FatTire: Declarative Fault Tolerance for Software-Defined Networks

Mark Reitblatt
Cornell University
reitblatt@cs.cornell.edu

Marco Canini
TU Berlin / T-Labs
m.canini@tu-berlin.de

Arjun Guha
Cornell University
arjun@cornell.edu

Nate Foster
Cornell University
jnfoster@cornell.edu

ABSTRACT

This paper presents FatTire, a new language for writing fault-tolerant network programs. The central feature of this language is a new programming construct based on regular expressions that allows developers to specify the set of paths that packets may take through the network as well as the degree of fault tolerance required. This construct is implemented by a compiler that targets the in-network fast-failover mechanisms provided in recent versions of the OpenFlow standard, and facilitates simple reasoning about network programs even in the presence of failures. We describe the design of FatTire, present algorithms for compiling FatTire programs to OpenFlow switch configurations, describe our prototype FatTire implementation, and demonstrate its use on simple examples.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network operating systems; D.3.2 [**Language Classifications**]: Specialized application languages

Keywords

Fast failover, fault tolerance, NetCore, Frenetic, OpenFlow

1. INTRODUCTION

"To find fault is easy, to do better may be difficult."
—Plutarch

Networks are expected to operate without disruption, even in the presence of device or link failures. Accordingly, many networks employ advanced mechanisms that allow routers and switches to rapidly respond to failures, restoring connectivity in 10s of milliseconds [20]. At the same time, networks are expected to do much more than provide connectivity—they must also provide rigorous security and performance guarantees, even while recovering from failures. For example, if a switch diverts traffic along a backup path due to a link failure, packets must not be allowed to circumvent the firewall, thereby violating the network's security policy.

HotSDN'13, August 16, 2013, Hong Kong, China.

The promise of software-defined networking (SDN) is to enable network designers to construct networks that meet their specific, end-to-end requirements, rather than forcing them to stitch together existing protocols, each with their own capabilities, features, and limitations. Although there has been some work on deploying failure-recovery *mechanisms* in SDN [7, 18], programmers today lack *abstractions* for specifying failure-recovery policies, as well techniques for automatically integrating those mechanisms into network programs. In practice, developers today must either add complicated failure-handling code to programs by hand, or throw correctness guarantees to the wind when failures occur.

We argue that SDN programmers should have high-level constructs that allow them to specify distinct policy concerns, such as forwarding, performance, security, and fault-tolerance. In addition, SDN programmers should be able to reason about the interactions between those constructs when they are combined in a single program. To this end, we present the design and implementation of a new language called FatTire that provides the following features:

1. *Expressive:* natural and orthogonal programming constructs that make it easy to describe forwarding and fault-tolerance policies.

2. *Efficient:* a proof-of-concept implementation based on translation to the fast-failover mechanisms provided in recent versions of OpenFlow.

3. *Correct:* a methodology for reasoning about the behavior of the system during periods of failure recovery, which enables verification of network-wide invariants.

The central feature of FatTire is a new programming construct based on regular expressions that allows programmers to declaratively specify sets of legal paths through the network, along with fault-tolerance requirements for those paths. The FatTire compiler takes programs specified in terms of paths and translates them to OpenFlow switch configurations that automatically respond to link failures without controller intervention. Compiling FatTire turns out to be significantly more challenging compared to other SDN languages like NetCore [4, 12] for several reasons: (i) FatTire programs are non-deterministic due to the use of regular expressions; (ii) the translation to individual switch configurations requires a global analysis, and (iii) there can be tricky interactions between paths when failures occur.

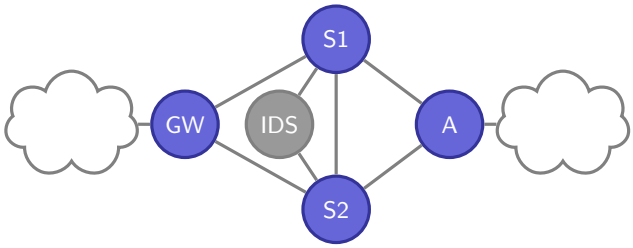

Figure 1: Example network.

