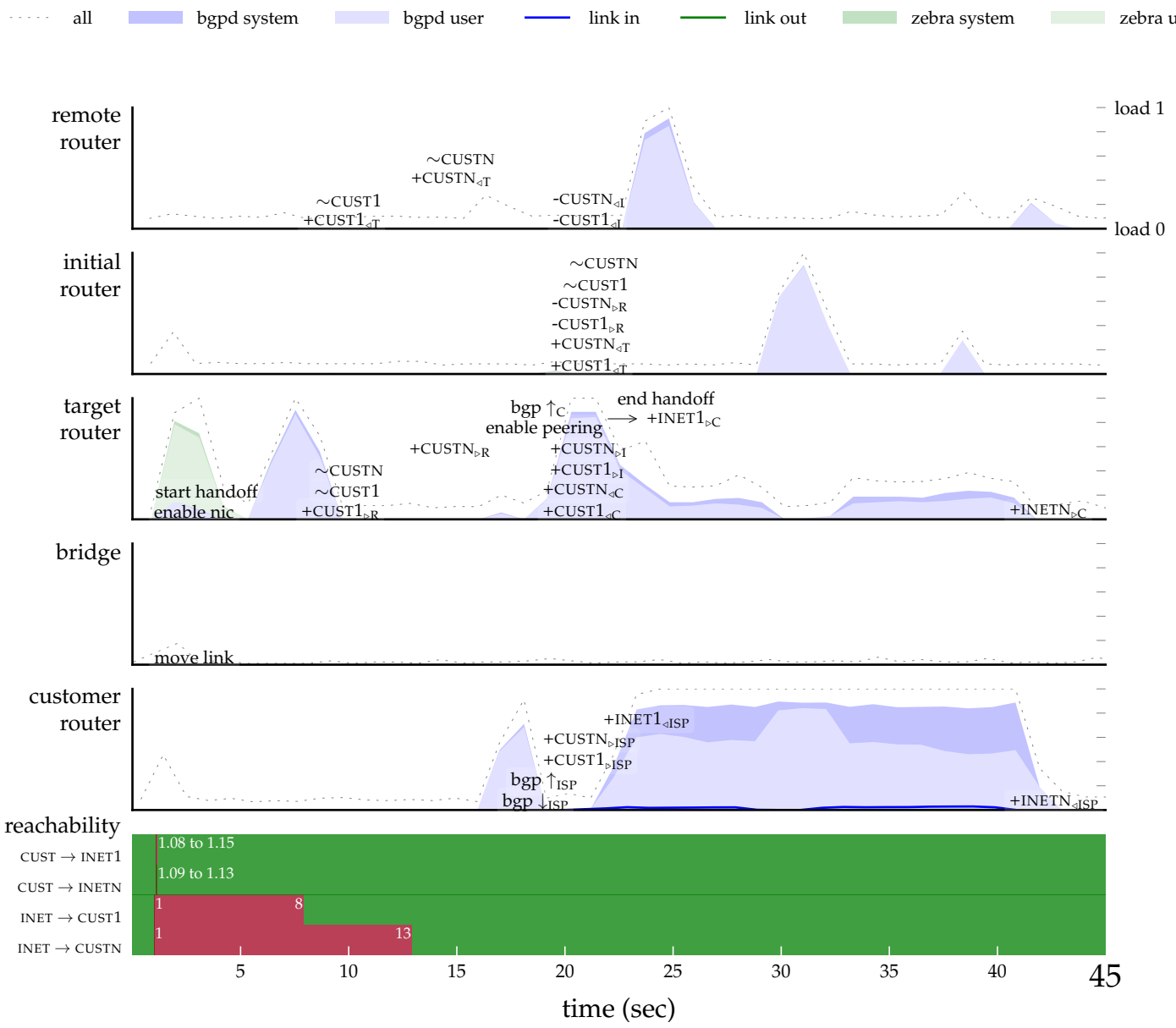


Figure A.20: Full system chart for rehoming with our initial soft handoff rehoming procedure, of Figure 7.2. This chart illustrates the trial with the minimal overall outage time.

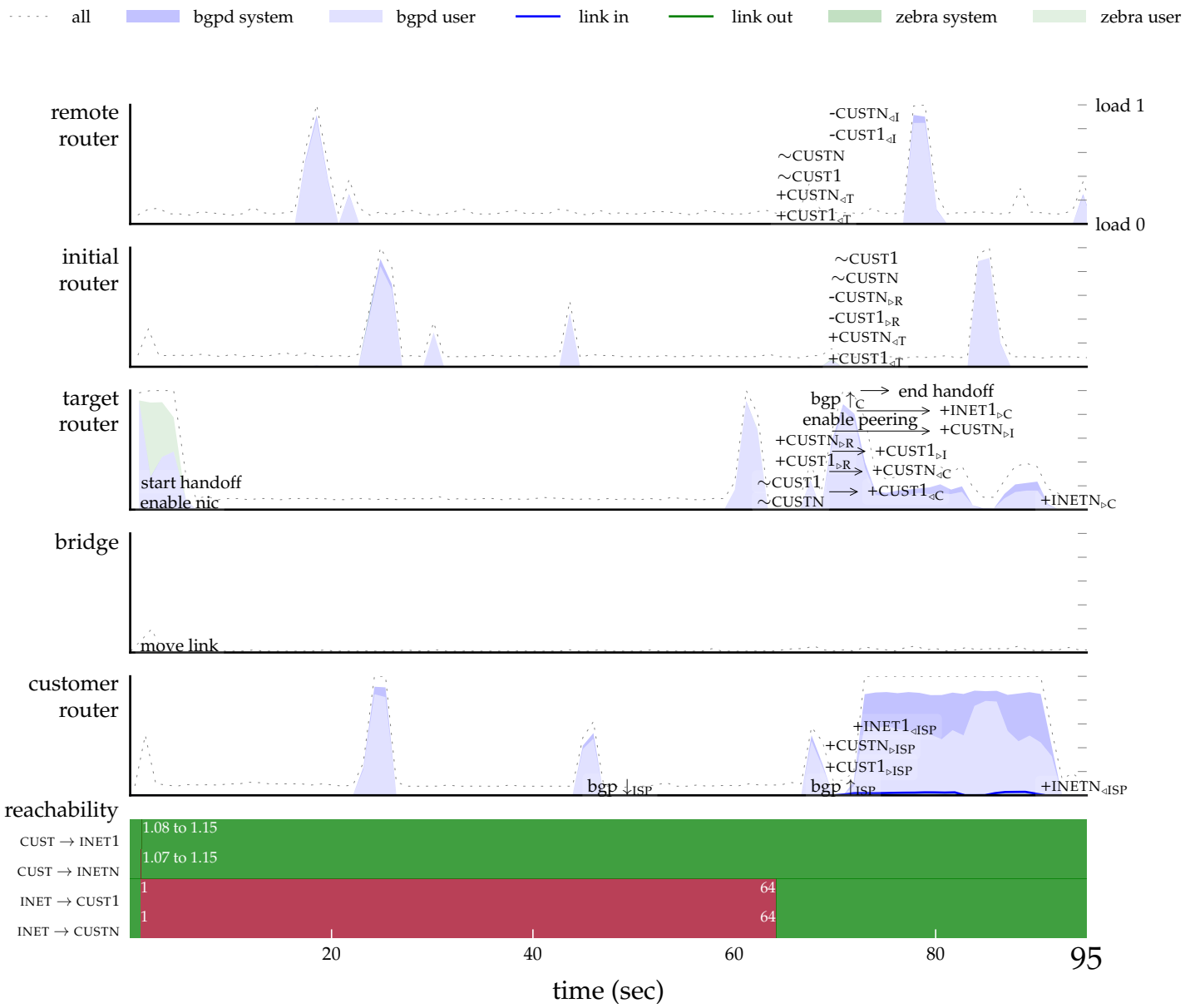


Figure A.21: Full system chart for rehoming with our initial soft handoff rehoming procedure, of Figure 7.2. This chart illustrates the trial with the maximal overall outage time.

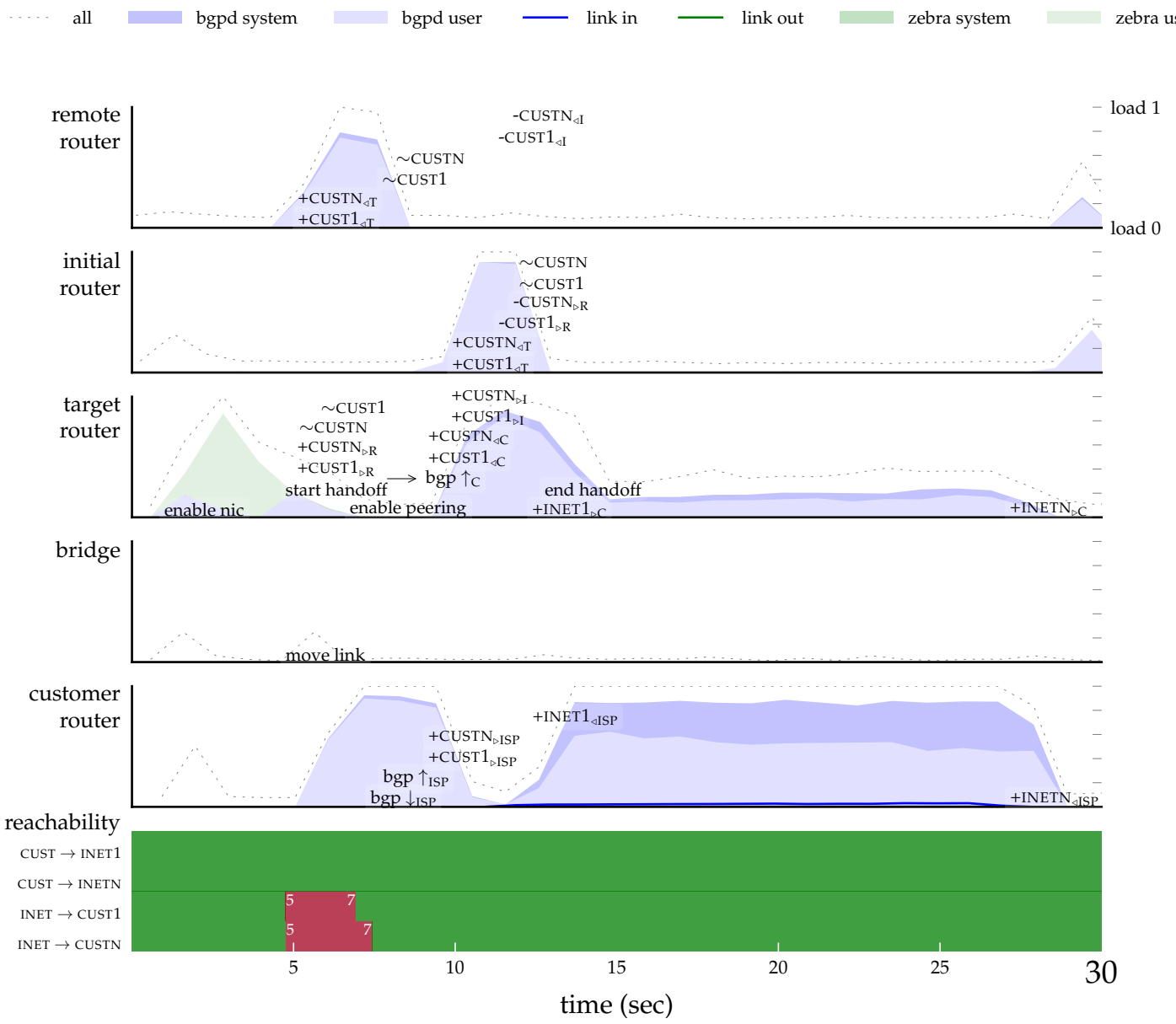


Figure A.22: Full system chart for ZIRO, with our changes to bgpd scheduling policies. This chart illustrates the trial with the maximal overall outage time.

