Building Bug-Tolerant Routers with Virtualization

Matthew Caesar and Jennifer Rexford
Princeton University

Abstract

Implementation bugs are a highly critical problem in wide-area networks. The software running on core routers is subject to vulnerabilities, coding mistakes, and misconfiguration. Unfortunately, these problems are often found *after* deployment in live networks, where they lead to outages, make networks prone to attack, and involve a challenging process to localize and debug. In this work, we propose a bug-tolerant router that runs multiple diverse copies of router software in parallel, such that each copy is unlikely to fail at the same time as the others. Diversity is achieved by varying the ordering and timing of routing messages, running different routing protocols, running code written by different implementers, etc. Because each copy is different, each copy will likely have a different output during an error, and hence a simple voting procedure is then used to decide which copy's output will "drive" packet forwarding and control-plane communication with other routers. In this paper we motivate our design, describe some design decisions and tradeoffs, and then conclude with a description of our ongoing work in building a prototype of this architecture.

1. Introduction

Much of the Internet's functionality is implemented in *software* running on routers. Internet routers typically run an operating system (e.g., Cisco IOS or JunOS) along with a suite of protocol daemons to perform routing and administration functions. Since this software is written by human programmers, it sometimes contains mistakes, or *implementation bugs*. The fact that these bugs can produce incorrect and unpredictable behavior, coupled with the mission-critical nature of core Internet routers, can produce disastrous results. Worse still, ISPs often run the same vendor equipment and protocols network-wide, increasing the probability that a bug causes simultaneous failures or a network-wide crash. Unfortunately, bugs are often discovered only *after* they cause major outages. Operators must wait for vendors to implement and release a patch for the bug, or find an intermediate work around on their own, leaving their networks vulnerable in the meantime.

As an example, some early BGP implementations assumed that the AS-path attribute would never have more than 100 hops. Incidents occurred in which routers received a route with a longer AS path, causing the router software access memory beyond the previously-allocated space, leading the software to crash. Worse yet, correctly-functioning routers throughout the Internet propagated the unusual route to a large number of buggy routers, leading hundreds of routers in the Internet to crash at nearly the same time. Even worse still, upon restarting, these buggy routers received the offending announcement a second time, and crashed again. While awaiting a software patch to fix the bug, network operators configured their routers to filter routes with long AS paths to avoid propagating routes that might cause their neighbors to crash.

Unfortunately, this was not an isolated incident. The high complexity and distributed nature of Internet routing has led to a rich variety and numerous high-profile bugs and outages [1, 2, 3, 4, 5, 6]. These bugs are notoriously difficult to reproduce and localize, since they may violate protocol invariants, be "heisenbugs" that change characteristics or disappear when investigated, or arise from interdependencies across several distributed routers. Worse still, these bugs may be *vulnerabilities* that remote attackers can exploit to compromise and control networks. Finally, we believe router software bugs will become an even more critical problem in the future, as router vendors start to open up their operating systems to third-party developers [7, 8], and as networks are deployed in developing regions with fewer resources to debug problems and upgrade software [9], and as other preventable sources of outages become less common (due to better protocols/practices for planned maintenance, better configuration automation and checking, etc.) [10].

As part of an ongoing study router software bugs, we manually classified the bugs listed in the Bugzilla bug repository web sites for the Quagga [11] and XORP [12] open-source routers, as summarized in Table 1. Although some bugs cause the router to crash, others allow the router to continue running while producing incorrect results, making the problems potentially very hard to detect. To complement our Bugzilla analysis, we are in the process of applying a variety of static and dynamic analysis tools to open-source router software to detect and characterize previously unreported bugs for these routers. We also plan to conduct black-box testing of commercial routers, and experiment with running commercial router software directly on a PC [13, 14].

In this paper we argue that instead of (or perhaps in addition to) finding ways to localize router bugs and quickly recover from them, router software should be architected from first principles with buggy code in mind. We propose the design of a *bug-tolerant* router that significantly reduces the likelihood of a software error affecting the network. To other routers, a bug-tolerant router appears like any other router: it runs the same protocols and forwards packets in the same way. However, internally, our bug-tolerant router consists

Table 1: Breakdown of bugs from bugzilla.quagga.net by their effect (whether they cause a router to crash, not crash but behave wrong, or not affect router behavior).