We have engineered a compiler for FatTire that correctly handles each of these issues.

Overall, the contributions of this paper are as follows:

- We design a new language for writing fault-tolerant SDN programs that provides paths as a basic programming construct (§3).
- We present algorithms for compiling FatTire programs to OpenFlow switches that take advantage of in-network fast-failover mechanisms (§4).
- We describe our prototype implementation of FatTire, which builds on the NetCore compiler (§5).
- We evaluate FatTire on a simple example program (§6).

The next section presents a practical example that motivates the need for declarative fault-tolerance programming abstractions. The following sections describe each of our main contributions in detail.

2. PROGRAMMING FAULT TOLERANCE

As motivation, consider the enterprise network shown in Figure 1, and assume we want to construct a configuration with the following properties:

(i) SSH traffic arriving at the gateway switch (GW) should be eventually delivered to the access switch (A),

(ii) incoming SSH traffic should traverse the middlebox (IDS) before being reaching internal hosts

(iii) the network should continue forwarding SSH traffic even if a single link fails.

It is easy to build a configuration with the first two properties. For instance, we can forward incoming SSH traffic along the path [GW,S1,IDS,S2,A]. But to provide the specified fault-tolerance property, each of the links in this primary path also needs a backup. There are numerous possible backup paths,

- [GW,S2,IDS,S2,A] if (GW, S1) fails,
- [GW,S1,S2,IDS,S2,A] if (S1, IDS) fails,
- [GW,S1,IDS,S1,A] if (IDS, S2) fails, and
- [GW,S1,IDS,S2,S1,A] if (S2, A) fails.

If the policy required protection against two link failures then we would also need backup links for the backup paths; three failures would require backups for the backups of the backups, and so on.

GW Ruletable and Grouptable

Match	Instructions
tpDst = 22	Group 1

Group	Type	Actions
1	FF	⟨Fwd S1⟩, ⟨Fwd S2⟩

S1 Ruletable and Grouptable

Match	Instructions
inPort = GW, tpDst = 22	Group 1
inPort = IDS, tpDst = 22	Group 2
inPort = S2, tpDst = 22	Group 2

Group	Type	Actions
1	FF	⟨Fwd IDS⟩, ⟨Fwd S2⟩
2	FF	⟨Fwd A⟩

S2 Ruletable and Grouptable

Match	Instructions
inPort = IDS, tpDst = 22	Group 1
inPort = S1, tpDst = 22	Group 2
inPort = GW, tpDst = 22	Group 2

Group	Type	Actions
1	FF	⟨Fwd A⟩, ⟨Fwd S1⟩
2	FF	⟨Fwd IDS⟩

Figure 2: Example ruletables and grouptables.

Even this simple example requires a non-trivial program. For example, traffic can reach S1 and S2 under at least four different scenarios. To ensure that traffic is handled correctly, it is necessary to consider every possible interaction between primary and backup paths—a tedious and error-prone task for the network programmer.

OpenFlow. To illustrate the complexity of building fault-tolerant configurations manually, consider how we would do this in OpenFlow. The following rule implements the primary path for SSH traffic on switch S1:

Match	Actions
inPort = GW, tpDst = 22	⟨Fwd IDS⟩

It consists of a *match* that specifies packet attributes (*e.g.*, transport destination port 22 for SSH traffic) and a list of *actions* that specify how to process matching packets. In this case, the rule states that all SSH traffic coming from GW should be forwarded to IDS. For simplicity, we have replaced the names of ports with the switches they are connected to—*e.g.*, in place of the name of the port connecting S1 to GW, we simply write GW.

Configuration updates. Early versions of OpenFlow did not support rules that depend upon switch state—*e.g.*, rules that test whether a link has failed or not. Hence, the only way to respond to failures was for the controller to explicitly intercede by installing new rules in response to the failure. For example, if the switch S1 detected a failure on the link to IDS, it would notify the controller which would then emit a new rule directing traffic along the backup link (S1, S2), as well as a new rule on S2 specifying how to forward the traffic coming from S1. This approach to dealing with failures works, in a sense, but it can take a substantial amount of time to restore connectivity [18]. Moreover, while the con-

troller is installing the new rules, traffic continues flowing through the network and may encounter partially installed and inconsistent ruletables on different switches, causing unexpected forwarding behaviors and potentially violating network policies. Techniques based on so-called *per-packet consistent updates* [17] ensure that such violations do not occur, but they are designed for planned change rather than rapid response to failures, and can further delay recovery.