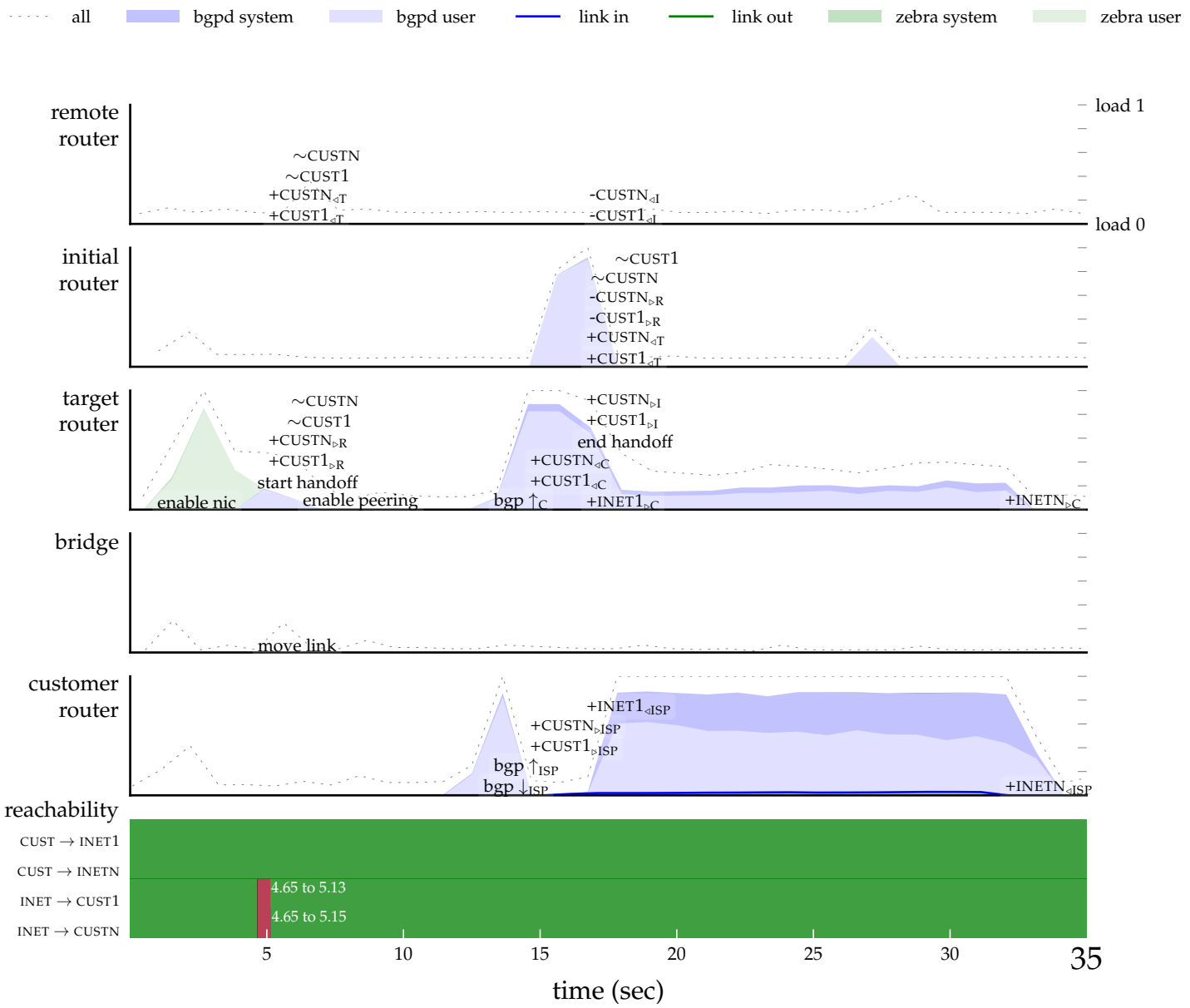


Figure A.23: Full system chart for ZIRO, following the removal of our changes to `bgpd` scheduling policies, but with our CPU optimizations for `zebra` and `bgpd` in place. This chart illustrates the trial with the maximal overall outage time.

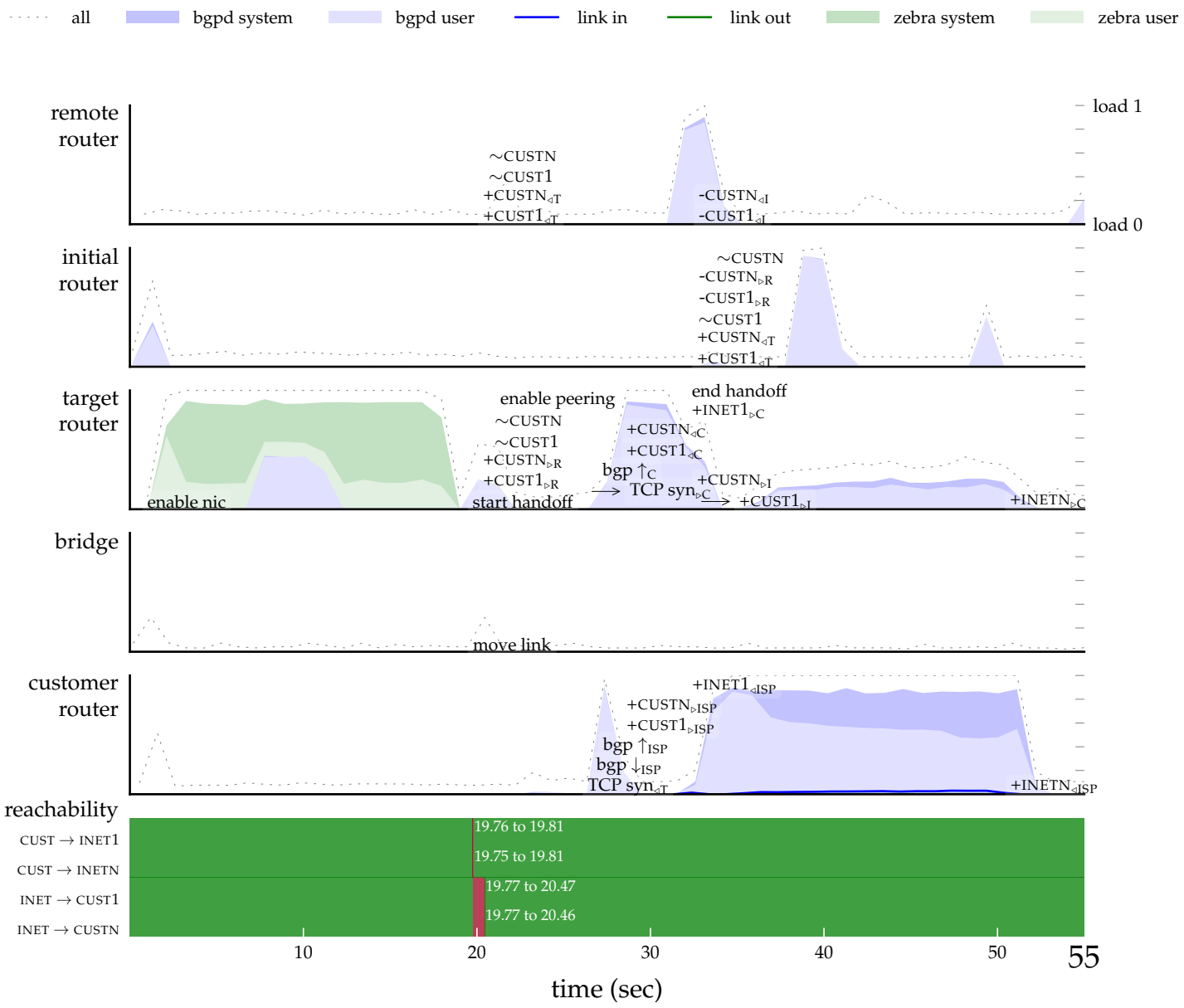


Figure A.24: Full system chart for ZIRO, following the removal of our CPU optimizations for `zebra` and `bgpd`. This chart illustrates the trial with the maximal overall outage time.

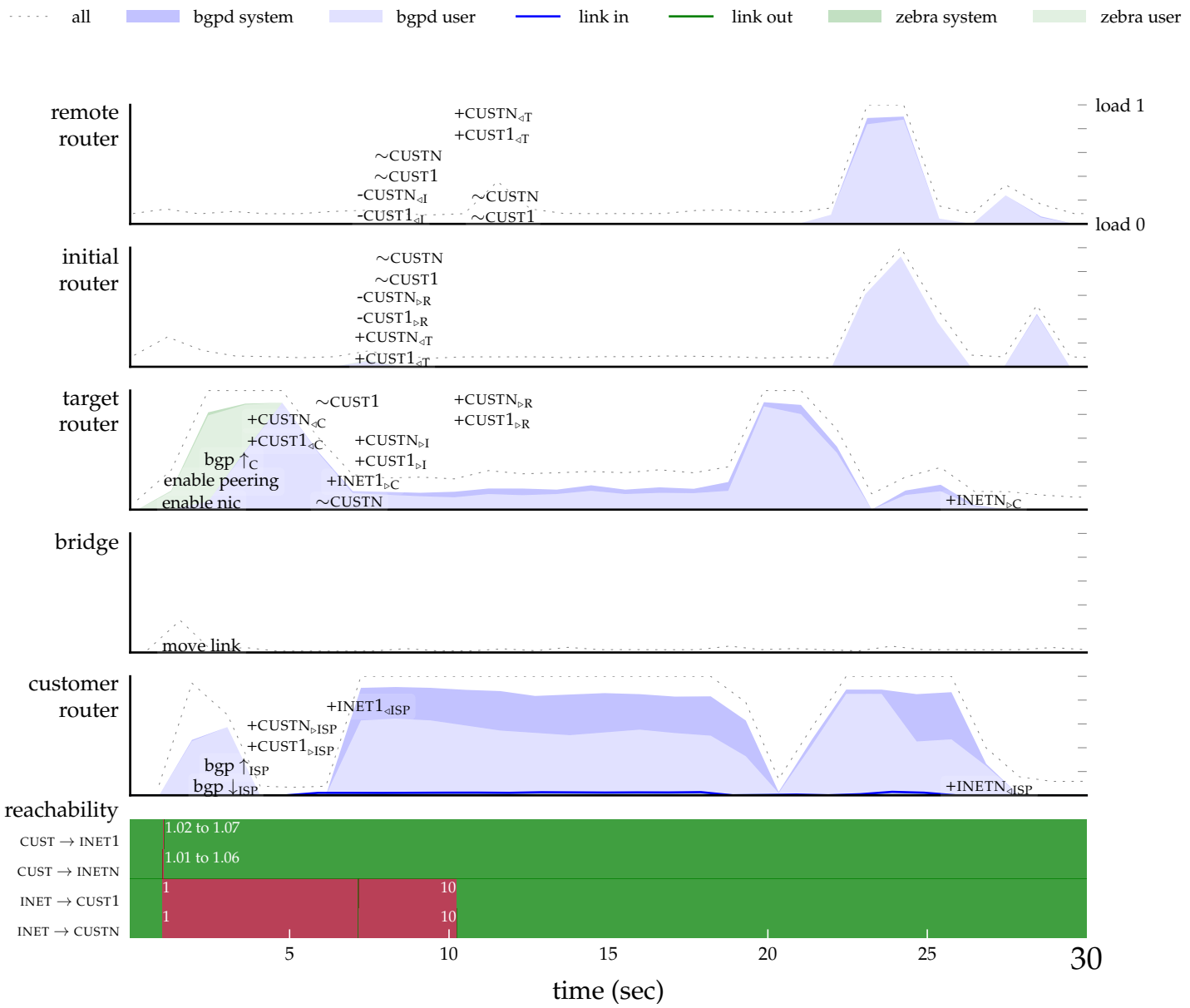


Figure A.25: Full system chart for a trial of rehoming following the patch of Listing 6.2. This trial exhibits the unsuccessful case of the race condition illustrated in Figure 7.3(b).

B
Source Code

Throughout this dissertation, we have presented source code listings for our changes to Quagga. In order to focus attention on the most important aspects of our changes, these listings have limited the amount of context they provide, and excluded those changes not directly related to our core insights.

While appropriate for explaining our insights, the code listings in the body of this dissertation due not provide sufficient detail to reproduce our results. To resolve this limitation, we now provide the full source code for our modifications to Quagga. These listings provide greater context for our changes, and detail ancillary changes required to support our core changes.

The listings herein employ the same typographical conventions as those in the body of the dissertation. Specifically: unmodified lines are shown in gray, removed lines are shown in gray with a horizontal bar, and added lines are shown in bold. Modified lines are indicated by a deletion followed by an addition.

