Towards Interactive Debugging for ISP Networks

Chia-Chi Lin[†], Matthew Caesar[†], Jacobus Van der Merwe[§]
[†]University of Illinois at Urbana-Champaign
[§]AT&T Labs – Research

ABSTRACT

The extreme complexity of Internet software leads to a rich variety of hard-to-isolate failure modes and anomalies. Research on debugging modern networked systems has thus far focused on "removing the human from the loop" by automatically detecting problems that violate predefined conditions. Here, we argue for a very different approach. Namely, we take the position that manual labor is a necessary evil of debugging problems in networked systems, but that this process would be vastly simpler with *in-network support* for debugging. We propose a network-layer substrate for *interactive debugging* that allows for tight controls on network execution, to provide reproducibility and performance isolation of the live network in highly distributed and dynamic environments.

1. INTRODUCTION

The Internet is the most complex distributed software infrastructure ever created. Unfortunately, the extreme complexity of this software makes it prone to *defects* introduced by human error. Network service and equipment developers introduce *bugs* and *vulnerabilities* into network software, and operators *misconfigure* equipment with devastating effects and high-profile outages [5, 16], and with high costs [2].

Conventional wisdom is that debugging large scale systems should be done with *fully automated* techniques, for example by checking the behavior of the system against a *model* that describes how the system should behave (which may place constraints on statistical behavior [12] or deterministic rules on orderings or contents of messages [6]). Unfortunately, while some problems can be identified this way, fully modeling the logic of a complex protocol requires complex models (increasing probability of bugs in the model), instrumenting protocols with models is often not possible due to lack of internal boundaries within modules, ossification of legacy code, privacy issues (e.g., when the developer and debugger belong to different institutions) and scaling issues of revealing internal state. In addition, model-based techniques typically "detect" problems instead of indicating how the problem can be repaired, still requiring the troubleshooter to understand the complex dynamics and state of the underlying network to design a solution. Finally, while automated techniques have been developed to localize memory faults [1] and avoid concurrency bugs [13], the larger class of *logical* or *semantic* errors seems to fundamentally require human knowledge to solve.

In this work, we take the position that manual labor by domain experts is a necessary evil of debugging problems in networked software, but that this process would be vastly simpler with network-level architectural support for debugging. As an early first step in this direction, we propose an architecture for *interactive debugging* of networked software. While tools such as *gdb* exist to isolate problems on a single machine, today we lack a practical gdb-style tool for interactive debugging of modern network software.

To clarify our goals, we give a simple example of how our proposed environment might be used in practice. Imagine that a network operator of an large ISP receives a phone call from a complaining customer, indicating that their Internet connectivity appears to fail at random times, with a large outage occurring the previous night. To diagnose the problem, the network operator creates a cloned instance of the live network based on the last checkpoint before the outage occurs, and places a breakpoint that is triggered when the route between the customer and a remote network becomes unavailable. The operator then runs the cloned instance. When the route is withdrawn, the cloned instance is *paused*, and the operator may then query and print to view the current state of the network. The operator notices that the path changed because a withdrawal message was received from a neighboring router, and hence the operator places a breakpoint triggered on the creation of that update. By repeating this process, the operator localizes the problem to a single router, which repeatedly sends updates, even though it is receiving no new external events. The operator forms a hypothesis that the router's software is faulty, and to test the hypothesis rolls back the router's software to an earlier version. To test the workaround, the operator plays the cloned network forward at high speed (injecting synthetic external updates) to see if any oscillations will take place over the next several days. Since the problem does not reoccur, the cloned network is merged back into the live network.

The ability to perform these functions would drastically simplify the ability to interactively troubleshoot network software. However, modern network infrastructures do not provide the underlying primitives and mechanisms to perform these operations. To ad-

dress this, we propose three key network-layer primitives to simplify interactive network debugging:

Isolation of the operational network: The mission-critical nature of modern networked systems coupled with their need to react to events in real-time means taking parts of the network offline for debugging purposes is not tenable. Hence, we propose a strawman architecture for a network software hosting infrastructure, that leverages virtualization technologies to provide strong resource isolation of the live network. Operators may *clone* the live network into a separate instance running atop our infrastructure, test workarounds and investigate faults within the contained environment, and *merge* the instance back into live network once the problem has been fixed.

