Cementing High Availability in OpenFlow with RuleBricks

Dan Williams
IBM T. J. Watson Research Center
Yorktown Heights, NY

Hani Jamjoom
IBM T. J. Watson Research Center
Yorktown Heights, NY

ABSTRACT

Controller applications in OpenFlow cannot be trivially augmented to support the various high availability (or failure recovery) models of server applications. Recent work on OpenFlow has largely assumed static replica configurations or has relied on controller developers to embed high availability support in their design. Instead, we present *RuleBricks*, a system for flexibly embedding high availability support in existing OpenFlow policies. RuleBricks introduces three key primitives: *drop*, *insert*, and *reduce*. We describe how these primitives can express various flow assignment and backup policies, demonstrating the one offered by the Chord protocol. We have implemented RuleBricks and the Chord assignment policy. Using simulation, we compare RuleBricks against a typical tree-based approach. We show that RuleBricks maintains linear scalability with the number of replicas on the Chord ring.

Categories and Subject Descriptors

C.2.1 [**Computer Communication Networks**]: Network Architecture and Design

Keywords

High Availability, OpenFlow, Software-Defined Networking

1. INTRODUCTION

The flexibility of OpenFlow [8] should simplify supporting high availability models of server applications. Today, new breeds of network controllers are showcasing the potential of OpenFlow (e.g., load balancers [5, 13], random host mutations [6], record and replay debugging [14], and application acceleration [12]). However, tools to help encode or reason about high availability are scarce, despite the emergence of higher level language constructs like Frenetic [4] and Pyretic [9].

This paper focuses on one primary question: *how can high availability (HA) policies be added to OpenFlow's forwarding rules?* We assume an environment that consists of multiple server replicas connected to an OpenFlow-enabled network fabric (Figure 1). Additionally, the number of replicas can grow or shrink, as expected by today's cloud deployments [1]. Network support for HA, in general, requires that if a switch forwards a particular flow to specific replica, when the replica fails, then that flow is reassigned to a designated backup.[1] That said, we specifically focus on the challenges that arise in an elastic environment:

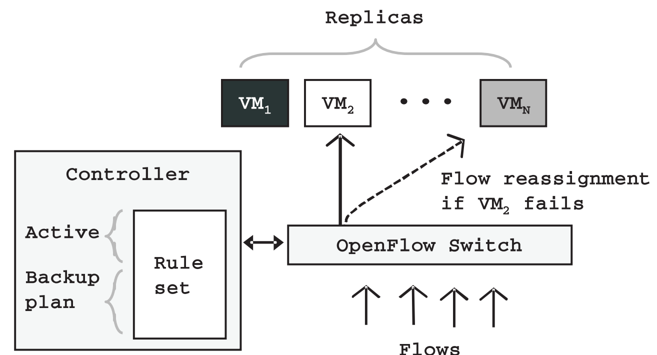

Figure 1: Assumed environment

- **Planning for failure.** Without adequate planning, the load from a dying replica may be spread unequally among the surviving replicas.
- **Limiting flow reassignment.** Flow reassignment introduces overhead in terms of reprogramming switches and any affinity violations (e.g., migrating flow-related state [10]).
- **Limiting rule explosion.** As replicas are created and destroyed, the number of rules required to specify flow assignments may become fragmented. If flow assignments cease to resemble the natural hierarchy of the IP address space, wildcard rules cannot be used to reduce the rule set.

We introduce a system, called *RuleBricks*, that can effectively embed a wide variety of HA policies into existing OpenFlow forwarding rules. RuleBricks augments a controller's forwarding rule set, which we refer to as the *active*

[1]We, thus, further assume a system is in place that enables the reassignment of flows to new replicas without corrupting session state (for example, by implementing the Split/Merge [10] abstraction).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotSDN'13, August 16, 2013, Hong Kong, China.
Copyright 2013 ACM 978-1-4503-2178-5/13/08 ...$15.00.