B.1 Implementation of router-id Spoofing

B.1.1 Core functionality

```
void
bgp_open_send (struct peer *peer)
{
  struct stream *s;
  ...

  /* Set open packet values. */
  stream_putc (s, BGP_VERSION_4);        /* BGP version */
  stream_putw (s, (local_as <= BGP_AS_MAX) ? (u_int16_t) local_as
                                           : BGP_AS_TRANS);
  stream_putw (s, send_holdtime);              /* Hold Time */
  stream_put_in_addr (s, &peer->local_id); /* BGP Identifier */
  if (peer->change_local_id.s_addr)      /* BGP Identifier */
    stream_put_in_addr (s, &peer->change_local_id);
  else
    stream_put_in_addr (s, &peer->local_id);
  ...
}
```

Listing B.1: Patch to bgp_open_send, in bgp_packet.c

```
/* BGP neighbor structure. */
struct peer
{
  ...
  /* Local router ID. */
  struct in_addr local_id;

  /* Local router ID. */
  struct in_addr change_local_id;

  ...
};

```

Listing B.2: Patch to struct peer, in bgpd.h

B.1.2 Configuration handling

```
DEFUN (neighbor_local_id,
       neighbor_local_id_cmd,
       NEIGHBOR_CMD2 "local-id A.B.C.D",
       NEIGHBOR_STR
       NEIGHBOR_ADDR_STR2
       "Specify a local-id\n"
       "IP address used as local router-id\n")
{
  struct peer *peer;
  struct in_addr id;
  int ret;

  peer = peer_and_group_lookup_vty (vty, argv[0]);
  if (! peer)
    return CMD_WARNING;

  ret = inet_aton (argv[1], &id);
  if (! ret)
    {
      vty_out (vty, "%% Malformed bgp router identifier%s", VTY_NEWLINE);
      return CMD_WARNING;
    }

  ret = peer_local_id_set (peer, &id);
  return bgp_vty_return (vty, ret);
}
```

Listing B.3: Patch to neighbor_local_id, in bgp_vty.c

```
int
peer_local_id_set (struct peer *peer, struct in_addr *id)
{
  struct bgp *bgp = peer->bgp;
  struct peer_group *group;
  struct listnode *node, *nnode;

  if (peer_group_active (peer))
    return BGP_ERR_INVALID_FOR_PEER_GROUP_MEMBER;

  if (peer->change_local_id.s_addr == id->s_addr)
    return 0;

  peer->change_local_id = *id;

  if (! CHECK_FLAG (peer->sflags, PEER_STATUS_GROUP))
    {
      if (peer->status == Established)
       {
         peer->last_reset = PEER_DOWN_RID_CHANGE;
         bgp_notify_send (peer, BGP_NOTIFY_CEASE,
                          BGP_NOTIFY_CEASE_CONFIG_CHANGE);
       }
      else
        BGP_EVENT_ADD (peer, BGP_Stop);

      return 0;
    }

  group = peer->group;
  for (ALL_LIST_ELEMENTS (group->peer, node, nnode, peer))
    {
      peer->change_local_id = *id;

      if (peer->status == Established)
       {
         peer->last_reset = PEER_DOWN_RID_CHANGE;
         bgp_notify_send (peer, BGP_NOTIFY_CEASE,
                          BGP_NOTIFY_CEASE_CONFIG_CHANGE);
       }
      else
        BGP_EVENT_ADD (peer, BGP_Stop);
    }

  return 0;
}
```

Listing B.4: Patch to peer_local_id_set, in bgpd.c

```
void
bgp_vty_init (void)
{
  ...
  install_element (BGP_NODE, &no_neighbor_peer_group_cmd);
  install_element (BGP_NODE, &no_neighbor_peer_group_remote_as_cmd);

  /* "neighbor local-id" commands. */
  install_element (BGP_NODE, &neighbor_local_id_cmd);

  /* "neighbor local-as" commands. */
  install_element (BGP_NODE, &neighbor_local_as_cmd);
  install_element (BGP_NODE, &neighbor_local_as_no_prepend_cmd);
  ...
}
```

Listing B.5: Patch to bgp_vty_init, in bgp_vty.c

```
static void
bgp_config_write_peer (struct vty *vty, struct bgp *bgp,
                       struct peer *peer, afi_t afi, safi_t safi)
{
  ...
  if (afi == AFI_IP && safi == SAFI_UNICAST)
    {
      ...
      /* local-as. */
      if (peer->change_local_as)
        if (! peer_group_active (peer))
          vty_out (vty, " neighbor %s local-as %u%s%s", addr,
                   peer->change_local_as,
                   CHECK_FLAG (peer->flags, PEER_FLAG_LOCAL_AS_NO_PREPEND) ?
                   " no-prepend" : "", VTY_NEWLINE);

      /* local-id */
      if (peer->change_local_id.s_addr)
        if (! peer_group_active (peer))
          vty_out (vty, " neighbor %s local-id %s%s", addr,
                   inet_ntoa(peer->change_local_id), VTY_NEWLINE);

      ...
    }
  ...
}
```

Listing B.6: Patch to bgp_config_write_peer, in bgpd.c

```
static void
peer_global_config_reset (struct peer *peer)
{
  peer->weight = 0;
  peer->change_local_as = 0;
  peer->change_local_id.s_addr = 0;
  ...
}
```

Listing B.7: Patch to peer_global_config_reset, in bgpd.c

```
static void
peer_group2peer_config_copy (struct peer_group *group, struct peer *peer,
                             afi_t afi, safi_t safi)
{
  ...
  conf = group->conf;
  ...

  /* remote-as */
  if (conf->change_local_as)
    peer->change_local_as = conf->change_local_as;

  /* change_local_id */
  if (conf->change_local_id.s_addr)
    peer->change_local_id.s_addr = conf->change_local_id.s_addr;

  ...
}
```

Listing B.8: Patch to peer_group2peer_config_copy, in bgpd.c

```
int                                                                    1
bgp_router_id_set (struct bgp *bgp, struct in_addr *id)                2
{                                                                      3
  struct peer *peer;                                                   4
  struct listnode *node, *nnode;                                       5
  ...                                                                  6
                                                                       7
  /* Set all peer's local identifier with this value. */               8
  for (ALL_LIST_ELEMENTS (bgp->peer, node, nnode, peer))               9
    {                                                                  10
      if (peer->change_local_id.s_addr != 0)                           11
        continue;                                                      12
                                                                       13
      IPV4_ADDR_COPY (&peer->local_id, id);                            14
      ...                                                              15
    }                                                                  16
  return 0;                                                            17
}                                                                      18
```

Listing B.9: Patch to bgp_router_id_set, in bgpd.c

```
extern int peer_allowas_in_set (struct peer *, afi_t, safi_t, int);    1
extern int peer_allowas_in_unset (struct peer *, afi_t, safi_t);       2
                                                                       3
extern int peer_local_id_set (struct peer *, struct in_addr *);        4
                                                                       5
extern int peer_local_as_set (struct peer *, as_t, int);               6
extern int peer_local_as_unset (struct peer *);                        7
```

Listing B.10: Patch to toplevel of bgpd.h

B.2 Improvements to Quagga's Graceful Restart Implementation

```
1  void
2  bgp_open_capability (struct stream *s, struct peer *peer)
3  {
4    ...
5    /* Graceful restart capability */
6    if (bgp_flag_check (peer->bgp, BGP_FLAG_GRACEFUL_RESTART))
7      {
8        size_t cap_len_pos, cap_len_pos_2, cap_end;
9
10       SET_FLAG (peer->cap, PEER_CAP_RESTART_ADV);
11       stream_putc (s, BGP_OPEN_OPT_CAP);
12       cap_len_pos = stream_get_endp(s);
13       stream_putc (s, CAPABILITY_CODE_RESTART_LEN + 2);
14       stream_putc (s, CAPABILITY_CODE_RESTART);
15       cap_len_pos_2 = stream_get_endp(s);
16       stream_putc (s, CAPABILITY_CODE_RESTART_LEN);
17       /* NB: also encodes restart flags (mukesh.20100419) */
18       stream_putw (s, peer->bgp->restart_time);
19     }
20
21       for (afi = AFI_IP ; afi < AFI_MAX ; afi++)
22         for (safi = SAFI_UNICAST ; safi < SAFI_MAX ; safi++)
23           if (peer->afc[afi][safi])
24             {
25               stream_putw (s, afi);
26               stream_putc (s, safi);
27               /*  flag: state is preserved (mukesh.20100518) */
28               stream_putc (s, RESTART_F_BIT);
29             }
30
31       cap_end = stream_get_endp(s)-1;
32       stream_putc_at (s, cap_len_pos, cap_end-cap_len_pos);
33       stream_putc_at (s, cap_len_pos_2, cap_end-cap_len_pos_2);
34     }
35   ...
36 }
```

Listing B.11: Patch to bgp_open_capability, in bgp_open.c

```
static int                                                                          1
bgp_capability_restart (struct peer *peer, struct capability_header *caphdr)       2
{                                                                                   3
  struct stream *s = BGP_INPUT (peer);                                              4
  ...                                                                               5
  ~~while (stream_get_getp (s) + 4 < end)~~                                         6
  while (stream_get_getp (s) + 4 <= end)                                            7
    {                                                                               8
      afi_t afi = stream_getw (s);                                                  9
      safi_t safi = stream_getc (s);                                                10
      u_char flag = stream_getc (s);                                                11
                                                                                    12
      ...                                                                           13
      if (!bgp_afi_safi_valid_indices (afi, &safi))                                 14
        {                                                                           15
          ...                                                                       16
        }                                                                           17
      else if (!peer->afc[afi][safi])                                               18
        {                                                                           19
          ...                                                                       20
        }                                                                           21
      else                                                                          22
        {                                                                           23
          ...                                                                       24
          SET_FLAG (peer->af_cap[afi][safi], PEER_CAP_RESTART_AF_RCV);              25
          if (CHECK_FLAG (flag, RESTART_F_BIT))                                     26
            SET_FLAG (peer->af_cap[afi][safi], PEER_CAP_RESTART_AF_PRESERVE_RCV);   27
                                                                                    28
        }                                                                           29
    }                                                                               30
  return 0;                                                                         31
}                                                                                   32
```

Listing B.12: Patch to bgp_capability_restart, in bgp_open.c

```
1  static int
2  bgp_open_receive (struct peer *peer, bgp_size_t size)
3  {
4    ...
5    /* Hack part. */
6    if (CHECK_FLAG (peer->sflags, PEER_STATUS_ACCEPT_PEER))
7      {
8          if (realpeer->status == Established
9              && CHECK_FLAG (realpeer->sflags, PEER_STATUS_NSF_MODE))
10         {
11           realpeer->last_reset = PEER_DOWN_NSF_CLOSE_SESSION;
12           SET_FLAG (realpeer->sflags, PEER_STATUS_NSF_WAIT);
13           bgp_clear_route_all(realpeer);
14         }
15        ...
16     }
17   ...
18   return 0;
19 }
```

Listing B.13: Patch to bgp_open_receive, in bgp_packet.c

```
static void                                                                    1
bgp_clear_route_table (struct peer *peer, afi_t afi, safi_t safi,              2
                       struct bgp_table *table, struct peer *rsclient,         3
                       enum bgp_clear_route_type purpose)                      4
{                                                                              5
  struct bgp_node *rn;                                                         6
                                                                               7
  ...                                                                          8
  for (rn = bgp_table_top (table); rn; rn = bgp_route_next (rn))               9
    {                                                                          10
      struct bgp_info *ri;                                                     11
      struct bgp_adj_in *ain;                                                  12
      struct bgp_adj_out *aout;                                                13
                                                                               14
      ...                                                                      15
      for (ri = rn->info; ri; ri = ri->next)                                   16
        if (ri->peer == peer || purpose == BGP_CLEAR_ROUTE_MY_RSCLIENT)        17
          {                                                                    18
            struct bgp_clear_node_queue *cnq;                                  19
                                                                               20
            /* both unlocked in bgp_clear_node_queue_del */                    21
            bgp_table_lock (rn->table);                                        22
            bgp_lock_node (rn);                                                23
            cnq = XCALLOC (MTYPE_BGP_CLEAR_NODE_QUEUE,                         24
                           sizeof (struct bgp_clear_node_queue));              25
            cnq->rn = rn;                                                      26
            cnq->purpose = purpose;                                            27
            work_queue_add (peer->clear_node_queue, cnq);                      28
            if (CHECK_FLAG (peer->sflags, PEER_STATUS_NSF_WAIT)) {             29
              /* queueing this request may lead to unexpected behavior,       30
                 if PEER_STATUS_NSF_WAIT changes. instead, process             31
                 immediately. (mukesh.20100610) */                             32
              bgp_clear_route_node(peer->clear_node_queue, cnq);               33
              XFREE (MTYPE_BGP_CLEAR_NODE_QUEUE, cnq);                         34
            } else {                                                           35
              bgp_lock_node (rn);                                              36
              work_queue_add (peer->clear_node_queue, cnq);                    37
            }                                                                  38
            break;                                                             39
          }                                                                    40
      ...                                                                      41
    }                                                                          42
  return;                                                                      43
}                                                                              44
```