Reproducibility of network execution: The ability to replicate the fault is improved if the system can replicate the precise sequence of steps that led to a bug being triggered. However, reproducing execution of networked software, which by its nature executes asynchronously and nondeterministically without centralized control, introduces some challenges. To address this, we propose several algorithms (leveraging previous work on distributed semaphores and checkpointing). These algorithms instill an ordering over message propagation in the live network, that can be reproduced exactly when running within a cloned instance, simplifying the ability for operators to reproduce rarely encountered exceptions.

Interactive stepping through execution: The tremendous load, scale, and rates of change of modern networks makes it hard for a human troubleshooter to build an understanding of the entire system.Debugging such a system becomes much easier if the troubleshooter has time to investigate and manipulate state, and slowly step through operation of individual messages. To achieve this in distributed networks, we develop a collection of protocols that provides tight control over execution of network software (e.g., allowing the operator to interactively step through distributed execution), while preserving the distributed characteristics of asynchronous communication (e.g., realistic convergence processes and communication patterns).

2. IN-NETWORK SUPPORT FOR DEBUGGING

In this section, we describe our overall approach towards supporting the three network-level primitives mentioned in Section 1. We start by proposing a strawman architecture for isolating the operational network based on *virtual service platforms*. Many of the key enabling technologies for this architecture have been developed in previous work [17, 4], however, we lack algorithms to interactively debug and step through network software atop this design. We hence proceed to describe two key algorithms, one for *interactive stepping*, and another to ensure *reproducibility* of network execution.

Isolating the operational network with *virtual service platforms*: To isolate the operational network from interactive debugging activities, our architecture leverages virtualization technologies to encapsulate networked software into *virtual service platforms* (VSPs), which individually provide the abstraction of a single dedicated hosting environment (virtual network) to the network software (Figure 1). VSPs are logically decoupled from their underlying hardware, and run atop a *virtual service hypervisor*, which manages network and computational resources across VSPs. VSPs may be *cloned* from other VSPs or directly from the live network (which itself is run as a VSP) to perform debugging, and when a fault is repaired the new VSP may be *merged* back into the live network. The virtual service hypervisor allocates resource slices to each VSP and performs priority-based scheduling to isolate the live network from VSP activity. Execution of each VSP is controlled with use of a *runtime manager*, which allows speedup/slowdown of execution, time reversal/replay of previous events, and pausing of execution to view a fixed state of the system. The human troubleshooter interacts with the system through a *debugging coordinator* (DC), which coordinates runtime operation of the entire virtual network. Individual network software processes run with *virtual service coordinators* (VSCs), which are parts of the VSP that corresponds to each physical node. The DC accepts debugging commands (e.g., to insert breakpoints, or to print state, or to manipulate execution) from the human troubleshooter. These commands are sent to a coordinator module, which translates them into low-level algorithmic instructions for VSCs. The DC receives event notifications from VSCs, for example when a new control packet is sent, or a router's FIB is updated. To allow the troubleshooter to step through, slow down, pause, reverse, and replay operation, the VSC runs each virtual instance atop a *runtime manager*, which controls process execution and timing. An *event filter* module determines which events the VSC will send to the DC, and a *backplane routing* process is run to construct routes via a separate virtual network that are isolated from the debugger (used to ensure the DC can always reach all VSCs).

Interactive stepping by *distributed lockstep execution*: To interactively step through execution, the runtime manager must coordinate execution across the distributed set of processes making up the software service. This is done by logically dividing the service's execution into a series of *timesteps*. These