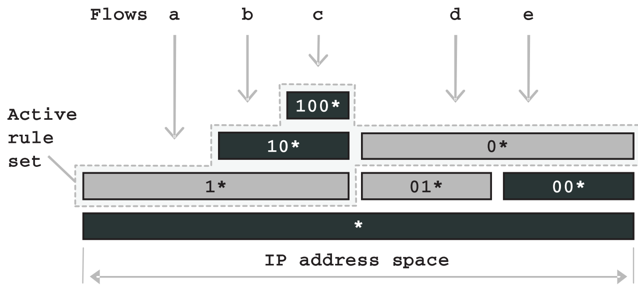

Figure 2: RuleBricks Overview

rule set, with a *backup plan*. The backup plan consists of the rules that get activated when one or more replicas die. In this paper, we focus on how the backup plan is created in a robust and efficient way.

Our work exploits two features in OpenFlow: (1) the hierarchical structure of OpenFlow's wildcard rules and (2) precedence rule execution. Conceptually, RuleBricks represents OpenFlow's forwarding rules as *bricks* covering the IP address space. Each brick directly maps to a single OpenFlow forwarding rule.[2] At the heart of RuleBricks are three brick primitives: *drop*, *insert*, and *reduce*. As the name implies, the drop primitive adds new bricks (active forwarding policies) on top of existing ones. The insert primitive adds new bricks directly underneath an existing policy to form a backup plan. Finally, the reduce primitive is a set of Tetris-like rule transformations that shrink the number of rules in the active and backup plans.

We demonstrate the expressiveness of RuleBricks by encoding the assignment policy introduced in the Chord [11] protocol, a popular distributed hash table (DHT) implementation. We show how RuleBricks can effectively encode the addition of both nodes and virtual nodes in a Chord ring. We also describe how the system reacts to failure.

We have implemented RuleBricks in Python. We have also implemented two variations of the Chord assignment policy: one that adheres to the hierarchical structure (rigid brick sizes) imposed by OpenFlow's prefix-based wildcard rules and one that implements flexible brick sizes. Using simulation, we measure the efficacy of RuleBricks on OpenFlow's rule table size and compare it against a typical tree-based approach. We show how RuleBricks maintains linear scalability with the number of replicas on the Chord ring and offers approximately 50% reduction in the active rule set when compared to a naïve tree-based implementation.

This paper is structured as follows. Section 2 details the design of RuleBricks. In Section 3, we show how RuleBricks can be used to encode Chord's assignment policy. In Section 4, we evaluate and discuss the efficacy of our system. Section 5 describes related work and Section 6 concludes.

2. RULEBRICKS

At the core of RuleBricks is a data structure that encodes both active and backup flow assignments to replicas in an elastic application. In this section, we describe the key characteristics of this structure in terms of its three primitives: *drop*, *insert*, and *reduce*.

[2]For simplicity, and without loss of generality, we only consider rules that match on source IP address.

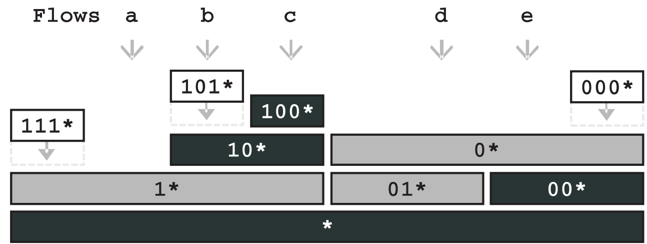

Figure 3: Example of the *drop* primitive

2.1 Drop: Encoding Active Rules

RuleBricks represents the flow assignment over the IP address space as a set of colored *bricks*. As depicted in Figure 2, each brick corresponds to a segment of the address space that can be identified using a prefix-based wildcard rule. The length of the prefix determines the width of the brick. All flows with source IP in the address range covered by a brick will be forwarded to the same replica; the color of the brick indicates which replica. Additionally, bricks can stack. The flow assignment is determined by the color of the top-most bricks. Stacked bricks correspond to overlapping rules, with the precedence encoded in the stacking order.[3]

Rules from exposed bricks at the top of the structure make up the *active rule set* (Figure 2). This is because flows will match their rules before the rules from any underlying (covered) brick. Therefore, the modification or addition of active rules consists of *dropping* new bricks onto the top of the structure, shown in Figure 3. Dropping new bricks effectively transfers the flows in the corresponding parts of the address space to the specified replica. In Figure 3, flow `b`, assigned to the black replica in Figure 2, is now assigned to a new replica (white). Alternatively, dropping bricks of existing colors redistributes the parts of the address space each replica covers (and likely the load on each replica).