Bug categ.	Bug type	# of bugs in Quagga	# of bugs in XORP
Cause router to crash	*Seg-fault*	3	8
	Memory leak	2	9
	Failed assert	14	6
	Outputs vty error	2	0
	Outputs log error	5	0
	Freezes/deadlocks	6	5
	Unspecified crashes	9	2
	Subtotal:	**41 (43%)**	**30 (49%)**
Wrong behavior	*Sends incorrect route*	25	16
	Security vuln.	2	1
	Incorrectly parses config	6	4
	Performance bugs	0	1
	Subtotal:	**33 (35%)**	**22 (36%)**
No effect	*Compile errors*	10	5
	Missing command/docs	10	4
	Subtotal:	**20 (21%)**	**9 (15%)**
	Totals:	**94**	**61**

of several *virtual routers* running in parallel. Each of these virtual-router instances is made different from the others, by modifying their execution environment (e.g., by reordering the routing updates they receive, by changing their configuration, or by modifying their layout in memory) or by modifying their internal structure (e.g., by running router code implemented by different programmers). This diversity decreases the likelihood that multiple router copies will simultaneously fail. To allow multiple virtual routers interact with the outside environment, *voting* is used on their outputs to decide which routes to use to forward packets, and which routes to export to neighbors.

Running multiple versions of a piece of code is known as *Software and Data Diversity* (SDD) [15] and has been widely applied in non-networked programs that require very high availability. Despite recent application of SDD to other kinds of computer systems [16, 17, 18], and despite increasing use of technologies such as HSRP [19] and VRRP [20] that enable rapid failover from one *physical* router to another, SDD has not been widely explored in the context of routing software. Although the distributed operation and tight performance requirements of routing protocols introduce challenges, routing software offers several unique opportunities to modify and customize SDD techniques: the modular construction of routers, their well-structured input/output, and their minimal dependence on history. Finally, we acknowledge the long-standing debate in the software-engineering community over whether it is possible to completely prevent software errors. We believe unforeseen interactions across protocols, the potential to misinterpret RFCs, the increasing functionality of Internet routing, and the ossification of legacy code and protocols in the Internet will make router errors a "fact-of-life" for the foreseeable future and we proceed under that assumption.

To the best of our knowledge, our work represents the first attempt to provide the strong foundations and principles of SDD towards building highly-available routing software. That said, our work can leverage and extend several existing technologies, including virtual networks [21, 22, 23], virtual routers [24, 25], virtual machine technologies [13, 26], open-source routers [11, 12, 27, 28] and traditional applications of software and data diversity [15]. In the following sections, we first give an overview of challenges and opportunities in applying software diversity to data networks (Section 2). We then describe the architecture of a bug-tolerant router, including several design decisions and tradeoffs (Section 3). We then conclude by describing our future plans to implement and evaluate our design (Section 4).

2. Improving routing software reliability

Routers run routing protocols that exchange reachability information to compute paths that reach destination address blocks. The protocols form the *control plane* that consists of routing processes, or *daemons*, that select and announce routes and populate their own Routing Information Bases (RIBs) that store the routes learned from their neighbors. The "best" routes in each RIB are combined to construct a single Forwarding Information Base (FIB) that the *data plane* uses to forward each packet to the next hop in its journey. Most routers have a clear separation between the control and data planes, with the control plane running in software and the data plane running in the operating system or in dedicated hardware. In this section, we describe some of the unique challenges in building reliable router software, the opportunities to apply customized SDD techniques to this environment.

2.1 Reliability challenges for routing software

Traditional SDD principles cannot be directly applied to network routing, as routing systems have several unique properties and requirements that must be taken into account (we will later address these in Section 3).

Fast reaction: Data networks are increasingly called upon to forward traffic with demanding performance requirements. This has led to substantial work on building routers that can react quickly to network changes. Router bugs can interfere with this goal. Before a router can forward packets after a crash, the crash must be detected (which may require manual intervention), the router must reload its routing table from all of its neighbors, and the network-wide routing protocol must reconverge. Worse still, if the router does not crash but produces incorrect output, long-term or perhaps even persistent outages may be triggered.