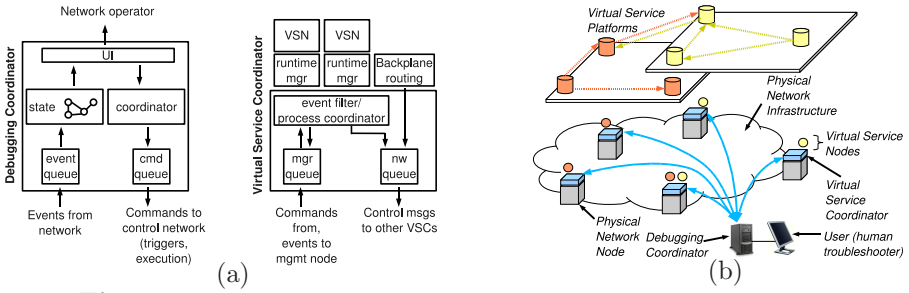

Figure 1: ***System architecture (a) component view (b) deployment view.***

timesteps may be chosen at various levels of granularity (per-unit-time, per-message, per-path-change). The VSP then runs the service software in virtual time, by executing it in lockstep across nodes, by alternating between two phases: first, VSPs exchange messages, and second, VSPs replay those messages to their local server software and collect outbound messages to be sent in the next cycle. A distributed semaphore is used by nodes to agree on when phases occur. We give more details in Section 3.1.

While this approach controls execution, it also limits the set of possible execution paths the service software may traverse by limiting dynamics such as convergence and randomness arising from event orderings. However, dynamics are important to debug as well, since they affect convergence behavior and other aspects of system-wide performance, and since some problems are only triggered in the presence of dynamics. To address this, we define an *ordering model* that produces a realistic ordering of events to be delivered to nodes. The VSP uses the ordering model to associate a virtual arrival time with each message, to determine when it should be delivered to the service software instance. The troubleshooter may additionally change ordering models to alter the execution environment in an effort to trigger new bugs. In future work we aim to derive ordering models that characterize the service's dynamics (e.g., by leveraging formal characterizations mapping events to a distributed set of routing protocol messages [10]), but for now our design simply assigns a pseudorandom ordering.

Reproducing network execution by *masking randomness*: Debugging a system becomes much easier if the operation of that system is reproducible. In such a system, a developer may re-execute the system multiple times, evaluating hypotheses and testing workarounds, and can expect the same execution each time. The developer may also be able to recreate rarely-encountered exceptions, by rolling the system back to a state when the fault was known to occur, and replaying from that point. For example, consider a simple single-computer program that executes differently based on a call to *rand()*.

Unfortunately, existing network software incorporates a high degree of randomness and nondeterminism in its execution, arising from varying packet orderings, delay and jitter, and other variables arising from distributed execution. To address this, our design manipulates the operation of the live network itself, to cause it to run in a reproducible way. Here, each VSC employs virtualization technologies to facilitate reproducibility (e.g., handling timers). Then, each VSP intercepts messages from the network before delivering them to the network software, and then uses a pseudorandom sequence to determine the exact orderings and timings at which to send the messages up to the network software. Coming up with this ordering may be done in a variety of ways, for example there already exist algorithms that can instill an ordering of events across a set of nodes [14], and these algorithms are fully compatible with our design. As an optimization to reduce overheads, we also consider an "optimistic" approach: each node independently decides on a pseudorandom sequence of events, and then lets the network execute in an arbitrary fashion. If the order in which events execute is different from the pseudorandom sequence, the network is "rolled back" to an earlier state, and played forward with the correct ordering. Now, predicting the ordering in which certain network events (e.g., link failures) will happen is an extremely challenging problem. However, in order to debug a system, it is not necessary to reproduce events taking place outside the system, such as link failures. Hence to simplify this problem, we only predict *internal* events taking place within the software in the VSC, and rely on logging to record *external* events such as link failures and externally-received routing updates (logs are only required when interfacing with external systems, for example if testing a system using a synthetically generated workload no logging is required). We give more details in Section 3.2.

3. ALGORITHM DESIGN

Next, we describe two algorithms that run within each VSP. First, the *live algorithm* (Section 3.1), instruments the production network to make its execution reproducible. The idea here is for the VSP to

manipulate the way the live network operates (by reordering messages, and controlling the time at which messages are delivered) so that the network software running above acts in a reproducible way. Second, the *lockstep algorithm* (Section 3.2), allows a cloned copy of the live network to be "stepped" through in a manner controlled by the human debugger. This algorithm works by making the cloned copy run in a lockstep fashion, so that in each step, control messages are only allowed to propagate a single hop. When provided with a checkpoint collected from the live network's operation, the lockstep algorithm reproduces the exact sequence of events that occurs in the production network after that checkpoint. Both algorithms eliminate the nondeterminism introduced by distributed execution, but they have different characteristics: the live algorithm adds reproducibility to the production network with acceptable overheads, while the lockstep algorithm enables execution to be paused and stepped through.