The number of bricks required to achieve coverage of the entire address space—and therefore the number of rules that must be installed—is affected by the width of each brick. Larger bricks introduce stronger limits on the number of rules. For example, the widest brick is 32 bits wide and covers all IP-addresses; only one brick of this type—corresponding to the `*` rule—is required to achieve coverage. An 8 bit wide brick covers all addresses from a particular /24 network. For a brick to be represented in a single rule, the prefix-based wildcard rules—like those in OpenFlow—restrict the flexibility when selecting brick size and alignment. In particular, bricks must conform to a binary tree structure. For example, any 8 bit wide brick must be aligned to the "all zeroes" address in some /24 network.

2.2 Insert: Encoding Backup Rules

Any brick that is fully or partially covered by another brick is part of the backup plan. As shown in Figure 4, when a replica fails, all bricks bearing the color of the dying replica effectively become invisible. The underlying bricks are exposed and therefore become part of the active rule set. Any flows that previously fell on now-invisible bricks are reassigned to the underlying brick. In Figure 4, flows

[3]A more formal description is possible using set notation and set theory; however, we leave this for future work.

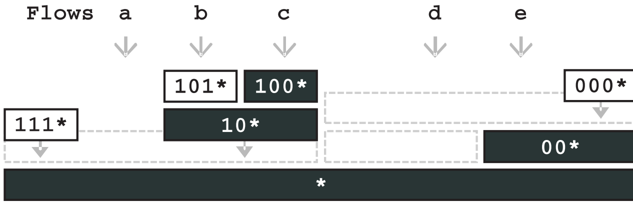

Figure 4: Reassignment on failure of the grey colored replica in Figure 2

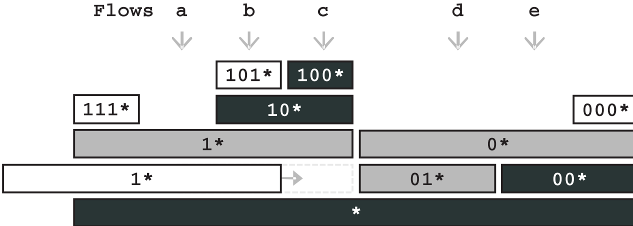

Figure 5: Example of the *insert* primitive

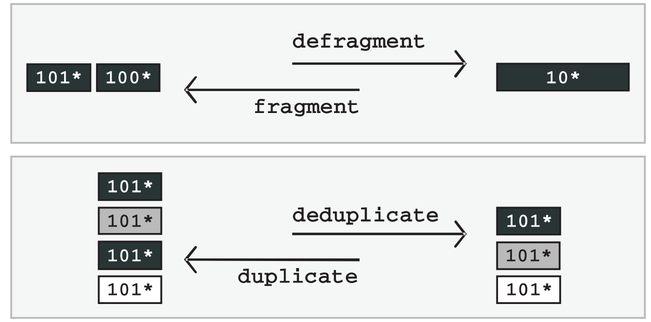

Figure 6: *Reduce* transformations

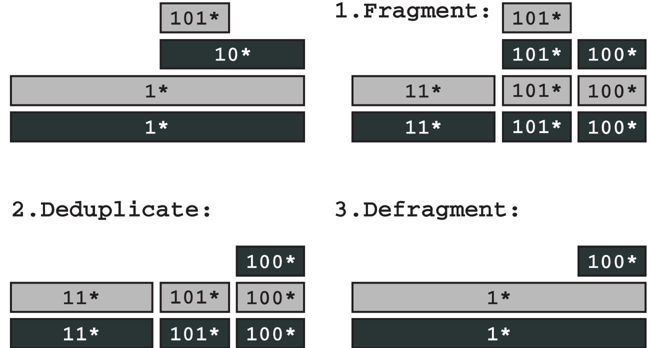

Figure 7: Example of *reduce* transformations

a, d, and e are all reassigned to the black replica when the grey replica is removed.