Listing B.14: Patch to bgp_clear_route_table, in bgp_route.c

```
/* Finite State Machine structure */
static const struct {
  int (*func) (struct peer *);
  int next_state;
} FSM [BGP_STATUS_MAX - 1][BGP_EVENTS_MAX - 1] =
{
  ...
  {
    /* Established, */
    {bgp_ignore,               Established}, /* BGP_Start                    */
    {bgp_stop,                    Clearing}, /* BGP_Stop                     */
    {bgp_stop,                    Clearing}, /* TCP_connection_open          */
    {bgp_stop,                    Clearing}, /* TCP_connection_closed        */
    {bgp_stop,                  Clearing},        /* TCP_connection_open_failed   */
    {bgp_stop,                    Clearing}, /* TCP_fatal_error              */
    {bgp_stop,                  Clearing},        /* ConnectRetry_timer_expired   */
    {bgp_fsm_holdtime_expire,     Clearing}, /* Hold_Timer_expired           */
    {bgp_fsm_keepalive_expire, Established}, /* KeepAlive_timer_expired      */
    {bgp_stop,                    Clearing}, /* Receive_OPEN_message         */
    {bgp_fsm_keepalive,        Established}, /* Receive_KEEPALIVE_message    */
    {bgp_fsm_update,           Established}, /* Receive_UPDATE_message       */
    {bgp_stop_with_error,         Clearing}, /* Receive_NOTIFICATION_message */
    {bgp_ignore,                     Idle}, /* Clearing_Completed           */
    {bgp_ignore,               Established}, /* Clearing_Completed           */
  },
  ...
};
```

Listing B.15: Patch to static const struct FSM, in bgp_fsm.c

B.3 Understanding CPU Utilization

B.3.1 Capturing scheduler statistics

```
int                                                                                    1
work_queue_run (struct thread *thread)                                                 2
{                                                                                      3
  ...                                                                                  4
  ++(wq->qlen_hist[MIN(get_order(listcount(wq->items)),                                5
                       sizeof(wq->qlen_hist) / sizeof(wq->qlen_hist[0]))]);            6
  for (ALL_LIST_ELEMENTS (wq->items, node, nnode, item))                               7
  {                                                                                    8
    ...                                                                                9
  }                                                                                    10
                                                                                       11
stats: ...                                                                             12
  wq->runs++;                                                                          13
  wq->cycles.total += cycles;                                                          14
  wq->yields += yielded; ...                                                           15
}                                                                                      16
```

Listing B.16: Patch to work_queue_run, in workqueue.c

```
static inline unsigned int                                                             1
get_order (unsigned int num)                                                           2
{                                                                                      3
  unsigned int order = 0;                                                              4
  assert(num);                                                                         5
  num = num / 2;                                                                       6
  while (num)                                                                          7
  {                                                                                    8
    ++order;                                                                           9
    num = num / 2;                                                                     10
  }                                                                                    11
  return order;                                                                        12
}                                                                                      13
```

Listing B.17: Patch to get_order, in workqueue.c

```
1  struct work_queue
2  {
3    ...
4    /* remaining fields should be opaque to users */
5    struct list *items;                /* queue item list */
6    unsigned long runs;                /* runs count */
7
8    unsigned long yields;              /* yield count */
9    /* histogram of log2(listcount(items)) when work_queue_run is called */
10   unsigned long qlen_hist[sizeof(unsigned int)*8];
11
12   ...
13 };
```

Listing B.18: Patch to struct work_queue, in workqueue.h

```
DEFUN(show_work_queues,
      show_work_queues_cmd,
      "show work-queues",
      SHOW_STR
      "Work Queue information\n")
{
  struct listnode *node;
  struct work_queue *wq;
  unsigned int i;

  vty_out (vty,
           "%c %8s %5s %8s %21s%s",
           ' ', "List","(ms) ","Q. Runs","Cycle Counts   ",
           VTY_NEWLINE);
  vty_out (vty,
           "%c %8s %5s %8s %7s %6s %6s %s%s",
           'P',
           "Items",
           "Hold",
           "Total",
           "Best","Gran.","Avg.",
           "Yields",
           "Name",
           VTY_NEWLINE);


  for (ALL_LIST_ELEMENTS_RO ((&work_queues), node, wq))
    {
      vty_out (vty,"%c %8d %5d %8ld %7d %6d %6u %s%s",
      vty_out (vty,"%c %8d %5d %8ld %7d %6d %6u %8ld %s%s",
               (CHECK_FLAG (wq->flags, WQ_UNPLUGGED) ? ' ' : 'P'),
               listcount (wq->items),
               wq->spec.hold,
               wq->runs,
               wq->cycles.best, wq->cycles.granularity,
                 (wq->runs) ?
                   (unsigned int) (wq->cycles.total / wq->runs) : 0,
               wq->yields,
               wq->name,
               VTY_NEWLINE);
      for (i=0; i < sizeof(wq->qlen_hist)/sizeof(wq->qlen_hist[0]); ++i) {
        vty_out (vty, "%lu ", wq->qlen_hist[i]);
      }
      vty_out (vty, "Qlen Histogram (1/2/4/../2%d)%s",
               sizeof(wq->qlen_hist)/sizeof(wq->qlen_hist[0])-1, VTY_NEWLINE);
    }

  return CMD_SUCCESS;
}
```

Listing B.19: Patch to show_work_queues, in workqueue.c

B.3.2 Capturing hash table statistics

Core code

```
struct hash
{
  ...
  /* Backet alloc. */
  unsigned long count;
  unsigned int count;

  /* Highwater mark. */
  unsigned int high_count;

  /* Name. We own the memory. */
  char *name;
};
```

Listing B.20: Patch to struct hash, in hash.h

```
/* Lookup and return hash backet in hash.  If there is no
   corresponding hash backet and alloc_func is specified, create new
   hash backet.  */
void *
hash_get (struct hash *hash, void *data, void * (*alloc_func) (void *))
{
  ...
  if (alloc_func)
    {
      ...
      hash->index[index] = backet;
      hash->count++;
      hash->high_count = MAX(hash->high_count, hash->count);
      return backet->data;
    }
  return NULL;
}
```

Listing B.21: Patch to hash_get, in hash.c

```
#include "hash.h"
#include "memory.h"
#include "linklist.h"
#include "command.h"

static struct list *hash_table_list = NULL;
```

Listing B.22: Patch to toplevel of hash.c

```
/* Allocate a new hash.  */
struct hash *
hash_create_size (unsigned int size, unsigned int (*hash_key) (void *),
                                     int (*hash_cmp) (const void *, const void *))
                  int (*hash_cmp) (const void *, const void *),
                  const char *name)
{
  ...
  hash->count = 0;
  hash->high_count = 0;
  hash->name = XSTRDUP (MTYPE_HASH_NAME, name);

  if (!hash_table_list)
    hash_table_list = list_new();
  listnode_add(hash_table_list, hash);

  return hash;
}
```

Listing B.23: Patch to hash_create_size, in hash.c

```
struct hash *
hash_create (unsigned int (*hash_key) (void *),
             int (*hash_cmp) (const void *, const void *))
             int (*hash_cmp) (const void *, const void *),
             const char *name)
{
  return hash_create_size (HASHTABSIZE, hash_key, hash_cmp);
  return hash_create_size (HASHTABSIZE, hash_key, hash_cmp, name);
}
```

Listing B.24: Patch to hash_create, in hash.c

```
/* Free hash memory.  You may call hash_clean before call this
   function.  */
void
hash_free (struct hash *hash)
{
  listnode_delete(hash_table_list, hash);
  XFREE (MTYPE_HASH_INDEX, hash->index);
  XFREE (MTYPE_HASH_NAME, hash->name);
  XFREE (MTYPE_HASH, hash);
}
```

Listing B.25: Patch to hash_free, in hash.c

```
DEFUN(show_hash_tables,
      show_hash_tables_cmd,
      "show hash-tables",
      SHOW_STR
      "Hash Table information\n")
{
  struct listnode *node;
  struct hash *hash;

  vty_out (vty, "%8s %8s %8s %s%s",
           "ItemsNow",
           "ItemsEver",
           "Buckets",
           "Name",
           VTY_NEWLINE);

  for (ALL_LIST_ELEMENTS_RO (hash_table_list, node, hash))
    {
      vty_out (vty, "%8u %8u %8u %s%s",
               hash->count,
               hash->high_count,
               hash->size,
               (hash->name ? hash->name : "(unnamed)"), VTY_NEWLINE);
    }

  return CMD_SUCCESS;
}
```

Listing B.26: Patch to show_hash_tables, in hash.c

Naming hash table allocations

```
void                                                                    1
bgp_sync_init (struct peer *peer)                                       2
{                                                                       3
#define STRLEN_NUM(num) (sizeof(#num)-1)                                4
  afi_t afi;                                                            5
  safi_t safi;                                                          6
  struct bgp_synchronize *sync;                                         7
  char baa_hash_name[SU_ADDRSTRLEN +                                    8
                     STRLEN_NUM(AFI_MAX) +                              9
                     STRLEN_NUM(SAFI_MAX) +                             10
                     sizeof("baa  /") + 1];                             11
                                                                        12
  for (afi = AFI_IP; afi < AFI_MAX; afi++)                              13
    for (safi = SAFI_UNICAST; safi < SAFI_MAX; safi++)                  14
      {                                                                 15
        snprintf(baa_hash_name, sizeof(baa_hash_name),                  16
                 "baa %s %d/%d", peer->host, afi, safi);                17
        sync = XCALLOC (MTYPE_BGP_SYNCHRONISE,                          18
                        sizeof (struct bgp_synchronize));               19
        FIFO_INIT (&sync->update);                                      20
        FIFO_INIT (&sync->withdraw);                                    21
        FIFO_INIT (&sync->withdraw_low);                                22
        peer->sync[afi][safi] = sync;                                   23
        peer->hash[afi][safi] = hash_create (baa_hash_key, baa_hash_cmp); 24
        peer->hash[afi][safi] = hash_create (baa_hash_key, baa_hash_cmp, 25
                                             baa_hash_name);            26
      }                                                                 27
#undef STRLEN_NUM                                                       28
}                                                                       29
```

Listing B.27: Patch to bgp_sync_init, in bgp_advertise.c

```
void
aspath_init (void)
{
  ashash = hash_create_size (32767, aspath_key_make, aspath_cmp);
  ashash = hash_create_size (32767, aspath_key_make, aspath_cmp, "aspath");
}
```

Listing B.28: Patch to aspath_init, in bgp_aspath.c

```
static void
cluster_init (void)
{
  cluster_hash = hash_create (cluster_hash_key_make, cluster_hash_cmp);
  cluster_hash = hash_create (cluster_hash_key_make, cluster_hash_cmp,
                              "cluster");
}
```

Listing B.29: Patch to cluster_init, in bgp_attr.c

```
static void
transit_init (void)
{
  transit_hash = hash_create (transit_hash_key_make, transit_hash_cmp);
  transit_hash = hash_create (transit_hash_key_make, transit_hash_cmp,
                              "transit");
}
```

Listing B.30: Patch to transit_init, in bgp_attr.c

```
static void
attrhash_init (void)
{
  attrhash = hash_create (attrhash_key_make, attrhash_cmp);
  attrhash = hash_create (attrhash_key_make, attrhash_cmp, "attr");
}
```

Listing B.31: Patch to attrhash_init, in bgp_attr.c

```
community_init (void)
{
  comhash = hash_create ((unsigned int (*) (void *))community_hash_make,
                         (int (*) (const void *, const void *))community_cmp);
                         (int (*) (const void *, const void *))community_cmp,
                         "community");
}
```

Listing B.32: Patch to community_init, in bgp_community.c

```
void
ecommunity_init (void)
{
  ecomhash = hash_create (ecommunity_hash_make, ecommunity_cmp);
  ecomhash = hash_create (ecommunity_hash_make, ecommunity_cmp, "ecommunity");
}
```

Listing B.33: Patch to ecommunity_init, in bgp_ecommunity.c

```
distribute_list_init (int node)
{
  disthash = hash_create ((unsigned int (*) (void *)) distribute_hash_make,
                          (int (*) (const void *, const void *)) distribute_cmp);
                          (int (*) (const void *, const void *)) distribute_cmp,
                          "distribute");
  ...
}
```

Listing B.34: Patch to distribute_list_init, in distribute.c

```
void
if_rmap_init (int node)
{
  ifrmaphash = hash_create (if_rmap_hash_make, if_rmap_hash_cmp);
  ifrmaphash = hash_create (if_rmap_hash_make, if_rmap_hash_cmp, "if_rmap");
  if (node == RIPNG_NODE) {
    install_element (RIPNG_NODE, &if_ipv6_rmap_cmd);
    install_element (RIPNG_NODE, &no_if_ipv6_rmap_cmd);
  } else if (node == RIP_NODE) {
    install_element (RIP_NODE, &if_rmap_cmd);
    install_element (RIP_NODE, &no_if_rmap_cmd);
  }
}
```