Fast failover. Recent versions of OpenFlow include support for conditional rules whose forwarding behavior depends on the local state of the switch. A new type of forwarding table called a *group table* contains entries whose rules include "an ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters" [2]. Action buckets provide the ability to define multiple forwarding behaviors. When the type of a group table is "FF" (fast failover),[1] each bucket is associated with a parameter that determines whether the bucket is live, and the switch forwards traffic to the first *live* bucket. As the parameter to determine liveness, the programmer either specifies an output port or a group number (to allow several groups to be chained together).

For example, here is the ruletable and grouptable for S1. We omit the liveness parameter—in this case, just an output port for each bucket:

Match	Instructions
inPort = GW, tpDst = 22	Group 1

Group	Type	Actions
1	FF	⟨Fwd IDS⟩, ⟨Fwd S2⟩

Note that the primary rule for S1 explicitly handles the backup case by directing traffic to a fast-failover group with a backup bucket installed. Figure 2 presents complete rule and group tables for our running example.

With fast-failover groups, the controller program must anticipate every possible failure and precompute appropriate backup paths (including working out the interactions between traffic on different backup paths), rather than reacting to link failures as they occur. Hence, while fast-failover groups make it possible to implement rapid failure recovery, using them correctly places a heavy burden on the SDN programmer. We argue that programmers should not have to write programs using the primitive fast-failover mechanisms provided in OpenFlow—the fault-handling logic quickly becomes complex and makes the meaning of the rest of the program difficult to follow. For example, in Figure 2, the failover logic completely obscures the otherwise straightforward network program.

3. THE FATTIRE LANGUAGE

FatTire (for **Fa**ult **T**olera**t**ing **R**egular **E**xpressions) is a new high-level programming language that provides constructs for writing programs in terms of paths through the network and explicit fault-tolerance requirements. The FatTire compiler generates ruletables and grouptables that provide the specified fault-tolerance while guaranteeing that traffic flows along the paths dictated by the program. This turns out to be non-trivial, since multiple paths often cross a given switch, and the correct behavior can depend on link failures elsewhere in the network.

[1]Other group types implement multicast and load sharing.

Path exp. $P ::= S \mid \star \mid P.P \mid (P||P) \mid (P\&P) \mid \neg P$

Switch ID	sw	$\in \mathbb{N}$	
Headers	h	::= `dlSrc`	*source* MAC
		$\mid$ `dlDst`	*destination* MAC
		$\mid \ldots$	
Predicate	pr	::= `any`	*wildcard*
		$\mid h = n$	*match header*
		$\mid$ `not` pr	*predicate negation*
		$\mid pr_1$ `and` pr_2	*predicate intersection*
Program	pol	::= $pr \Rightarrow P$ `with` n	*atomic policy*
		$\mid pol_1 \uplus pol_2$	*policy union*
		$\mid pol_1 \pitchfork pol_2$	*policy intersection*

Figure 3: FatTire syntax.

$$
\begin{array}{rcl}
(pr_1 \Rightarrow P_1 \texttt{ with } k_1) & \pitchfork & (pr_2 \Rightarrow P_2 \texttt{ with } k_2) \\
& \equiv & \\
(pr_1 \texttt{ and } pr_2) & \Rightarrow & (P_1 \& P_2) \texttt{ with } \max(k_1, k_2) \\
\\
(pr_1 \Rightarrow P_1 \texttt{ with } k_1) & \uplus & (pr_2 \Rightarrow P_2 \texttt{ with } k_2) \\
& \equiv & \\
(pr_1 \texttt{ and } (\texttt{not } pr_2)) & \Rightarrow & (P_1) \texttt{ with } k_1 \\
\uplus \quad (pr_1 \texttt{ and } pr_2) & \Rightarrow & (P_1 || P_2) \texttt{ with } \max(k_1, k_2) \\
\uplus \quad ((\texttt{not } pr_1) \texttt{ and } pr_2) & \Rightarrow & (P_2) \texttt{ with } k_2
\end{array}
$$

Figure 4: FatTire normalization rules.