To plan for flow reassignment in case a replica fails,[4] additional layers of bricks can be introduced underneath bricks using the *insert* primitive. For example, in Figure 5, a white brick is inserted underneath a grey brick. In this case, if grey were to fail, flow a would be assigned to the white replica, while flows d and e would be assigned to the black replica. By inserting the white brick, better balance is achieved post-failure than in the scenario depicted in Figure 4.

We believe this structure enables the expression of non-trivial plans for flow assignment in the face of failure in an elastic workload. By selecting widths and colors of bricks, and dropping or inserting them appropriately, a network controller can navigate the tradeoff between load balancing objectives, number of flow reassignments, and rule explosion. We discuss an example policy structure in detail in Section 3.

2.3 Reduce: Towards Efficient Encoding

RuleBricks implements a *reduce* primitive to help eliminate redundancy in the brick structure that stems from replicas coming and going in an elastic application. The reduce primitive is built from two brick transformations, depicted in Figure 6. Two horizontal bricks of the same color can be *defragmented*, or merged together into a larger brick. Defragment can only be performed on bricks whose wildcard rules are identical except for the least significant bit of the prefix. In other words, defragmented bricks cannot violate the binary tree property discussed in Section 2.1. Two vertical bricks of the same color can be *deduplicated*, in which the underlying brick is removed. Deduplication does not affect the policy, because the top brick will always be exposed before a brick of the same color below it.

Counter to intuition, the inverse of each transformation (fragment and duplicate in Figure 6) are also useful for the reduce primitive. For example, Figure 7 depicts a simple example in which a sequence of fragmentation, deduplication, and defragmentation are applied to reduce redundancy in the rule structure. Examples can also be constructed in which duplication ultimately leads to a reduction in the rule structure.

We are exploring algorithms to automatically apply the two transformations to an arbitrary rule structure. To date, we have implemented an algorithm that heuristically defragments as many rules as possible, introducing duplication if necessary. Then the algorithm deduplicates bricks, starting with the bricks with the least exposure. Developing an optimal algorithm is a subject of future work.

3. CHORD: AN EXAMPLE POLICY

To demonstrate how RuleBricks can be used to plan for failure, we next describe the encoding of an example flow assignment policy. Specifically, we encode a scheme similar to that popularized by distributed hash tables (e.g., Chord [11]) and adapted in systems like Dynamo [2].

3.1 Background

In a distributed hash table (e.g., Chord), participating nodes are each assigned a random identifier in a 32-bit identifier space. The id space is thought of as a ring. Objects are stored in the DHT by one (or more) of the participating nodes. To select the node in which an object should be stored, a hash function is computed over the object. The resulting hash results in a location on the id space ring. The object is stored at the node with the closest id clockwise around the ring (the successor) from the hash location.

If a node is removed, all objects that were mapped to that node will be naturally mapped to the node's successor. Similarly, when a new node is added, some of the objects

[4]Again, we assume the appropriate state management or replication processes are in place for flow reassignment.

mapped to the new node's successor will now be mapped to the new node, since the new node shrinks the address space covered by its successor.

Balanced load is achieved by introducing multiple "virtual nodes," each with its own random identifier on the ring, for each participating node. This reduces the probability that any one node will store objects for a disproportionately large subset of the id space and enables load from a failing node to be spread across multiple surviving nodes. The properties of virtual nodes and the design for nodes that come and go make the Chord policy attractive for flow assignment between replicas in an elastic environment.

3.2 Encoding

Figure 8 depicts the encoding of a Chord-like policy in RuleBricks. As in Chord, each replica is responsible for objects that map to a portion of the address space. In this case, objects are flows, and the address space is the hierarchical IP address space.[5] When only one replica (white) is in the system, one brick is dropped to cover the entire address space (Figure 8(a)).