Large configuration space: Much of an Internet router's operation can be customized via *configuration languages*. These languages allow a network operator to balance traffic load across links, filter malicious traffic, and prefer more desirable routes. The highly flexible nature of routing protocols leads to a vast number of execution paths. Unfortunately, these execution paths are only executed under certain configurations and hence are extremely difficult to fully test and debug.

Concurrency: By their very nature, network components coordinate via distributed mechanisms. Hence in order to perform an operation, such as setting up a path or forwarding a packet, multiple routers must be involved, and a fault

at any intermediate router may interfere with the packet's delivery. In addition, to achieve high performance, parallel techniques are often used in architecture of individual routers. In software, multiple processes/threads are used to simultaneously perform multiple operations. Unfortunately, building and programming concurrent systems has been a long-standing research challenge [30]. This increases the likelihood of router bugs, and also increases the difficulty of localizing the problem.

2.2 SDD opportunities for routing software

While network routing has several differences from environments in which SDD has been previously applied, there are also several aspects of network routing that may make it particularly well-suited to *customized* versions of SDD:

Small dependence on past history: Most typically, the computation a router performs only depends on the set of routes currently advertised by its neighbors. This makes it easy to detect and repair bugs "after-the-fact": if a router advertises incorrect information due to a bug, it can later simply send a routing update to "overwrite" the incorrect information, which means bug detection does not have to take place instantaneously. This also simplifies creation/migration/deletion of existing router instances. For example, cloning an existing router instance need not involve storing and replaying every routing update received by the original router, but may instead be done by only replaying the set of currently advertised routes.

Modular construction and well-defined interfaces: Routers are often composed of multiple self-contained units which can be easily decoupled from each other. The interfaces between these modules form a natural place where their outputs may be reconciled or voted between. For example, separation often occurs along protocol boundaries (XORP *modules* and Quagga *daemons*) as well as functional boundaries (XORP/Quagga control-plane vs. Click/kernel data plane). This allows SDD to be applied separately to each component. For example, if the control plane encounters a bug, it can be restarted without affecting the data plane. Also, several control planes may coordinate (perhaps by voting) to populate a single data plane.

Well-structured input and output: The operation of routing protocols is well-defined by specifications such as *RFCs* [31]. This has several benefits. First, this means that alternate implementations of the protocol can be built to these specifications (N-version programming [15]). This has lead to multiple open-source implementations of routing protocols existing today which are expected to be interoperable. Second, protocol behavior can sometimes be captured by models [32, 33], which provide formal notions of correctness. These models can be used to check correctness and detect faults that don't cause routers to crash. Moreover, several tools exist which take protocol specifications and generation functional code, which can be used to generate additional versions. Finally, a router's computation can be expected to complete after a certain period of time, which may simplify detection of bugs that prevent termination (hangs/looping).

Multiple ways to achieve the same objective: The flexibility in building and configuring networks often allows multiple ways to achieve the same goal. First, multiple protocols exist that perform the same operation (e.g., OSPF and IS-IS). Second, the flexibility in configuration languages lead to a wide space of semantically-equivalent configurations for routers. While these configurations should lead to the same outputs, each one may trigger different bugs and failure modes. Finally, routes to different destinations are often independent, and delays within reason affect only timing and not the final answer. Hence, artificially reordering or delaying certain updates will not change the steady-state outcome of route selection.

Can survive brief outages: Decades of dealing with faulty networks has led network architects and networked application designers to plan for short outages and variable delays in their code. Routing protocols such as BGP and OSPF retransmit and probe for liveness. Many routers can survive a short control-plane outage without interrupting forwarding in the data plane. Even if a short data-plane outage does occur, end-to-end Internet protocols retransmit or otherwise gracefully deal with loss. This allows us to use techniques that quickly *recover* from bugs, as opposed to more heavyweight *prevention*-based methods.

3. System architecture

We present our system architecture in three phases. First, we describe the architecture of a *bug-tolerant* router that leverages virtualization to run multiple diverse instances of router software in parallel. Then, we describe how to decrease the likelihood that these virtual routers encounter the same software bugs at the same time. Last, we consider extensions that run multiple routing daemons, or entire virtual networks, in parallel to increase the opportunities for software diversity.