Listing B.35: Patch to if_rmap_init, in if_rmap.c

```
/* Allocate new thread master.  */
struct thread_master *
thread_master_create ()
{
  if (cpu_record == NULL)
    cpu_record
      = hash_create_size (1011, (unsigned int (*) (void *))cpu_record_hash_key,
                          (int (*) (const void *, const void *))cpu_record_hash_cmp);
                          (int (*) (const void *, const void *))cpu_record_hash_cmp,
                          "cpu_record");

  return (struct thread_master *) XCALLOC (MTYPE_THREAD_MASTER,
                                           sizeof (struct thread_master));
}
```

Listing B.36: Patch to thread_master_create, in thread.c

B.3.3 Miscellany

```
/* Initialize command interface. Install basic nodes and commands. */
void
cmd_init (int terminal)
{
  ...
  if (terminal)
    {
      ...
      install_element (VIEW_NODE, &show_hash_tables_cmd);
      install_element (ENABLE_NODE, &show_hash_tables_cmd);
    }
  srand(time(NULL));
}
```

Listing B.37: Patch to cmd_init, in command.c

```
extern struct hash *hash_create (unsigned int (*) (void *),
                                 int (*) (const void *, const void *));
                                 int (*) (const void *, const void *),
                                 const char *name);
extern struct hash *hash_create_size (unsigned int, unsigned int (*) (void *),
                                      int (*) (const void *, const void *));
                                      int (*) (const void *, const void *),
                                      const char *name);
..
/* Helper, exported for command.c */
extern struct cmd_element show_hash_tables_cmd;
```

Listing B.38: Patch to toplevel of hash.h

```
struct memory_list memory_list_lib[] =
{
  ...
  { MTYPE_HASH,                  "Hash"                        },
  { MTYPE_HASH_BACKET,           "Hash Bucket"                 },
  { MTYPE_HASH_INDEX,            "Hash Index"                  },
  { MTYPE_HASH_NAME,             "Hash Name"                   },
  ...
};
```

Listing B.39: Patch to struct memory_list, in memtypes.c

```
#include "memory.h"
#include "log.h"
#include <lib/version.h>
#include "thread.h"
#include "vector.h"
#include "vty.h"
#include "command.h"
#include "workqueue.h"
#include "hash.h"
```

Listing B.40: Patch to toplevel of command.c

B.4 Reducing CPU Utilization

B.4.1 Resolving scheduler bug

```
int                                                                                 1
work_queue_run (struct thread *thread)                                              2
{                                                                                   3
  ...                                                                               4
  ++(wq->qlen_hist[MIN(get_order(listcount(wq->items)),                             5
                       sizeof(wq->qlen_hist) / sizeof(wq->qlen_hist[0]))]);         6
  while (listcount(wq->items))                                                      7
  for (ALL_LIST_ELEMENTS (wq->items, node, nnode, item))                            8
  {                                                                                 9
    ...                                                                             10
  }                                                                                 11
  ...                                                                               12
}                                                                                   13
```

Listing B.41: Patch to work_queue_run, in workqueue.c

B.4.2 Improving hash table performance

```
...
#define BGP_ATTR_UNIQ_COUNT_EST 55000
...
```

Listing B.42: Patch to toplevel of bgp_attr.h

```
void
bgp_sync_init (struct peer *peer)
{
  afi_t afi;
  safi_t safi;
  struct bgp_synchronize *sync;

  for (afi = AFI_IP; afi < AFI_MAX; afi++)
    for (safi = SAFI_UNICAST; safi < SAFI_MAX; safi++)
      {
       ...
       peer->hash[afi][safi] = hash_create (baa_hash_key, baa_hash_cmp);
       if ((afi == AFI_IP) && (safi == SAFI_UNICAST)) {
         peer->hash[afi][safi] = hash_create_size (BGP_ATTR_UNIQ_COUNT_EST,
                                                   baa_hash_key, baa_hash_cmp);
       } else {
         peer->hash[afi][safi] = hash_create (baa_hash_key, baa_hash_cmp);
       }
      }
}
```

Listing B.43: Patch to bgp_sync_init, in bgp_advertise.c

```
static void                                                                    1
attrhash_init (void)                                                           2
{                                                                              3
  attrhash = hash_create (attrhash_key_make, attrhash_cmp);                    4
  /* NB: make the hash twice as large as the number of expected attributes,    5
     to account for copies made due to, e.g., next-hop-self.                   6
     (mukesh.20100815) */                                                      7
  attrhash = hash_create_size (BGP_ATTR_UNIQ_COUNT_EST * 2,                    8
                               attrhash_key_make, attrhash_cmp);               9
}                                                                              10
```

Listing B.44: Patch to attrhash_init, in bgp_attr.c

B.5 Scheduling Optimizations

B.5.1 Improving session establishment time

Core functionality

```
int
peer_open (struct peer *peer)
{
  /* force back to idle, but set timer to zero, for immediate open */
  peer->v_start = 0;
  BGP_EVENT_ADD (peer, BGP_Stop);
  return 0;
}
```

Listing B.45: Patch to peer_open, in bgpd.c

User interface

```
DEFUN (open_ip_bgp_peer,
       open_ip_bgp_peer_cmd,
       "open ip bgp (A.B.C.D|X:X::X:X)",
       OPEN_STR
       IP_STR
       BGP_STR
       "BGP neighbor IP address to open\n"
       "BGP IPv6 neighbor to open\n")
{
  return bgp_open_vty (vty, argv[0]);
}
```

Listing B.46: Patch to open_ip_bgp_peer, in bgp_vty.c

```
static int
bgp_open_vty (struct vty *vty, const char *arg)
{
  struct bgp *bgp;

  bgp = bgp_get_default ();
  if (bgp == NULL)
    {
      vty_out (vty, "No BGP process is configured%s", VTY_NEWLINE);
      return CMD_WARNING;
    }

  return bgp_open (vty, bgp, arg);
}

```

Listing B.47: Patch to bgp_open_vty, in bgp_vty.c

```
static int
bgp_open (struct vty *vty, struct bgp *bgp, const char *arg)
{
  int ret;
  struct peer *peer;
  union sockunion su;

  /* Make sockunion for lookup. */
  ret = str2sockunion (arg, &su);
  if (ret < 0)
    {
      vty_out (vty, "Malformed address: %s%s", arg, VTY_NEWLINE);
      return CMD_WARNING;
    }
  peer = peer_lookup (bgp, &su);
  if (! peer)
    {
      vty_out (vty, "%%BGP: Unknown neighbor - \"%s\"%s", arg, VTY_NEWLINE);
      return CMD_WARNING;
    }

  ret = peer_open (peer);

  if (ret < 0)
    bgp_open_vty_error (vty, peer, ret);

  return CMD_SUCCESS;
}
```

Listing B.48: Patch to bgp_open, in bgp_vty.c

```
static void
bgp_open_vty_error (struct vty *vty, struct peer *peer, int error)
{
  vty_out (vty, "%%BGP: unable to open %s%s", peer->host, VTY_NEWLINE);
}
```

Listing B.49: Patch to bgp_open_vty_error, in bgp_vty.c

```
void                                                                    1
bgp_vty_init (void)                                                     2
{                                                                       3
  ...                                                                   4
  /* "clear ip bgp commands" */                                         5
  install_element (ENABLE_NODE, &clear_ip_bgp_all_cmd); ...             6
                                                                        7
  install_element (ENABLE_NODE, &open_ip_bgp_peer_cmd);                 8
                                                                        9
  /* "clear ip bgp neighbor soft in" */                                 10
  install_element (ENABLE_NODE, &clear_ip_bgp_all_soft_in_cmd);         11
  ...                                                                   12
}                                                                       13
```

Listing B.50: Patch to bgp_vty_init, in bgp_vty.c

Miscellany

```
1 extern int peer_maximum_prefix_set (struct peer *, afi_t, safi_t, u_int32_t, u_char, int, u_-
  int16_t);
2 extern int peer_maximum_prefix_unset (struct peer *, afi_t, safi_t);
3
4 extern int peer_open (struct peer *);
5
6 extern int peer_clear (struct peer *);
7 extern int peer_clear_soft (struct peer *, afi_t, safi_t, enum bgp_clear_type);
```

Listing B.51: Patch to toplevel of bgpd.h

```
1 #define NO_STR "Negate a command or set its defaults\n"
2 #define REDIST_STR "Redistribute information from another routing protocol\n"
3 #define CLEAR_STR "Reset functions\n"
4 #define OPEN_STR "Open functions\n"
5 #define RIP_STR "RIP information\n"
6 #define BGP_STR "BGP information\n"
7 #define OSPF_STR "OSPF information\n"
```

Listing B.52: Patch to toplevel of command.h

B.5.2 Improving route propagation delay

```
int                                                                                    1
bgp_write (struct thread *thread)                                                      2
static int                                                                             3
_bgp_write (struct thread *thread, unsigned int max_packets)                           4
{ ...                                                                                  5
  unsigned int count = 0; ...                                                          6
    /* Nonblocking write until TCP output buffer is full.  */                          7
  while (1)                                                                            8
    { ...                                                                              9
      s = bgp_write_packet (peer); ...                                                 10
      writenum = stream_get_endp (s) - stream_get_getp (s); ...                        11
      num = write (peer->fd, STREAM_PNT (s), writenum); ...                            12
      if (++count >= BGP_WRITE_PACKET_MAX)                                             13
      if ((max_packets) && (++count >= max_packets))                                   14
        break;                                                                         15
    }                                                                                  16
  ...                                                                                  17
}                                                                                      18
                                                                                       19
int                                                                                    20
bgp_write (struct thread *thread)                                                      21
{                                                                                      22
  _bgp_write(thread, BGP_WRITE_PACKET_MAX);                                            23
}                                                                                      24
                                                                                       25
```

Listing B.53: Patch to bgp_write, in bgp_packet.c

```
1  static int
2  bgp_update_receive (struct peer *peer, bgp_size_t size)
3  {
4    ...
5    /* NLRI is processed only when the peer is configured specific
6       Address Family and Subsequent Address Family. */
7    if (peer->afc[AFI_IP][SAFI_UNICAST])
8      {
9        ...
10       if (! attribute_len && ! withdraw_len)
11         {
12           struct listnode *node, *nnode;
13           struct peer *p;
14
15           /* End-of-RIB received */
16           SET_FLAG (peer->af_sflags[AFI_IP][SAFI_UNICAST],
17                     PEER_STATUS_EOR_RECEIVED);
18
19           /* NSF delete stale route */
20           if (peer->nsf[AFI_IP][SAFI_UNICAST])
21             bgp_clear_stale_route (peer, AFI_IP, SAFI_UNICAST);
22
23           for (ALL_LIST_ELEMENTS (peer->bgp->peer, node, nnode, p))
24             {
25               struct thread t;
26               time_t oldsync;
27
28               oldsync = p->synctime;
29               t.arg = p;
30
31               p->synctime = bgp_clock() + 1;
32               if (bgp_write_proceed(p))
33                 {
34                   /* cancel any pending write thread, since we're taking
35                      care of writes here. (mukesh.20100819). */
36                   BGP_WRITE_OFF(p->t_write);
37                   _bgp_write(&t, 0);
38                 }
39               p->synctime = oldsync;
40             }
41
42           if (BGP_DEBUG (normal, NORMAL))
43             zlog (peer->log, LOG_DEBUG, "rcvd End-of-RIB for IPv4 Unicast from %s",
44                   peer->host);
45         }
46     }
47   ...
48 }
```

Listing B.54: Patch to bgp_update_receive, in bgp_packet.c

B.5.3 Improving route processing delay, Part I

```
static int                                                                      1
bgp_update_receive (struct peer *peer, bgp_size_t size)                         2
{ ...                                                                           3
  /* NLRI is processed only when the peer is configured specific               4
     Address Family and Subsequent Address Family. */                          5
  if (peer->afc[AFI_IP][SAFI_UNICAST])                                          6
    { ...                                                                       7
      if (! attribute_len && ! withdraw_len)                                    8
        { ...                                                                   9
          struct thread t;                                                      10
          int res;                                                              11
                                                                                12
          /* End-of-RIB received */                                             13
          SET_FLAG (peer->af_sflags[AFI_IP][SAFI_UNICAST],                      14
                    PEER_STATUS_EOR_RECEIVED); ...                              15
                                                                                16
          res = WQ_SUCCESS;                                                     17
          t.arg = bm ? bm->process_main_queue : NULL;                           18
          while (bm && bm->process_main_queue && bm->process_main_queue->items  19
                 && listcount(bm->process_main_queue->items) &&                 20
                 (res == WQ_SUCCESS))                                           21
            res = work_queue_run(&t);                                           22
                                                                                23
          for (ALL_LIST_ELEMENTS (peer->bgp->peer, node, nnode, p))             24
            {                                                                   25
              struct thread t;                                                  26
              time_t oldsync;                                                   27
                                                                                28
              if (p == peer)  /* expedite "upstream", not "downstream". */      29
                continue;     /* (mukesh.2010020) */                            30
              ...                                                               31
            } ...                                                               32
        }                                                                       33
    } ...                                                                       34
}                                                                               35
```