3.1 Interfacing with the Production Network

In order to make the live network's execution reproducible, we need to ensure the ordering of messages propagating through the network can be precisely regenerated after-the-fact. We do this by having the VSP running beneath the live network instilling a pseudorandom *ordering* on the way messages generated by the software propagate through the network. Given a log of network-level events, the system's execution can then be precisely generated by regenerating this ordering. To do this, we need to solve two problems: (i) we need some way to come up with the pseudorandom ordering (ii) we need some way to perform the rollback/reversal when the predicted ordering is violated.

Algorithm 1 live_recv(**pkt**)

```
1: history.insert(pkt)
2: if history.back() != pkt then
3:     rollback(history, pkt)
4: else
5:     deliver(pkt)
6:     history.update(pkt)
7: end if
```

Ordering packets: Since we rollback whenever the ordering is violated, we could simply choose any arbitrary pseudorandom ordering. However, to improve performance, we would like to minimize the number of times we have to roll back. Hence, as an optimization we select orderings that are likely to happen anyway, given the typical delays of network links (Algorithm 1). In particular, we have each node select a pseudorandom sequence that closely matches the ordering of messages the node would expect to receive from simultaneous events. For example, if a node is connected to one upstream node with a low-delay link and another with a high-delay link, it is reasonable to expect packets to arrive from the former node sooner. To instill this ordering, a VSC maintains a *history* of messages it has received so far (older entries in this list are garbage collected after some delay to reduce state requirements). The history is sorted by the source and incoming interface of the message according to a pseudorandom ordering. Every time a VSC receives a message, it will insert it into the history, and also send it up to the network software. If later on another message arrives that should have been sent up to the software first (before a message that was already sent up) according to the ordering, the network software will be rolled back, and the messages will be played back in the order given by the pseudorandom sequence.

Algorithm 2 rollback(**history**, **pkt**)

```
1: for tpkt ∈ [history.back(), history.search(pkt)) do
2:     history.search(tpkt).restore()
3:     history.search(tpkt).cleanup()
4: end for
5: for tpkt ∈ [history.search(pkt), history.back()] do
6:     live_receive(tpkt)
7: end for
```

Algorithm 3 cleanup(**pkt**)

```
1: for tpkt ∈ [history.back(), history.search(pkt)] do
2:     history.search(tpkt).restore()
3:     history.search(tpkt).cleanup()
4: end for
5: for tpkt ∈ (history.search(pkt), history.back()] do
6:     live_receive(tpkt)
7: end for
```

Rolling back: To roll back misordered messages, we employ two algorithms: Algorithm 2 rolls back the state of network software, and Algorithm 3 cleans up outputs generated by misordered messages (the `restore()` method relies on existing virtualization technologies to rollback state, our focus here is only on determining causal relationships necessary to perform the rollback across nodes). For example, if a node reads in packets p_1, p_2, and outputs a packet p_3, and we later find out p_1 is going to be rolled back, then the question of whether p_2 and p_3 need to be rolled back depends on their relationship with p_1. In sight of this, we assume after delivering a packet to the network software, a VSC knows if the software will produce any packet in response of the input. In addition, a VSC could pinpoint which output packet is corresponding to which input. If the software doesn't read in packets it won't use immediately (which is the case for common network software), the assumption trivially holds; otherwise, we could modify the network software or use virtualization technologies to provide explicit information to the VSC. With this assumption, a VSC records additional information with each entry in the his-

tory: the changes of the state of the software before and after processing the given message, and the exact output produced. Then, the VSC could rollback messages received out of order by restoring the local state, cleaning up output packets sent, and redeliver the messages.