3.1 Voting among multiple virtual routers

In this section we propose the design of a *bug-tolerant* router. While it is possible to apply SDD to improve bug-resilience in both the control and data planes, we focus primarily on improving the resilience of the control plane. There are two key reasons for this. First, we would like our architecture to be implementable as a software upgrade to existing routers, without requiring extensions or redesigns of router hardware. Secondly, the packet lookup and forwarding algorithms used in the data plane are typically simpler than the distributed protocols used in the control plane. This fact, coupled with the widespread success of hardware modeling and validation, substantially reduces the number of bugs in the data plane.

Our design consists of a replicated control plane connected to a single data plane, as shown in Figure 1. In particular, we create multiple diverse, functionally equivalent copies of control-plane software running in parallel. Since each copy is diverse (different layout in memory, different

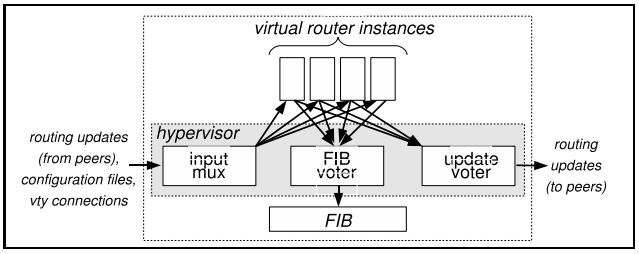

Figure 1: Architecture of a bug-tolerant router.

ordering/timing of updates, different code bases, etc.), the multiple copies are unlikely to fail at the same time. Collectively, these multiple instances of the control plane must work together to appear as a single control plane, to the data plane as well as external routers. This is achieved through the use of a *hypervisor*. The hypervisor is responsible for monitoring the health of virtual router instances, rebooting and re-syncing failed instances, diversifying their inputs, and mediating between their outputs.

The hypervisor is composed of three components to perform these functions: output mediation is performed with a *voter*, replicating and diversifying inputs is done with a *mux*, and maintaining and monitoring health of the virtual routers is done with a *controller*.

Mediating output with a *voter*: In our design, multiple control plane instances need to collectively maintain a single FIB. This is done with a *voter*, which accepts a route from each instance, and decides which one to populate in the FIB. One simple way to do this would be to perform majority voting: if an instance begins to perform incorrectly, its output will differ form the outputs of the other instances. However, this requires that all virtual routers respond with their outputs before the result is sent to the FIB. This slows reaction (e.g., to failures) to the speed of the slowest instance. Given some approaches to increasing diversity may substantially slow computation time, such as compiling with optimizations disabled, or instrumenting code with correctness/model-checking tools, this approach may be undesirable. Worse still, voting cannot be applied immediately, as transient routing decisions may differ across instances, and hence voting can only be done after instances reach their steady-state outputs.

Hence, our voter instead works by letting a single instance "drive" the route selection. That is, one virtual router is assigned as the *master*, and the others as *standbys*. If the master begins producing outputs that differ from the standbys, or crashes, then one of the standbys is chosen as the master. However, the downside of this approach is that an "incorrect" route may be written to the FIB. To ensure these routes are quickly overwritten with correct entries, when failing over from a master to a standby, the FIB entries that differ from the standby's table are immediately overwritten. To reduce failure probability even further, our architecture allows a hybrid approach, here voting is performed across the first k of N virtual routers to finish computation. In addition, this voting must only be performed after routes reach steady state. To ensure that happens, all router instances are periodically cloned (using copy-on-write techniques to speed reaction) at certain fixed intervals and allowed to converge, and voting takes place only over the clones. In addition to computing which routes are sent to the FIB, we use a similar approach to decide which routing updates are sent to neighboring routers.

Replicating and diversifying input with a *mux*: Internet routers form *peering sessions* with their neighbors, on which they receive routing updates. These routing updates are then forwarded to the daemon responsible for processing the corresponding protocol. Since our bug-tolerant router must appear as a single router to its neighbors, we need some way for routing updates to be transparently "multicast" to each of the virtual router instances. In addition, configuration files and operator vty (terminal) input need to be sent to each virtual router, to ensure configuration changes are applied to each instance. These functions are also done with an *input mux*. The input mux is also responsible for increasing diversity of these inputs, to increase the chance that different instances fail at different times, as discussed in more detail in Section 3.2.