Listing B.55: Patch to bgp_update_receive, in bgp_packet.c

```
int
work_queue_run (struct thread *thread)
{
  ...
  while (listcount(wq->items))
    for (ALL_LIST_ELEMENTS (wq->items, node, nnode, item))
      {
        ...
        do
          {
            ret = wq->spec.workfunc (wq, item->data);
            item->ran++;
          }
        while ((ret == WQ_RETRY_NOW)
               && (item->ran < wq->spec.max_retries)); ...
      }
  ...
  return 0;
  return ret;
}
```

Listing B.56: Patch to work_queue_run, in workqueue.c

```
#include "thread.h"
#include "workqueue.h"
#include "stream.h"
#include "network.h"
```

Listing B.57: Patch to toplevel of bgp_packet.c

B.5.4 Improving route processing delay, Part II

Core functionality

```
void                                                                          1
bgp_announce_route (struct peer *peer, afi_t afi, safi_t safi)                2
{                                                                             3
  ...                                                                         4
  /* First update is deferred until ORF or ROUTE-REFRESH is received */       5
  if (CHECK_FLAG (peer->af_sflags[afi][safi], PEER_STATUS_ORF_WAIT_REFRESH))  6
    return;                                                                   7
                                                                              8
  /* First update is deferred until peer has sent End-of-RIB */              9
  if (CHECK_FLAG (peer->af_sflags[afi][safi], PEER_STATUS_EOR_WAIT))          10
    return;                                                                   11
                                                                              12
  if (safi != SAFI_MPLS_VPN)                                                  13
    bgp_announce_table (peer, afi, safi, NULL, 0);                            14
  ...                                                                         15
}                                                                             16
```

Listing B.58: Patch to bgp_announce_route, in bgp_route.c

```
1  static int
2  bgp_update_receive (struct peer *peer, bgp_size_t size)
3  { ...
4    if (peer->afc[AFI_IP][SAFI_UNICAST])
5      { ...
6        if (! attribute_len && ! withdraw_len)
7          { ...
8            /* End-of-RIB received */
9            SET_FLAG (peer->af_sflags[AFI_IP][SAFI_UNICAST],
10                     PEER_STATUS_EOR_RECEIVED);
11
12           /* NSF delete stale route */
13           if (peer->nsf[AFI_IP][SAFI_UNICAST])
14             bgp_clear_stale_route (peer, AFI_IP, SAFI_UNICAST);
15
16           res = WQ_SUCCESS; ...
17           for (ALL_LIST_ELEMENTS (peer->bgp->peer, node, nnode, p))
18             { ...
19             }
20
21           if (CHECK_FLAG (peer->af_sflags[AFI_IP][SAFI_UNICAST],
22                           PEER_STATUS_EOR_WAIT))
23             {
24               UNSET_FLAG (peer->af_sflags[AFI_IP][SAFI_UNICAST],
25                           PEER_STATUS_EOR_WAIT);
26               bgp_announce_route (peer, AFI_IP, SAFI_UNICAST);
27               if (bgp_write_proceed(peer))
28                 BGP_WRITE_ON(peer->t_write, bgp_write, peer->fd);
29             }
30           ...
31         }
32     } ...
33 }
```

Listing B.59: Patch to bgp_update_receive, in bgp_packet.c

Core support: Deadlock avoidance

```
void                                                                                    1
bgp_open_capability (struct stream *s, struct peer *peer)                               2
{ ...                                                                                   3
  /* Graceful restart capability */                                                     4
  if (bgp_flag_check (peer->bgp, BGP_FLAG_GRACEFUL_RESTART))                            5
    { ...                                                                               6
      SET_FLAG (peer->cap, PEER_CAP_RESTART_ADV);                                       7
      stream_putc (s, BGP_OPEN_OPT_CAP);                                                8
      cap_len_pos = stream_get_endp(s);                                                 9
      stream_putc (s, CAPABILITY_CODE_RESTART_LEN + 2);                                 10
      stream_putc (s, CAPABILITY_CODE_RESTART);                                         11
      cap_len_pos_2 = stream_get_endp(s);                                               12
      stream_putc (s, CAPABILITY_CODE_RESTART_LEN);                                     13
      /* NB: also encodes restart flags (mukesh.20100419) */                            14
      stream_putw (s, peer->bgp->restart_time);                                         15
                                                                                        16
      /* NB: restart_time also encodes restart flags (mukesh.20100822) */               17
      if (CHECK_FLAG (peer->flags, PEER_FLAG_RECEIVE_FIRST)) {                          18
        stream_putw (s, peer->bgp->restart_time | RESTART_F_BIT);                       19
      } else {                                                                          20
        stream_putw (s, peer->bgp->restart_time);                                       21
      }                                                                                 22
      ...                                                                               23
    }                                                                                   24
  ...                                                                                   25
}                                                                                       26
```

Listing B.60: Patch to bgp_open_capability, in bgp_open.c

```
static int
bgp_capability_restart (struct peer *peer, struct capability_header *caphdr)
{
  struct stream *s = BGP_INPUT (peer); ...

  SET_FLAG (peer->cap, PEER_CAP_RESTART_RCV);
  restart_flag_time = stream_getw(s);
  if (CHECK_FLAG (restart_flag_time, RESTART_R_BIT))
    restart_bit = 1;
    {
      restart_bit = 1;
      SET_FLAG (peer->sflags, PEER_STATUS_NSF_RESTARTED);
    }
  UNSET_FLAG (restart_flag_time, 0xF000);
  peer->v_gr_restart = restart_flag_time;
  ...
}
```

Listing B.61: Patch to bgp_capability_restart, in bgp_open.c

```
static int
bgp_establish (struct peer *peer)
{
  ...
  if (CHECK_FLAG (peer->flags, PEER_FLAG_RECEIVE_FIRST) &&
      ! CHECK_FLAG (peer->sflags, PEER_STATUS_NSF_RESTARTED))
    for (afi = AFI_IP ; afi < AFI_MAX ; afi++)
      for (safi = SAFI_UNICAST ; safi < SAFI_MAX ; safi++)
        if (CHECK_FLAG (peer->af_cap[afi][safi], PEER_CAP_RESTART_AF_RCV))
          SET_FLAG (peer->af_sflags[afi][safi], PEER_STATUS_EOR_WAIT);

  bgp_announce_route_all (peer);
  ...
}
```

Listing B.62: Patch to bgp_establish, in bgp_fsm.c

Core support: Update suppression

```
static int
bgp_process_announce_selected (struct peer *peer, struct bgp_info *selected,
                               struct bgp_node *rn, afi_t afi, safi_t safi)
{
  ...
  /* First update is deferred until ORF or ROUTE-REFRESH is received */
  if (CHECK_FLAG (peer->af_sflags[afi][safi],
      PEER_STATUS_ORF_WAIT_REFRESH))
    return 0;

  /* First update is deferred until peer has sent End-of-RIB */
  if (CHECK_FLAG (peer->af_sflags[afi][safi], PEER_STATUS_EOR_WAIT))
    return 0;

  switch (rn->table->type)
    {
      case BGP_TABLE_MAIN:
      /* Announcement to peer->conf.  If the route is filtered,
         withdraw it. */
        if (selected && bgp_announce_check (selected, peer, p, &attr, afi, safi))
          bgp_adj_out_set (rn, peer, p, &attr, afi, safi, selected);
        else
          bgp_adj_out_unset (rn, peer, p, afi, safi);
        break;
      ...
    }
  ...
}
```

Listing B.63: Patch to bgp_process_announce_selected, in bgp_route.c

```
static struct stream *
bgp_write_packet (struct peer *peer)
{ ...
  for (afi = AFI_IP; afi < AFI_MAX; afi++)
    for (safi = SAFI_UNICAST; safi < SAFI_MAX; safi++)
      {
        if (CHECK_FLAG (peer->cap, PEER_CAP_RESTART_RCV) &&
            CHECK_FLAG (peer->af_sflags[afi][safi], PEER_STATUS_EOR_WAIT))
          continue;

        adv = FIFO_HEAD (&peer->sync[afi][safi]->withdraw);
        if (adv)
          {
            s = bgp_withdraw_packet (peer, afi, safi);
            if (s)
              return s;
          }
      }

  for (afi = AFI_IP; afi < AFI_MAX; afi++)
    for (safi = SAFI_UNICAST; safi < SAFI_MAX; safi++)
      {
        if (CHECK_FLAG (peer->cap, PEER_CAP_RESTART_RCV) &&
            CHECK_FLAG (peer->af_sflags[afi][safi], PEER_STATUS_EOR_WAIT))
          continue;

        adv = FIFO_HEAD (&peer->sync[afi][safi]->update);
        if (adv)
          { ...
            if (s)
              return s;
          }
        ...
      }
    ...
}
```

Listing B.64: Patch to bgp_write_packet, in bgp_packet.c

Configuration handling

```
/* neighbor receive-first */
DEFUN (neighbor_receive_first,
       neighbor_receive_first_cmd,
       NEIGHBOR_CMD2 "receive-first",
       NEIGHBOR_STR
       NEIGHBOR_ADDR_STR2
       "wait for peer end-of-rib before sending out routes\n"
       "requires graceful restart\n")
{
  return peer_flag_set_vty (vty, argv[0], PEER_FLAG_RECEIVE_FIRST);
}
```

Listing B.65: Patch to neighbor_receive_first, in bgp_vty.c

```
void
bgp_vty_init (void)
{
  ...
  /* "neighbor receive-first" commands. */
  install_element (BGP_NODE, &neighbor_receive_first_cmd);

  /* "neighbor passive" commands. */
  install_element (BGP_NODE, &neighbor_passive_cmd);
  install_element (BGP_NODE, &no_neighbor_passive_cmd);
  ...
}
```

Listing B.66: Patch to bgp_vty_init, in bgp_vty.c

```
static void
bgp_config_write_peer (struct vty *vty, struct bgp *bgp,
                       struct peer *peer, afi_t afi, safi_t safi)
{
  ...
  /************************************
   ****** Global to the neighbor ******
   ************************************/
  if (afi == AFI_IP && safi == SAFI_UNICAST)
    {
      ...
      /* Local interface name. */
      if (peer->ifname)
        vty_out (vty, " neighbor %s interface %s%s", addr, peer->ifname,
                 VTY_NEWLINE);

      /* Receive first. */
      if (CHECK_FLAG (peer->flags, PEER_FLAG_RECEIVE_FIRST))
        if (! peer_group_active (peer) ||
            ! CHECK_FLAG (g_peer->flags, PEER_FLAG_RECEIVE_FIRST))
          vty_out (vty, " neighbor %s receive-first%s", addr, VTY_NEWLINE);

      /* Passive. */
      if (CHECK_FLAG (peer->flags, PEER_FLAG_PASSIVE))
        if (! peer_group_active (peer) ||
            ! CHECK_FLAG (g_peer->flags, PEER_FLAG_PASSIVE))
          vty_out (vty, " neighbor %s passive%s", addr, VTY_NEWLINE);
      ...
    }
  ...
}
```

Listing B.67: Patch to bgp_config_write_peer, in bgpd.c

Miscellany

```
static const struct peer_flag_action peer_flag_action_list[] =
  {
    { PEER_FLAG_PASSIVE,                 0, peer_change_reset },
    { PEER_FLAG_SHUTDOWN,                0, peer_change_reset },
    { PEER_FLAG_DONT_CAPABILITY,         0, peer_change_none },
    { PEER_FLAG_OVERRIDE_CAPABILITY,     0, peer_change_none },
    { PEER_FLAG_STRICT_CAP_MATCH,        0, peer_change_none },
    { PEER_FLAG_DYNAMIC_CAPABILITY,      0, peer_change_reset },
    { PEER_FLAG_DISABLE_CONNECTED_CHECK, 0, peer_change_reset },
    { PEER_FLAG_RECEIVE_FIRST,           0, peer_change_none },
    { 0, 0, 0 }
  };
```