Grouping events: The above algorithms provide reproducibility. However, in the worst case (if there were a very long causal chain), the system would have to perform many rollbacks. To bound the number of rollbacks, we group packets in the system into *events* according to their causal relationships. We then divide the timeline into blocks, group events appearing in a single block together, and then independently impose an ordering on each group. One VSC is assigned to periodically broadcast *group numbers*. Each VSC maintains a local copy of the latest group number received and tags the packets triggering an event with the number. The output packets generated by a particular input packet are assigned the same group number as the input packet. The history is then divided into groups, and the ordering is instilled only within each group. With this, an entry in the history can be removed after two times the maximum propagation time across the network after its group becomes inactive since all messages that would be ordered before it would arrive by then.

3.2 Reproducing Execution in a Cloned Instance

The live algorithm supports reproducibility with minimum perturbation to the production network. However, it doesn't offer a easy way to step through the execution, and thus make it less desirable to be run in a cloned instance where delay and overheads are not the major issue. In light of this, we introduce a *lockstep algorithm* that operates in a debugging environment which could be run on the same physical infrastructure as the live network, or a separate server-based testing network. Instead of applying the same mechanism as the live algorithm, the lockstep algorithm achieves reproducibility by forcing the network to run in a *lockstep* fashion. This is done by explicitly queuing messages between nodes (and timer events), and playing them at precise intervals. Since it is deployed in a debugging environment, where requirements for delay and overheads are not that demanding, the mechanisms are justified. To make the network run in lockstep, the algorithm instructs each VSC to cycle through two phases: a *transmission* phase and a *processing* phase. The algorithm coordinates all VSCs with a mechanism similar to a distributed semaphore to make sure they are in the same phase at the same time. Between transitions through these phases, each VSC uses buffers and timeouts to eliminate the nondeterminism introduced by the communication subsystem, and hence make it reproducible.

Transmission phase: In this phase, each VSC sends all messages generated in the previous processing phase. In particular, the VSC sends out packets in a *sending buffer* (filled in the processing phase) and stores all packets received in a *receiving buffer* (events from live network logs may also be injected during this phase). To ensure reproducibility, contents of the receiving buffer are sorted in the same way as the one running the live algorithm in the production network. To indicate readiness to transition to the processing phase, a VSC sends an *end of sending* packet when it has no further packets to send.

Processing phase: In this phase, each VSC processes all messages received during the previous transmission phase. In particular, the VSC sends all messages in the receiving buffer up to the network software, and enqueues the software's generated packets into the sending buffer. The VSC waits for the network software to finish processing packets before transferring to the transmission phase.

4. EVALUATION

Our design simplifies troubleshooting, but comes with several costs. As an early first step towards measuring these costs, we evaluate our design for OSPF running atop several network topologies constructed with the BRITE topology generator, with link delays set to 1ms plus a $(0.0, 0.5]$ ms exponentially distributed random delay. We compare a baseline network running OSPF directly atop the physical network with a network running OSPF within a VSP. For realistic failure conditions, we extract link failure events from Abilene ISIS traces over the month of Jan. 2009 (giving 209 events), and map events to random links in the BRITE topology (we also ran simulations directly over the Abilene topology and acquired similar results). Each run is repeated 10 times (standard deviation bars in charts lack visibility due to their small size). The grouping time for the live algorithm is set to 25 ms. We measured performance penalties along several axes:

Delay and control overhead: Figure 2a shows the additional convergence delay incurred by the live algorithm, which is somewhat high. While this delay may be tolerable for certain protocols (e.g., BGP uses an MRAI timer to intentionally slow convergence for scalability), it may harm convergence of other protocols. To address this, we implemented the optimization described earlier in Section 3.1, which instills an ordering of events that is close to what we would expect the network to do (based on observed link delays and propagation times), which vastly re-

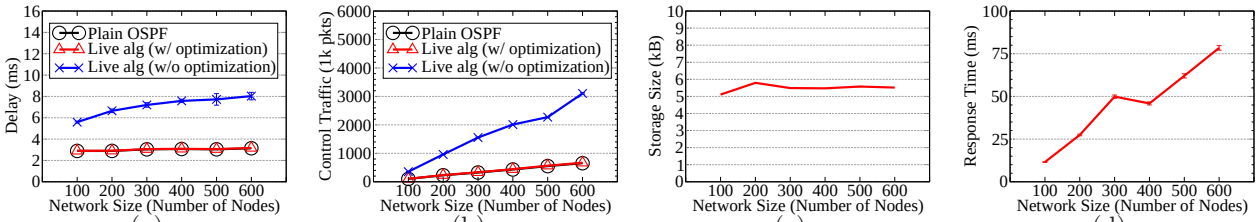

Figure 2: ***Live algorithm: (a) delay (b) overheads (c) storage. Lockstep algorithm: (d) response time per step.***