Maintaining and monitoring health with a *controller*: Since bugs may cause router instances to enter incorrect states, or crash, our architecture needs to ensure these failure conditions can be detected and repaired. This is handled by the *controller*. The controller interfaces with the voter and mux modules, and monitors the inputs and outputs of routers. Instances that crash, or leak excessive memory, or appear to be infinite-looping, or repeatedly give incorrect answers, are assumed to be behaving improperly and restarted. In traditional software, determining whether a piece of code is running correctly, or whether computation will terminate is a hard problem [34]. However, a routing protocol's computation is well-defined, and can be reliably assumed to return after a reasonable amount of time has passed. After restarting an instance, its state may be refreshed by cloning the memory segment of another virtual process (if the two instances share the same binary) or by reloading the routing information (if they don't).

3.2 Increasing diversity among virtual routers

In order to maximize the benefit of software redundancy, it is imperative that the individual virtual routers are as diverse as possible. Increasing software diversity has been a long-standing challenge in the software-engineering literature [15, 16, 17, 18]. However, the unique features of routing protocols mentioned in Section 2.2 allow us to take several unique approaches towards increasing diversity[1]:

Different code bases: The instances could each be developed by different implementers. For example, Quagga [11], XORP [12] and OpenBGPd [27] could be run in parallel.

Different software versions: The instances could be different versions of the same router. For example, Quagga v0.96,

[1]To be functionally equivalent, sources of non-determinism such as age-based tie-breaking and non-deterministic MED must be disabled. This is often done by operators anyway because they lead to unpredictable output.

Quagga v0.97, and Quagga v0.99 could be run in parallel.

Different configurations: The instances could be the same code base, but be configured in different yet semantically-equivalent ways. That is, unspecified preferences between routes could be randomized, or multiple configuration files could be used, each written by a different human operator.

Different messages timing and ordering: The timings and orderings of update messages received from peers could be randomized, and some update messages may be dropped completely (for example, by forcibly withdrawing a route).

Different subsets of address blocks: The routers may be identical, but may run for different overlapping subsets of the network. For example, we may run one Quagga instance that ignores routes not between 0.0.0.0/8 to 160.0.0.0/8, and a second Quagga instance that ignores routes not between 100.0.0.0/8 to 255.0.0.0/8. In this case, the arbitrator must be configured to perform voting for a particular route only across instances that handle that route.

Different execution environments: The execution environment of each instance may be modified by the operating system. For example, the layout in memory, or the ordering of threads/process execution may be randomized.

In addition to comparing results across multiple virtual routers in parallel, routing decisions could be validated against formal models of the route-selection process. For example, models can be used to compute the "best routes" each router should ultimately pick, based on the topology, routing configuration, and externally-learned routes [32]. Similarly, network operators can specify certain invariants they expect to hold in steady state, such as exporting the same set of IP prefixes via each BGP session with a particular neighboring network. Applying these checks to the actual RIBs (computed by the routing protocols in a distributed and dynamic fashion) can detect a variety of subtle bugs in the routing software or routing configuration.

3.3 Diversity at the process and network level

The principle of SDD works by running multiple instances of a piece of code in parallel. The modular nature of networks presents the opportunity to apply SDD at multiple locations and at varying levels of granularity. Our discussion so far has implicitly assumed *router-level* redundancy, i.e., that the code running in parallel would be the operating system and entire protocol suite of a single router. However, our architecture also enables two additional forms of redundancy: *network-level* redundancy, where entire virtual networks are run in parallel, and *process-level* redundancy, where individual routing processes and threads running inside a router implementation are replicated. While any one of these approaches may be used in isolation, we believe building bug-tolerant networks should take advantage of and use all three if possible, since they are non-conflicting and hence may be done simultaneously in our architecture.

Network-level redundancy: Instead of running individual routers in parallel, ensembles of routers may collectively run multiple entire *virtual networks* in parallel. In this approach, the outputs of a single router are not merged into a single FIB, or as a single RIB advertised to its neighbors. Instead, routers maintain a separate FIB for each virtual network, and voting is used at border routers to decide which virtual network will be used to forward packets. Data packets arriving at a border router are encapsulated with an identifier of that virtual network. This approach offers several advantages in increasing software diversity:

Diverse routing protocols: Different virtual networks could employ multiple independent network-wide configurations. For example, one virtual network may run OSPF, while another may run IS-IS, which may be difficult (if not impossible) to do with just router-level redundancy.