To illustrate the main features of FatTire, consider the running example from the preceding section. In FatTire, rather than manually crafting the ruletables and grouptables in Figure 2, we can simply write the following program, which generates the same tables:

$$
\begin{array}{l}
(\texttt{tpDst} = 22 \Rightarrow [\star.\mathsf{IDS}.\star]) \\
\pitchfork\ (\texttt{tpDst} = 22 \Rightarrow [\star] \texttt{ with } 1) \\
\pitchfork\ (\texttt{any} \Rightarrow [\mathsf{GW}.\star.\mathsf{A}])
\end{array}
$$

This program has three components. The first is the security policy, given by the first line, which states that all SSH traffic must traverse the IDS. We use regular expressions over switches to describe legal paths. The second is the fault-tolerance policy, given by the `with` annotation, which states that forwarding must be resilient to a single link failure. The third is the routing policy, given by the second line, which states that traffic from the gateway (GW) must be forwarded to the access switch (A), along any path. The top-level program intersects the routing and security policies, which means that all paths must satisfy both. The overall result is that SSH traffic (i) always traverses the IDS, and (ii) is resilient to single link failures, and (iii) is routed along a path from GW to A.

Note that in this program, all three pieces of functionality are described clearly and independently, without explicitly reasoning about failure scenarios, primary and backup paths, or the interactions between them.

The full syntax of FatTire is shown in Figure 3. The language is inspired by previous work on NetCore [4, 12], but adds support for paths and regular-expresions, fault-tolerance annotations, and an intersection operator on policies. In examples, we will often omit fault-tolerance and

assume a default fault-tolerance annotation of 0. Semantically, intersecting two policies results in a policy whose paths are the paths described by both policies and whose fault-tolerance is the maximum of the fault-tolerance provided by the individual policies.

In addition to intersection, policies can be unioned together. For example, the output of a MAC learning module would be the union of the individual policies encoding the locations of each known host:

$$\begin{array}{rl} & (\texttt{dlDst} = 00{:}00{:}00{:}00{:}00{:}01 \Rightarrow [\star.\mathsf{S1}]) \\ \uplus & (\texttt{dlSrc} = 00{:}00{:}00{:}00{:}00{:}01 \Rightarrow [\mathsf{S1}.\star]) \end{array}$$

The next section describes how to compile FatTire programs to OpenFlow ruletables and grouptables.

4. THE FATTIRE COMPILER

Compilation of a FatTire policy proceeds in four phases:

1. We normalize the input policy to a union of *atomic policies*, each with non-overlapping predicates.
2. We construct a fault-tolerant *forwarding graph* for each atomic policy.
3. We translate the forwarding graphs to *policies* in NetCore, extended with a left-biased union operator, and add explicit logic for transitioning between forwarding graphs when failures occur.
4. We compile the resulting policies to OpenFlow using an extension of the NetCore compiler that translates left-biased union using fast-failover groups.

The next few paragraphs describe these phases in detail.

Normalization. The first phase of compilation normalizes the input policy into a union of atomic policies with disjoint predicates. First, the input policy is converted to a Disjunctive Normal Form, with unions of intersections of atomic policies, and then the intersections are eliminated using the first rule in Figure 4. The resulting union of atomic policies is then iteratively refined using the second rule in Figure 4 until the atomic policies match disjoint sets of packets. Normalizing serves two purposes: (i) it combines the separate policies (security, routing, *etc.*) into a single coherent policy and (ii) it divides the program into disjoint pieces that can be compiled independently. In theory, normalization can take exponential time, but in practice the input predicates are mostly disjoint so it converges quickly. The normalized policy for our running example consists of a single rule,

$$(\texttt{tpDst} = 22 \Rightarrow [\mathsf{GW}.\star.\mathsf{IDS}.\star.\mathsf{A}] \texttt{ with } 1)$$

which captures security, routing, and fault-tolerance.