duces the number of rollbacks. With this optimization, we see a large reduction in delay (less than 3% increase over plain OSPF, causing the two lines to overlap in the figure). The optimization produces a similar decrease in control overhead (Figure 2b).

Storage requirements: Our approach requires additional state to be kept at routers, for the *history* of received packets needed for rollback. Figure 2c shows these requirements to be small and increase slowly with network size, and should fit in the megabytes or gigabytes of DRAM available in modern routers. Furthermore, to reproduce events in a deterministic system, one only needs to log external events, so a secondary benefit of our design is in reducing log sizes by a factor of 480 to 3000.

Response time: To support *interactive* debugging, our system should have fast response to commands from the human troubleshooter. Figure 2d shows the response time to execute a single *step* command of the system (where a single step moves between two phases of the lockstep algorithm). We find that latency is generally under 100 milliseconds.

5. RELATED WORK AND CONCLUSION

Related work: Architectural principles to increase observability of the network include tracing, performance monitoring, root cause notification, beaconing and probing, and logging observations (e.g., [7, 12]). As well as automatically pinpointing faults: root cause analysis, system models, anomaly detection, and correlating observations (e.g., [6]). However, this work focuses solely on non-interactive techniques (which could be used to supplement our design). On the other hand, there are works focusing on using recorded states to roll back [11] or replay [9] the systems, but they are mainly limited to standalone systems. There has also been older work on interactive debugging, some of it on parallel systems (e.g., [15, 8, 3]), but this work was prevalent only before the advent of modern distributed networks and protocols, with recent research focusing on entirely removing the human from the loop. Finally, we owe much to work on network virtualization [4, 17] and distributed algorithms [14] which form the foundations of our design.

Conclusion: The high complexity of ISP networks coupled with the rich variety of faults they undergo will require humans to be "in-the-loop" to diagnose problems for the foreseeable future. To address this, we propose a strawman architecture for *interactive debugging* of network software. This architecture presents a number of challenges, and we draw from previous work as well as propose new algorithms to solve some of them. For future work, we will refine this design, by implementing primitives for cloning and merging network state, and evaluate our design through a realistic system implementation.

6. REFERENCES

[1] Coverity Inc. `www.coverity.com`.

[2] Software errors cost U.S. economy $59.5 billion annually. *NIST News Release*, October 2002. `www.nist.gov/public_affairs/releases/n02-10.htm`.

[3] PDB parallel debugger. July 2009. `http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=/com.ibm.cluster.pe_linux43.opuse.doc/am102_pdbman.html`.

[4] R. Alimi, Y. Wang, and Y. Yang. Shadow configuration as a network management primitive. *SIGCOMM*, August 2008.

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

[6] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, May 2005.

[7] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: a pervasive network tracing framework. In *NSDI*, April 2007.

[8] G. Fox, R. Williams, and P. Messina. *Parallel Computing Works! (Chapter 5.3: Parallel Debugging)*. Morgan Kaufmann, 1994.

[9] D. Geels, G. Altekar, P. Maniatis, I. Stoica, and T. Roscoe. Friday: global comprehension for distributed replay. In *NSDI*, April 2007.

[10] T. Griffin. What is the sound of one route flapping? *presentation made at the Network Modeling and Simulation Summer Workshop*, 2002.

[11] D. R. Jefferson and A. Motro. The time warp mechanism for database concurrency control. pages 474–481, 1986.

[12] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *SIGCOMM*, August 2004.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access-interleaving invariants. In *IEEE Micro*, January-February 2007.

[14] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[15] D. McDowell and D. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, December 1989.

[16] P. Roberts. Cisco tries to quash vulnerability talk at Black Hat. In *eWEEK*, July 2005.

[17] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive. In *SIGCOMM*, August 2008.