Faster convergence: Running multiple virtual networks may lead to faster routing-protocol convergence, since individual physical routers do not have to wait for their internal virtual routers to vote and agree on a result before forwarding an update.

However, deployment may become more challenging, since this approach relies on network-wide deployment. That said, it is possible to use tunnels to traverse routers that are not instrumented with our virtualization technology. In addition, the network-wide approach may introduce more control overhead, as updates must be separately exchanged for each virtual network. In general, the routing-protocol traffic is a small fraction of the total load in most high-speed networks, making this a relatively minor concern. Still, it should be possible to reduce overhead by suppressing redundant update messages, or just transmitting deltas of their contents.

Process-level redundancy: In addition, SDD may be applied at a finer granularity, by creating redundant yet diverse executions of individual router processes or threads. For example, rather than voting among virtual routers just before installing forwarding-table entries in the FIB, voting could be performed amongst multiple *routing daemons* to construct a single *RIB*. This approach offers several advantages:

Lightweight operation: Cloning and restarting only individual processes or threads may speed reaction, and reduce memory usage and computational requirements.

Finer-grained control: During times of load, only mission-critical components may be cloned to reduce resource usage. Also, voting could be more tightly integrated into processing, for example, by voting at the end of the decision process.

However, code development may become more challenging, since this approach relies on knowing which parts of the code are functionally equivalent, and under what conditions this holds true. Unlike "router-wide" or "network-wide" approaches, the execution environment is often unaware of internal interfaces between blocks of router code, and which may be replicated in a functionally equivalent way. That said, router code is often designed in a modular fashion, being composed of well-isolated processes and daemons. For example, in XORP multiple *rib modules*, and in Quagga multiple *bgpd daemons*, may be run in parallel.

Modifying router software to conform to a common API

would enable replication and composition of modules from different code bases. This API would define a collection of modules within a router, how they interact, and how outputs from multiple instances of each can be combined with voters. Though practically challenging, there are some promising initial signs in this direction. For example, common APIs like *libvirt* are beginning to emerge in the area of host virtualization [35]. In addition, existing routers already have a clear separation between the control and data planes, and some general APIs exist for populating the data plane [36]. Also, the Quagga routing software [11] has its own open API that defines how individual routing processes communicate in the context of a single router. In addition, having a common "router hypervisor" would be useful for supporting other advanced features, such as the migration of virtual routers from one physical platform to another [22]. Commercial router vendors may be understandably reticent to interface their router software with other vendors, but it is still possible for a single vendor to benefit from these techniques, for example by running multiple BGP processes within a single instance of Cisco IOS. In addition, the move toward supporting third-party software on commercial routers [7, 8] relies on having clearly-specified APIs.

4. Conclusions

In this paper, we described how to improve resilience of networks to bugs by applying Software and Data Diversity (SDD) techniques to router design. We presented an early architecture which may be tenable in practice. Our approach is amenable to incremental deployment, and can be run only within a single ISP, a single subnet, or even a single router.

We are currently implementing and deploying a prototype of our system in the context of the VINI [21] testbed. In this environment, we first plan to taxonomize and study the behavior of bugs in networked software. Secondly, we plan to develop and evaluate an efficient hypervisor implementation, and study the behavior of different voting algorithms. We also plan to study other software engineering techniques to avoid bugs, for example by incorporating *rollback* operations, or by replaying/reordering state changes to localize the bug. Finally, we plan to extend our testbed to study bugs in commercial, closed-source routers, by leveraging freely available tools to run Cisco/Juniper operating system code directly on Linux workstations [13, 14].

5. References

[1] T. Akin, "Cisco router forensics," in *Blackhat Briefings*, July 2002. `www.blackhat.com/presentations/bh-usa-02/bh-us-02-akin-cisco/bh-us-02-akin-cisco.ppt`.

[2] J. Evers, "Trio of Cisco flaws may threaten networks," in *CNET News*, January 2007.

[3] B. Brenner, "Cisco IOS flaw prompts symantec to raise threat level," in *Information Security Magazine*, September 2005.