Constructing fault-tolerant forwarding graphs. The second phase of compilation constructs a fault-tolerant forwarding graph that is consistent with the program's path expression, and has as many backup paths as the fault-tolerance annotation requires. We represent these paths as a forwarding graph with backup links. If the policy is incompatible with the topology, either because it requires an impossible path (*e.g.*, forwarding between unconnected nodes), or because there is not enough redundancy to support the required fault tolerance, the compiler halts with an error.

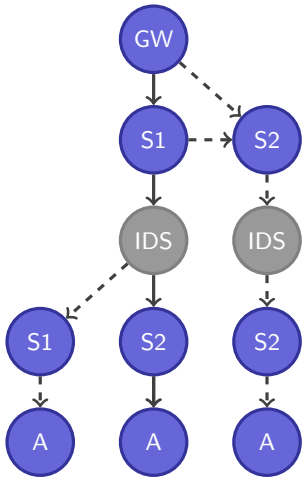

Figure 5: Example forwarding graph.

Figure 5 shows the forwarding graph for the SSH policy, using solid lines to indicate primary paths and dashed lines to indicate backup paths. Nodes along the primary path [GW,S1,IDS,S2,A] have a primary and backup rule, while nodes along the backup paths have only a backup rule. Because the policy only requires resilience to a single link failure, once traffic has been diverted to a secondary path we no longer need backup rules. Note that S1 and S2 each appear twice along certain paths. Because we keep track of the incoming port in each rule, we can handle topological cycles, as long as any repeated switches are reached along a different link each time they are visited in the cycle.

The full details of the algorithm to compute fault-tolerant forwarding graphs can be found in our implementation. It is based on a breadth-first-traversal of the graph through two mutually recursive functions. The first function takes a primary path and recurses down it, installing fault-tolerant trees at each node. The second function takes a node and does a breadth-first recursion across its children, installing backup paths for that node. If one of the functions fails, it backtracks by either picking a new primary path, or choosing a different ordering on the children in the traversal.

We use regular expression derivatives [14] to keep track of the legal backup paths from a given node as we recurse through the graph. The derivative of a regular expression R with respect to a character c is the set of all strings t such that $c \cdot t \in R$. In this context, the derivative of a path regular expression with respect to a given switch s is the set of legal paths starting at s. By taking the derivative of the regular expression at each hop, we keep track of the current position in the regular expression, and how it can continue to be expanded into a legal path. For example, when we compile the regular expression from the example, [GW.$\star$.IDS.$\star$.A], we first iterate over every switch (start of a path) and take the derivative with respect to that switch. Because the regular expression starts with the explicit hop GW, the derivative at any switch other than GW will be the empty regular expression, while the derivative at GW will be the remainder [$\star$.IDS.$\star$.A]. The algorithm continues, performing a breadth-first search through the switches that can possibly start a legal path (have a non-empty derivative), until we have satisfied the regular expression.

Forwarding graph to NetCore. The third compiler phase converts the forwarding graph to an equivalent NetCore program. Standard NetCore programs do not support the fast-failover groups of recent versions of OpenFlow, so we extended the language and compiler with a new left-biased policy operator. The policy behaves like the left sub-policy unless it fails (by forwarding out a dead port), and otherwise behaves like the right sub-policy.

To convert the forwarding graphs into NetCore programs, we iterate over each node in the forwarding graph, create a new group whose fail-over actions forward along its links (in order), and generate a rule that applies that action for that group. For example, because the highest S1 node has two children, IDS and S2, the generated NetCore policy for S1 would handle SSH traffic from GW using a left-biased union of policies that forward to S2 and IDS. To finish the job, we union the NetCore policies together, add tags to distinguish traffic on each forwarding graph, and add additional logic to switch between forwarding graphs when failures occur.