Listing B.68: Patch to struct peer_flag_action, in bgpd.c

```
/* BGP neighbor structure. */
struct peer
{ ...
  /* Peer status flags. */
  u_int16_t sflags; ...
#define PEER_STATUS_NSF_MODE          (1 << 5) /* NSF aware peer */
#define PEER_STATUS_NSF_WAIT          (1 << 6) /* wait comeback peer */
#define PEER_STATUS_NSF_RESTARTED     (1 << 7) /* peer set R bit in NSF
                                                  capability. */

  /* Peer status af flags (reset in bgp_stop) */
  u_int16_t af_sflags[AFI_MAX][SAFI_MAX]; ...
#define PEER_STATUS_EOR_SEND          (1 << 5) /* end-of-rib send to peer */
#define PEER_STATUS_EOR_RECEIVED      (1 << 6) /* end-of-rib received from peer */
#define PEER_STATUS_EOR_WAIT          (1 << 7) /* waiting for end-of-rib from
                                                  peer before sending updates
                                                  to peer. */
  ...
}
```

Listing B.69: Patch to struct peer, in bgpd.h

B.6 Soft Handoff

Core functionality

```
static int
bgp_announce_check (struct bgp_info *ri, struct peer *peer, struct prefix *p,
                    struct attr *attr, afi_t afi, safi_t safi)
{ ...
  /* Route-Reflect check. */
  if (peer_sort (from) == BGP_PEER_IBGP && peer_sort (peer) == BGP_PEER_IBGP)
  /* NB: special-case prefixes with LOCAL_FLAG_REFLECT, as though they were
     originated by us, rather than reflected by us. (mukesh.20100823) */
  if ((peer_sort (from) == BGP_PEER_IBGP && peer_sort (peer) == BGP_PEER_IBGP)
      && !(ri->attr->local_flags & LOCAL_FLAG_REFLECT))
    reflect = 1;
  else
    reflect = 0;

  /* IBGP reflection check. */
  if (reflect)
    {
      /* A route from a Client peer. */
      if (CHECK_FLAG (from->af_flags[afi][safi], PEER_FLAG_REFLECTOR_CLIENT))
        { ...
          if (bgp_flag_check (bgp, BGP_FLAG_NO_CLIENT_TO_CLIENT))
            if (CHECK_FLAG (peer->af_flags[afi][safi], PEER_FLAG_REFLECTOR_CLIENT))
              return 0;
        }
      else
        {
          /* A route from a Non-client peer. Reflect to all other
             clients. */
          if (! CHECK_FLAG (peer->af_flags[afi][safi], PEER_FLAG_REFLECTOR_CLIENT))
            return 0;
        }
    }
  ...
  return 1;
}
```

Listing B.70: Patch to bgp_announce_check, in bgp_route.c

```
void
bgp_soft_reconfig_in (struct peer *peer, afi_t afi, safi_t safi)
{
  struct bgp_node *rn;
  struct bgp_table *table;
  struct listnode *node, *nnode;
  struct peer *p;
  struct thread t;
  int res;

  if (peer->status != Established)
    return;

  if (safi != SAFI_MPLS_VPN)
    bgp_soft_reconfig_table (peer, afi, safi, NULL);
  ...

  res = WQ_SUCCESS;
  t.arg = bm ? bm->process_main_queue : NULL;
  while (bm && bm->process_main_queue && bm->process_main_queue->items
         && listcount(bm->process_main_queue->items) &&
         (res == WQ_SUCCESS))
    res = work_queue_run(&t);

  for (ALL_LIST_ELEMENTS (peer->bgp->peer, node, nnode, p))
    {
      if (p->status != Established)
        continue;

      t.arg = p;
      p->synctime = bgp_clock() + 1;
      /* cancel any pending write thread, since we're taking
         care of writes here. (mukesh.20100625). */
      BGP_WRITE_OFF(p->t_write);
      bgp_write(&t);
    }
}
```

Listing B.71: Patch to bgp_soft_reconfig_in, in bgp_route.c

```
1  static int
2  bgp_update_receive (struct peer *peer, bgp_size_t size)
3  { ...
4    if (peer->afc[AFI_IP][SAFI_UNICAST])
5      { ...
6        if (! attribute_len && ! withdraw_len)
7          { ...
8            /* End-of-RIB received */
9            SET_FLAG (peer->af_sflags[AFI_IP][SAFI_UNICAST],
10                     PEER_STATUS_EOR_RECEIVED);
11
12           /* NSF delete stale route */
13           if (peer->nsf[AFI_IP][SAFI_UNICAST])
14             bgp_clear_stale_route (peer, AFI_IP, SAFI_UNICAST);
15
16           res = WQ_SUCCESS; ...
17           for (ALL_LIST_ELEMENTS (peer->bgp->peer, node, nnode, p))
18             { ...
19             }
20
21           if (CHECK_FLAG (peer->af_sflags[AFI_IP][SAFI_UNICAST],
22                           PEER_STATUS_EOR_WAIT))
23             { ...
24             }
25
26           if (BGP_DEBUG (normal, NORMAL))
27             zlog (peer->log, LOG_DEBUG, "rcvd End-of-RIB for IPv4 Unicast from %s",
28                   peer->host);
29
30           if (bgp_flag_check (peer->bgp, BGP_FLAG_LOG_NEIGHBOR_CHANGES))
31             zlog_info ("rcvd End-of-RIB for IPv4 Unicast from %s", peer->host);
32         }
33     }
34   ...
35 }
```

Listing B.72: Patch to bgp_update_receive, in bgp_packet.c

Configuration handling

```
/* For reflect set. */
static route_map_result_t
route_set_reflect (void *rule, struct prefix *prefix, route_map_object_t type, void *object)
{
  struct bgp_info *bgp_info;

  if (type == RMAP_BGP)
    {
      bgp_info = object;
      bgp_info->attr->local_flags |= LOCAL_FLAG_REFLECT;
    }

  return RMAP_OKAY;
}

/* Set reflect rule structure. */
struct route_map_rule_cmd route_set_reflect_cmd =
{
  "reflect",
  route_set_reflect,
  NULL,
  NULL,
};
```

Listing B.73: Patch to route_set_reflect, in bgp_attr.h

```
DEFUN (set_reflect,
       set_reflect_cmd,
       "set reflect",
       SET_STR
       "Local flag indicating route should be reflected to IBGP peers\n" )
{
  return bgp_route_set_add (vty, vty->index, "reflect", NULL);
}
```

Listing B.74: Patch to set_reflect, in bgp_attr.h

```
DEFUN (no_set_reflect,
       no_set_reflect_cmd,
       "no set reflect",
       NO_STR
       SET_STR
       "Local flag indicating route should be reflected to IBGP peers\n" )
{
  return bgp_route_set_delete (vty, vty->index, "reflect", NULL);
}
```

Listing B.75: Patch to no_set_reflect, in bgp_attr.h

```
void
bgp_route_map_init (void)
{
  ...
  route_map_install_set (&route_set_ecommunity_rt_cmd);
  route_map_install_set (&route_set_ecommunity_soo_cmd);
  route_map_install_set (&route_set_reflect_cmd);
  ...
  install_element (RMAP_NODE, &no_set_originator_id_cmd);
  install_element (RMAP_NODE, &no_set_originator_id_val_cmd);
  install_element (RMAP_NODE, &set_reflect_cmd);
  install_element (RMAP_NODE, &no_set_reflect_cmd);
  ...
}
```

Listing B.76: Patch to bgp_route_map_init, in bgp_attr.h

Miscellany

```
...
/* Local flags for BGP paths. */
#define LOCAL_FLAG_REFLECT      0x01
...
struct attr
{
  ...
  /* Path origin attribute */
  u_char origin;

  /* Local flags */
  u_char local_flags;
};
...
```

Listing B.77: Patch to toplevel of bgp_attr.h

Bibliography

[1] Adobe Systems. Adobe Flash Media Server ActionScript 2.0 Language Reference / ActionScript 2.0 Language Reference / NetStream class / NetStream.bufferTime. API documentation, retrieved March 20, 2011. http://livedocs.adobe.com/flashmediaserver/3.0/hpdocs/help.html?content=00000175.html#157469. (Cited in Section 7.5.2.)

[2] Adobe Systems. Adobe Unveils Flash Media Server 3.4 Software. Press Release. http://www.adobe.com/aboutadobe/pressroom/pressreleases/pdfs/200811/111708AdobeFMS35.pdf, November 2008. (Cited in Section 7.5.2.)

[3] Sharad Agarwal, Chen-Nee Chuah, Supratik Bhattacharyya, and Christophe Diot. Impact of BGP dynamics on router CPU utilization. In *Passive and Active Measurement Workshop*, 2004. (Cited in Section 8.2.2.)

[4] Mukesh Agrawal, Susan R. Bailey, Albert Greenberg, Jorge Pastor, Panagiotis Sebos, Srinivasan Seshan, Kobus van der Merwe, and Jennifer Yates. RouterFarm: Towards a dynamic, manageable network edge. In *Proceedings of the ACM SIGCOMM Workshop on Internet Network Management*, September 2006. (Cited in Sections 8.2.1 and 3.)

[5] Akamai Technologies. The State of the Internet. Whitepaper. http://www.akamai.com/dl/whitepapers/Akamai_state_of_internet_q32010.pdf, 3rd Quarter 2010. (Cited in Section 7.5.2.)

[6] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 131–145, Banff, Canada, October 2001. (Cited in Section 8.2.1.)

[7] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Rohit Rao. Improving Web availability for clients with MONET. In *2nd Symposium on Network Systems Design and Implementation (NSDI 2005)*, Boston, MA, May 2005. (Cited in Sections 2 and 8.2.1.)

[8] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, 1997. (Cited in Section 5.1.)

[9] AT&T. Global IP Network Averages. Web site, retrieved March 20, 2011. http://ipnetwork.bgtmo.ip.att.net/pws/averages.html, 2011. (Cited in Section 7.5.2.)

[10] Tony Bates, Ravi Chandra, and Enke Chen. BGP route reflection — An alternative to full mesh IBGP. RFC 2796, April 2000. (Cited in Section 8.3.5.)

[11] Glenn Gabriel Ben-Yosef. The evolution of resiliency. http://www.networkworld.com/supp/ii2003/0224intelinfranetwork.html, February 2003. (Cited in Section 8.2.1.)

[12] Peter Bickford. Worth the wait? *Netscape DevEdge: View Source*, October 1997. (Cited in Section 7.5.2.)

[13] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the eye of the beholder: meeting users' requirements for internet quality of service. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 297–304, 2000. (Cited in Section 7.5.2.)

[14] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the SIGCOMM '94 Symposium on Communications Architectures and Protocols*, August 1994. (Cited in Section 8.2.1.)

[15] Henning Brauer. A Secure BGP Implemenation. EuroBSDCon 2004. http://quigon.bsws.de/papers/euroBSDCon2004/index.html. (Cited in Section 8.2.3.)

[16] Henning Brauer and Claudio Jeker. OpenBGPD. http://www.openbgpd.org/. (Cited in Section 2.1.4.)

[17] Di-Fa Chang, Ramesh Govindan, and John Heidemann. An empirical study of router repsonse to large BGP routing table load. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*, pages 203–208, Marseille, France, 2002. (Cited in Section 8.2.2.)

[18] Cisco Systems. Cisco Nonstop Forwarding for BGP: Deployment & Troubleshooting. http://www.cisco.com/en/US/products/ps6550/products_white_paper09186a008016317c.shtml. (Cited in Sections 1.1 and 8.2.1.)

[19] Cisco Systems. GRP redundant processor support. http://www.cisco.com/en/US/products/sw/iosswrel/ps1824/products_feature_guide09186a00800a17ca.html. (Cited in Section 8.2.1.)

[20] Cisco Systems. Route processor redundancy plus for the Cisco 12000 series Internet router. http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120st/120st17/rpr_plus.htm. (Cited in Section 8.2.1.)

[21] Cisco Systems. Stateful switchover. http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120s/120s22/sso120s.htm. (Cited in Section 8.2.1.)

[22] Cisco Systems. A brief overview of Packet over SONET APS. Cisco website, Document ID 13566, July 2004. (Cited in Section 8.2.1.)

[23] Cisco Systems. Cisco Nonstop Forwarding with Stateful Switchover Deployment Guide. http://www.cisco.com/en/US/technologies/tk869/tk769/technologies_white_paper0900aecd801dc5e2.pdf, August 2006. (Cited in Section 8.2.1.)

[24] Cisco Systems. Cisco IOS IP Routing: BGP Command Reference. http://www.cisco.com/en/US/docs/ios/iproute_bgp/command/reference/irg_bgp1.pdf, July 2010. (Cited in Section 7.2.4.)

[25] Cisco Systems. Cross-platform release notes for Cisco IOS release 12.4 T, part 5: Caveats for 12.4(11)T through 12.4(24)T3. http://www.cisco.com/en/US/docs/ios/12_4t/release/notes/124TCAVS.html, August 2010. (Cited in Sections 2 and 4.7.)

[26] Rohit Dube. A comparison of scaling techniques for BGP. *Computer Communication Review*, 29(3):44–46, 1999. (Cited in Section 8.3.5.)