[4] W. Knight, "Router bug threatens 'Internet backbone'," in *New Scientist Magazine*, July 2003.

[5] J. Duffy, "BGP bug bites Juniper software," in *Network World*, December 2007.

[6] P. Roberts, "Cisco tries to quash vulnerability talk at Black Hat," in *eWEEK (Ziff Davis Inc.)*, July 2005.

[7] "Cisco opening up IOS," in *Network World*, December 2007.

[8] "Juniper networks delivers industry-first platform for customer and partner application development on carrier-class network operating system," in *Juniper Networks, Inc. (press release)*, December 2007.

[9] R. Tongia, "Connectivity and the digital divide: Technology, policy, and design tradeoffs for developing regions," in *Telecommunications Policy Research Conference*, September 2006.

[10] A. Kuatse, R. Teixeira, and M. Meulle, "Characterizing network events and their impact on routing," in *Proc. CoNEXT*, December 2007.

[11] "Quagga software routing suite," `www.quagga.net`.

[12] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, "Designing extensible IP router software," in *Proc. NSDI*, May 2005.

[13] "Cisco 7200 simulator," (software to run Cisco IOS images on desktop PCs) `www.ipflow.utc.fr/index.php/Cisco_7200_Simulator`.

[14] "Olive," (software to run Juniper OS images on desktop PCs) `juniper.cluepon.net/index.php/Olive`.

[15] L. Chen and A. Avizienis, "N-version programming: A fault tolerance approach to reliability of software operation," in *Proc. Fault-Tolerant Computing Symposium*, June 1978.

[16] E. Berger and B. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," in *Proc. Programming Languages Design and Implementation*, June 2006.

[17] R. Rodrigues, M. Castro, and B. Liskov, "Base: Using abstraction to improve fault tolerance," in *Proc. ACM SOSP*, October 2001.

[18] Y. Zhou, D. Marinov, W. Sanders, C. Zilles, M. d'Amorim, S. Lauterburg, and R. Lefever, "Delta execution for software reliability," in *Proc. Hot Topics in Dependibility*, June 2007.

[19] T. Li, B. Cole, P. Morton, and D. Li, "Cisco hot standby router protocol (HSRP)." RFC 2281, March 1998.

[20] R. Hinden, "Virtual router redundancy protocol (VRRP)." RFC 3768, April 2004.

[21] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, "In VINI veritas: Realistic and controlled network experimentation," in *Proc. ACM SIGCOMM*, August 2006.

[22] Y. Wang, J. van der Merwe, and J. Rexford, "VROOM: Virtual ROuters On the Move," in *Proc. HotNets*, November 2007.

[23] N. Feamster, L. Gao, and J. Rexford, "How to lease the Internet in your spare time," in *ACM Computer Communication Review*, January 2007.

[24] D. McPherson, C. Parker, R. Hartani, D. Ward, P. Agarwal, S. Poretsky, and D. O'Leary, "Panel abstract: Core network design and vendor prophecies," February 2006.

[25] M. Kolon, "Intelligent logical router service," in *White Paper (Juniper Networks, Inc.)*, October 2004.

[26] "Xen," (hypervisor software), `www.xen.org/`.

[27] "Openbgpd," `www.openbgpd.org`.

[28] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The Click modular router," in *ACM Trans. Comp. Sys.*, August 2000.

[29] R. Alimi, Y. Wang, and Y. R. Yang, "Shadow configuration as a network management primitive," in *Proc. ACM SIGCOMM*, August 2008.

[30] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The landscape of parallel computing research: A view from Berkeley," Tech. Rep. EECS-2006-183, UC Berkeley, December 2006.

[31] J. Moy, "OSPF Version 2." RFC 2328, April 1998.

[32] N. Feamster and J. Rexford, "Network-wide prediction of BGP routes," in *IEEE/ACM Trans. Networking*, April 2007.

[33] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *Proc. NSDI*, May 2005.

[34] M. Sipser in *Introduction to the Theory of Computation, Section 4.1 (The Halting Problem)*, PWS Publishing, 1997.

[35] "The virtualization API." `libvirt.org/`.

[36] "IETF forwarding and control element separation (ForCES)," (software) `www.ietf.org/html.charters/forces-charter.html`.