NetCore to fast-failover OpenFlow. The final compiler phase translates NetCore policies extended with the left-biased union operator into OpenFlow fast-failover groups. We have extended the standard NetCore compiler with support for left-biased union. To illustrate, the generated rule and group table for S1 would be:

Match	Instructions
`inPort = GW and tpDst = 22`	⟨`Group 1`⟩

Group	Type	Actions
1	FF	⟨`Fwd IDS`⟩, ⟨`Fwd S2`⟩

5. IMPLEMENTATION

We have built a full working prototype compiler for the FatTire language in OCaml. The compiler takes as input a FatTire program and a topology, and emits a NetCore policy as output. We have also extended the NetCore compiler and run-time system to support left-biased union and OpenFlow fast-failover groups as described above. These developments can be found at `http://frenetic-lang.org`.

6. EVALUATION

To evaluate the performance of our FatTire implementation, we conducted a simple experiment in Mininet [10] using the CPqD OpenFlow 1.3 software switch [1]. We used `iperf` to transfer 100MB of data between a host attached to GW and one attached to A in the topology depicted in Figure 1 (modified slightly so that S2 and IDS are co-located on the same node) and measured the time needed to complete the transfer. Note that because we used a network simulator and software switches, the absolute completion times are not meaningful, but relative comparisons are meaningful. We used a 2.4GHz machine with 8 cores and 32GB of RAM, and repeated the experiment 75 times.

We compared the completion times under two scenarios: in the first, the network had no failures, so packets could simply be forwarded along the primary path to their destination, [GW, S2, A]. In the second scenario, we broke the link between GW and S2 after 20 seconds, forcing traffic to traverse a longer backup path [GW,S1,S2,A].

The results are shown in a boxplot in Figure 6. Note that after the link fails, even with the additional processing delay incurred by fast-failover and the longer backup path, the FatTire completion time is only marginally higher than the baseline completion time. Overall, this preliminary experiment demonstrates that FatTire programs are able to respond extremely rapidly to failures, as desired.

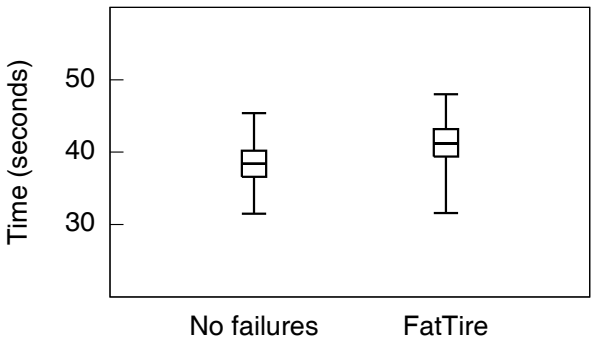

Figure 6: Transfer completion time achieved by fast failover as enabled by FatTire is only slightly higher than when no failure occurs.

7. RELATED WORK

There is a large body of work on techniques for recovering from failures in many diverse settings [3, 9, 15, 16, 20]. Recently, Liu *et al.* [11] argued that connectivity recovery should be realized as a data-plane service. Their work dovetails with ours by providing mechanisms for implementing the policy expressed using our abstractions. Likewise, recent work on integrating fault-tolerance and traffic engineering by Suchara *et al.* [19] could potentially be used in conjunction with our abstractions.

In the context of SDN, Kempf *et al.* [7] proposed a fault management approach similar to MPLS global path protection, which they argue should be part of OpenFlow. However, their focus is on extending the OpenFlow switch software with end-to-end path monitoring capabilities. Their work is orthogonal to ours in that monitoring capabilities may be used to detect path failures in our scheme. Kuźniar *et al.* [8] proposed a system that provides automatic failure recovery on behalf of failure-agnostic controller modules. Our approach is substantially different in that we develop a declarative language to let developers express fault-tolerance requirements and provide a compiler that targets OpenFlow fast-failover mechanisms.

The Flow-based Management Language (FML) [5] also addressed the problem of policy specification using a declarative language. FML does not express fault-tolerance policies. Our path regular expressions generalize the waypointing constraints of FML. Similarly, the NetPlumber verifier [6] uses a property specification language based on regular expressions on paths. Their language is used to verify network configurations, while FatTire generates configurations that are correct-by-construction.