[27] European Advanced Networking Test Center. Cisco XR series service separation architecture tests. http://www.eantc.de/fileadmin/eantc/downloads/test_reports/2003-2005/EANTC-Summary-Report-Cisco-12kXR.FINAL.pdf, May 2005. (Cited in Section 8.2.1.)

[28] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and Sack TCP. *Computer Communication Review*, 26(3):5–21, July 1996. (Cited in Section 8.2.1.)

[29] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. The case for separating routing from routers. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, August 2004. (Cited in Section 8.3.5.)

[30] Anja Feldmann, Hongwei Kong, Olaf Maennel, and Alexander Tudor. Measuring BGP pass-through times. In *Passive and Active Measurement Workshop*, April 2004. (Cited in Section 8.2.2.)

[31] Jay Fenlason. Gnu gprof. http://sourceware.org/binutils/docs/gprof/index.html, 2003. (Cited in Section 5.1.)

[32] Ondrej Filip, Libor Forst, Pavel Machek, Martin Mares, and Ondrej Zajicek. The BIRD internet routing daemon. http://bird.network.cz. (Cited in Section 2.1.4.)

[33] Ondrej Filip, Pavel Machek, Martin Mares, and Ondrej Zajicek. BIRD Programmer's Documentation. http://bird.network.cz/?get_doc&f=prog.html. (Cited in Section 8.2.3.)

[34] Sally Floyd, Tom Henderson, and Andrei Gurtov. The NewReno Modification for TCP's Fast Recovery Algorithm. RFC 3782, April 2004. (Cited in Section 8.2.1.)

[35] Ruomei Gao, Constantinos Dovrolis, and Ellen W. Zegura. Interdomain ingress traffic engineering through optimized as-path prepending. In *In Proceedings of IFIP Networking*, 2005. (Cited in Section 7.1.1.)

[36] Joel Gottlieb, Albert Greenberg, Jennifer Rexford, and Jia Wang. Automated provisioning of BGP customers. *IEEE Network magazine*, Nov/Dec 2003. (Cited in Section 8.3.3.)

[37] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. In *ACM SIGCOMM Computer Communication Review*, October 2005. (Cited in Section 8.3.5.)

[38] Timothy G. Griffin and Brian J. Premore. An experimental analysis of BGP convergence time. In *9th International Conference on Network Protocols*, 2001. (Cited in Section 8.2.2.)

[39] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing extensible IP router software. In *2nd Symposium on Network Systems Design and Implementation (NSDI 2005)*, 2005. (Cited in Sections 2.1.4 and 6.5.2.)

[40] Andy Heffernan. Protection of BGP Sessions via the TCP MD5 Signature Option. RFC 2385, August 1998. (Cited in Section 8.2.1.)

[41] John A. Hoxmeier and Chris Dicesare. System response time and user satisfaction: An experimental study of browser-based applications. In *Proceedings of the Association of Information Systems Americas Conference*, pages 10–13, 2000. (Cited in Section 7.5.2.)

[42] Geoff Huston. BGP in 2009. RIPE 60 Meeting. http://www.ripe.net/ripe/meetings/ripe-60/presentations/Huston-BGP_in_2009.pdf, May 2009. (Cited in Section 8.3.5.)

[43] Gianluca Iannaccone, Chen-Nee Chuah, Supartik Bhattacharyya, and Christophe Diot. Feasibility of IP restoration in a tier-1 backbone. In *IEEE Networks Magazine, Special Issue on Protection, Restoration and Disaster Recovery*, March 2004. (Cited in Sections 1 and 8.2.1.)

[44] International Telecommunication Union. World telecommunication/ICT development report 2010 – monitoring the WSIS targets. http://www.itu.int/dms_pub/itu-d/opb/ind/D-IND-WTDR-2010-PDF-E.pdf. (Cited in Section 1.)

[45] Internet Systems Consortium. ISC domain survey. http://ftp.isc.org/www/survey/reports/2010/07/. (Cited in Section 1.)

[46] Kunihiro Ishiguro. GNU Zebra. http://www.zebra.org. (Cited in Section 2.1.4.)

[47] Paul Jakma, Vincent Jardin, Denis Ovsienko, Andrew Schorr, Hasso Tepper, Greg Troxel, and David Young. Quagga current releases. http://www.quagga.net/download/. (Cited in Section 8.2.3.)

[48] Paul Jakma, Vincent Jardin, Denis Ovsienko, Andrew Schorr, Hasso Tepper, Greg Troxel, and David Young. Quagga routing software suite. http://www.quagga.net. (Cited in Section 2.1.4.)

[49] Paul Jakma, Denis Ovsienko, and Joakim Tjernlund. Wishlist for any future redesign of the Zebra RIB. https://bugzilla.quagga.net/show_bug.cgi?id=431, 2007. (Cited in Section 5.)

[50] Eric Keller, Jennifer Rexford, and Jacobus van der Merwe. Seamless BGP migration with router grafting. In *7th USENIX Symposium on Networked Systems Design and Implementation*, 2010. (Cited in Sections 1.1 and 8.2.1.)

[51] D. Richard Kuhn. Sources of Failure in the Public Switched Telephone Network. *IEEE Computer*, 30:31–36, April 1997. (Cited in Section 1.)

[52] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed Internet routing convergence. In *Proceedings of the SIGCOMM 2000 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 175–187, Stockholm, Sweden, 2000. (Cited in Sections 8.2.2 and 8.2.3.)

[53] Craig Labovitz, Abha Ahuja, and Farnam Jahanian. Experimental study of internet stability and backbone failures. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, June 1999. (Cited in Section 1.)

[54] Craig Labovitz and Masaki Hirabaru. Routing technology final report (NCR-9318902). (Cited in Section 2.1.3.)

[55] Craig Labovitz, G. Robert Malan, and Farnam Jahanian. Internet Routing Instability. In *Proceedings of the SIGCOMM '97 Symposium on Communications Architectures and Protocols*, Cannes, France, 1997. (Cited in Sections 8.2.2 and 8.2.3.)

[56] Craig Labovitz, G. Robert Malan, and Farnam Jahanian. Origins of Internet Routing Instability. In *Proceedings of IEEE INFOCOM 1999*, New York, NY, 1999. (Cited in Sections 8.2.2 and 8.2.3.)

[57] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Lawrence G. Roberts, and Stephen Wolff. A brief history of the Internet. `http://www.isoc.org/internet/history/brief.shtml`. (Cited in Section 1.)

[58] John Levon. OProfile internals. `http://oprofile.sourceforge.net/doc/internals/index.html`, 2003. (Cited in Sections 5 and 5.1.)

[59] Olaf Maennel and Anja Feldmann. Realistic BGP traffic for test labs. In *Proceedings of the SIGCOMM '99 Symposium on Communications Architectures and Protocols*, August 2002. (Cited in Section 8.2.2.)

[60] David A. Maltz and Pravin Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proceedings of IEEE INFOCOM 1998*, March 1998. (Cited in Section 1.1.)

[61] David A. Maltz, Jibin Zhan, Geoffrey Xie, Hui Zhang, Gisli Hjalmtysson, Albert Greenberg, and Jennifer Rexford. Structure Preserving Anonymization of Router Configuration Data. In *Proceedings of ACM Internet Measurement Conference 2004*, Sicily, Italy, 2004. (Cited in Section 8.3.3.)

[62] David Meyer, Lixiz Zhang, and Kevin Fall. Report from the IAB Workshop on Routing and Addressing. RFC 4984, September 2007. (Cited in Section 8.3.5.)

[63] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68, pages 267–277. (Cited in Section 7.5.2.)

[64] John Moy. OSPF version 2. RFC 2328, April 1998. (Cited in Section 8.2.1.)

[65] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003. (Cited in Section 3.1.)

[66] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kıcıman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California at Berkeley, Berkeley, CA, March 2002. (Cited in Section 1.3.)

[67] Johan Petersson. What is `linux-gate.so.1`? `http://www.trilithium.com/johan/2005/08/linux-gate/`, 2005. (Cited in Section 5.1.)

[68] Marguerite Reardon. IP reliability. *Light Reading*, March 2003. (Cited in Sections 1, 8.2.1, and 1.)

[69] Yahov Rekhter, Tony Li, and Susan Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006. (Cited in Sections 2.1.6, 3.2.2, 3.2.4, 3.2.5, 6.2.2, 6.3.1, 7.1.2, 7.2.2, and 8.3.2.)

[70] Yakov Rekhter, Robert G Moskowitz, Daniel Karrenberg, Geert Jan de Groot, and Eliot Lear. Address allocation for private internets. RFC 1918, February 1996. (Cited in Section 7.1.1.)

[71] Alvaro Retana and Russ White. BGP Custom Decision Process. Internet Draft. http://tools.ietf.org/html/draft-retana-bgp-custom-decision-00, October 2002. (Cited in Section 8.3.2.)

[72] Carsten Rossenhövel. 40-Gig Router Test Results. *Light Reading*, November 2004. (Cited in Sections 8.2.1 and 8.2.2.)

[73] Srihari R. Sangle, Yakov Rekhter, Rex Fernando, John G. Scudder, and Enke Chen. Graceful restart mechanism for BGP. RFC 4724, January 2007. (Cited in Sections 3.4, 4, 4.1.2, 4.2.1, 4.3, 6.3, 6.4, and 7.)

[74] Panagiotis Sebos, Jennifer Yates, Guangzhi Li, Dan Rubenstein, and Monica Lazer. An integrated IP/optical approach for efficient access router failure recovery. In *Optical Fiber Communications Conference*. IEEE, 2003. (Cited in Sections 1.1 and 8.2.1.)

[75] Ian Shields. Learn Linux, 101: Manage shared libraries. http://www.ibm.com/developerworks/linux/library/l-lpic1-v3-102-3/, 2010. (Cited in Section 5.1.)

[76] Renata Teixeira, Aman Shaikh, Tim Griffin, and Jennifer Rexford. Dynamics of hot-potato routing in IP networks. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems*, 2004. (Cited in Section 8.2.2.)

[77] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Conn., 1983. (Cited in Section 8.2.3.)

[78] Edward R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, Conn., 1990. (Cited in Section 8.2.3.)

[79] Edward R. Tufte. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, Cheshire, Conn., 1997. (Cited in Section 8.2.3.)

[80] Edward R. Tufte. *Beautiful Evidence*. Graphics Press, Cheshire, Conn., 2006. (Cited in Section 8.2.3.)

[81] University of Oregon Advanced Network Technology Center. University of Oregon Route Views Project. http://www.routeviews.org/. (Cited in Sections 2.1.3 and 8.2.2.)

[82] J. Van der Merwe, A. Cepleanu, K. D'Souza, B. Freeman, A. Greenberg, D. Knight, R. McMillan, D. Moloney, J. Mulligan, H. Nguyen, M. Nguyen, A. Ramarajan, S. Saad, M. Satterlee, T. Spencer, D. Toll, and S. Zelingher. Dynamic connectivity management with an intelligent route service control point. In *Proceedings of the 2006 SIGCOMM Workshop on Internet Network Management*, INM '06, pages 29–34. ACM, 2006. (Cited in Section 8.3.5.)

[83] Yi Wang, Eric Keller, Brian Biskeborn, Jacobus van der Merwe, and Jennifer Rexford. Virtual routers on the move: Live router migration as a network-management primitive. In *Proceedings of the SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, August 2008. (Cited in Sections 1.1 and 8.2.1.)

[84] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association. (Cited in Section 2.1.2.)

[85] Qiang Wu, Yong Liao, Tilman Wolf, and Lixin Gao. Benchmarking BGP routers. In *Proc. of IEEE International Symposium on Workload Characterization (IISWC)*, Boston, MA, September 2007. (Cited in Sections 2.1.3, 8.2.2, 8.2.3, 8.3.4, and 8.3.5.)

[86] XORP, Inc. XORP BGP routing daemon, version 1.6. `http://xorp.org/releases/1.6/docs/bgp/bgp.pdf`, 2009. (Cited in Sections 6.5.2 and 8.2.3.)

[87] Beichuan Zhang, Vamsi Kambhampati, Mohit Lad, Daniel Massey, and Lizia Zhang. Identifying BGP routing table transfers. In *ACM SIGCOMM 2005 Workshop on Mining Network Data*, pages 213–218, Philadelphia, Pennsylvania, 2005. (Cited in Section 8.2.2.)

Colophon

This document was prepared with GNU Emacs, and set using XeLaTeX. System charts were produced using the matplotlib plotting package, with text rendered by LaTeX. Various other illustrations were created with Dia, GIMP, Graphviz, Inkscape, matplotlib, and OpenOffice. The text is set primarily in TeX Gyre Pagella, with the Hindi text in the dedication set in Kalimati, and the stylized signature in the acknowledgements set in Zapfino.