NetCore [4, 12, 13] is an expressive language for specifying network forwarding configurations. Because it specifies forwarding in terms of hop-by-hop forwarding, it is difficult to express failure recovery. FatTire is a higher-level language built on top of NetCore that abstracts over network paths. Because FatTire compiles into NetCore, FatTire pro-

grams can be used as ordinary NetCore programs, and can be combined using the parallel and sequential composition operators offered in NetCore.

8. FUTURE WORK

We are currently working to enrich and expand the path expression language with abstractions that express more fine-grained fault-tolerance specifications such as shared risk link groups or non-uniform link reliability. In the future we would like to integrate existing failure-recovery and detection mechanisms (*e.g.*, [19]) into our system. Our language only deals with link-level failures—a switch-level failure can be modeled as a failure of each adjacent link. Adding first-class support for switch-level fault tolerance is future work. As with any failure-recovery solution, failover is only half of the remedy. We also plan to enrich our approach so that, after the failure information propagates to the controller, we recompute a new network-wide forwarding state that continues to guarantee the required fault-tolerance level while making better use of overall network resources (*e.g.*, redistributing traffic load) in response to encountered failures. Finally, we also plan to explore the application of FatTire's path abstractions to other domains such as expressing performance and QoS requirements, and using them with existing traffic engineering solutions.

Acknowledgements. We wish to thank the HotSDN reviewers for many insightful comments and suggestions. Our work is supported in part by the NSF under grants CNS-1111698, CCF-1253165, and CCF-0964409; the ONR under award N00014-12-1-0757; and by a Google Research Award.

9. REFERENCES

[1] CPqD OpenFlow 1.3 Software Switch, July 2012. `http://github.com/CPqD/ofsoftswitch13`.

[2] OpenFlow Switch Specification 1.3.1, March 2013. `http://bit.ly/of-131`.

[3] S. Bryant, S. Previdi, and M. Shand. A Framework for IP and MPLS Fast Reroute Using Not-via Addresses. *IETF Internet Draft*, June 2013. `http://datatracker.ietf.org/doc/draft-ietf-rtgwg-ipfrr-notvia-addresses/`.

[4] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-Verified Network Controllers . In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Seattle, WA*, June 2013.

[5] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical Declarative Network Management. In *ACM SIGCOMM Workshop on Research on Enterprise Networking (WREN)*, 2009.

[6] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking using Header Space Analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[7] James Kempf, Elisa Bellagamba, András Kern, David Jocha, Attila Takács, and Pontus Sköldström. Scalable Fault Management for OpenFlow. In *IEEE International Conference on Communications (ICC)*, 2012.

[8] Maciej Kuźniar, Peter Perešíni, Nedeljko Vasić, Marco Canini, and Dejan Kostić. Automatic Failure Recovery for Software-Defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Software-Defined Networking (HotSDN)*, August 2013.

[9] Amund Kvalbein, Audun Fosselie Hansen, Tarik Čičic, Stein Gjessing, and Olav Lysne. Multiple Routing Configurations for Fast IP Network Recovery. *IEEE/ACM Transactions on Networking*, 17(2):473–486, April 2009.

[10] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[11] Junda Liu, Aurojit Panda, Ankit Singla, P. Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring Connectivity via Data Plane Mechanisms. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[12] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A Compiler and Run-time System for Network Programming Languages. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 217–230, January 2012.

[13] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[14] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, March 2009.

[15] P. Pan, G. Swallow, and A. Atlas. Fast reroute extensions to RSVP-TE for LSP tunnels. *IETF RFC 4090*, 2005.

[16] Alex Raj and Oliver C. Ibe. A survey of IP and multiprotocol label switching fast reroute schemes. *Computer Networks*, 51(8):1882–1907, June 2007.

[17] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, August 2012.

[18] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. Enabling Fast Failure Recovery in OpenFlow Networks. In *International Workshop on the Design of Reliable Communication Networks (DRCN)*, 2011.

[19] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. Network Architecture for Joint Failure Recovery and Traffic Engineering. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2011.

[20] Jean-Philippe Vasseur, Mario Pickavet, and Piet Demeester. *Network Recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS*. Morgan Kaufmann, 2005.