When a new replica (black) is created, it is assigned a random identifier on the ring. To encode a Chord-like policy, any active flows with the black replica id as a direct successor should be mapped to the black replica. Furthermore, if either replica were to fail, the other replica should inherit all flows. In the example in Figure 8(b), the black replica is assigned the address with the prefix `01`, followed by zeroes. For active flows, all addresses prefixed with `00` are reassigned by dropping a black `00*` brick. The backup rules for the case in which black fails are automatically covered because the white brick is underneath. An additional black brick is inserted underneath the white brick to provide backup rules in case white fails.

Figure 8(c) shows the arrival of another replica (grey) in the system. This time, two active bricks (`100*` and `01*`) are dropped. With these rules, the grey replica becomes responsible for part of the address space covered by white, with automatic backup back to white. Two backup bricks (`00*` and `*`) are also inserted, to ensure that the Chord-like policy will be enforced if black fails.

Virtual nodes are implemented identically to new replicas, except they share a color with other virtual nodes for the same replica. Figure 8(d) shows the addition of a white virtual node. One active brick (`01*`) is dropped and two backup bricks are inserted (`00*` and `*`). Figure 8(e) shows the result of reducing the structure by performing a sequence of operations from Figure 6.

The number of bricks required to specify an arbitrary address range can vary greatly because of the restriction that bricks must be able to represented as part of a binary tree. An address range that can be represented using bricks of larger sizes (closer to the root of the binary tree) will require fewer rules. In the next section, we evaluate brick size and its effect on the rule set size for Chord-like policies.

4. EVALUATION AND DISCUSSION

We have implemented RuleBricks in Python. As a flow assignment policy, we have implemented two variants of the

[5]Hash-based rules are not yet standard in OpenFlow. The implications of a non-uniform mapping of flows in the address space are discussed in Section 4.

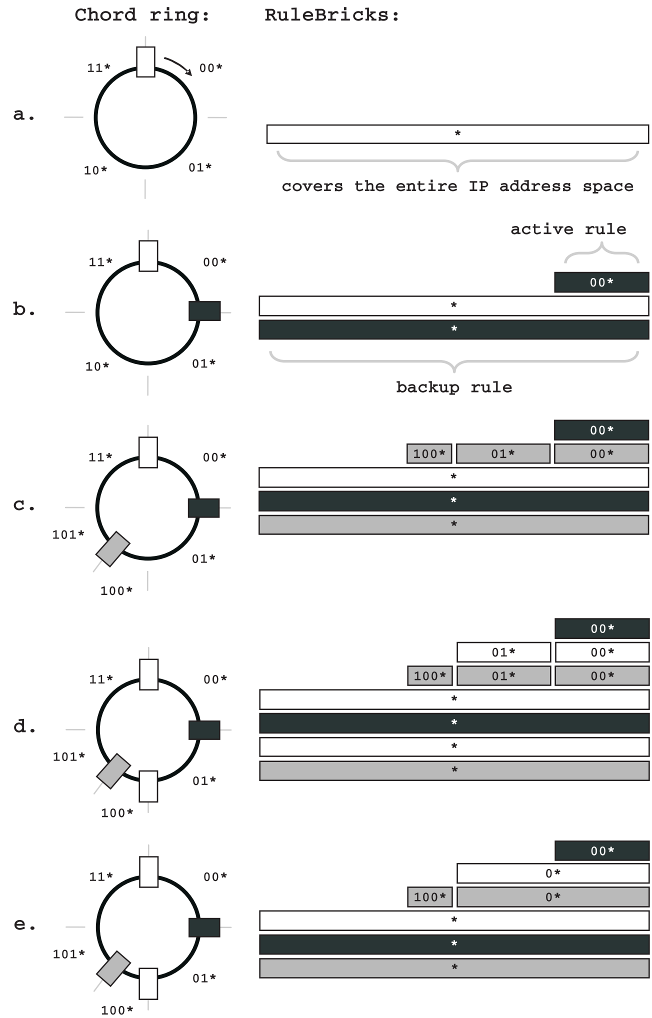

Figure 8: Encoding Chord nodes in RuleBricks. (a), (b), and (c) introduce three replicas; (d) introduces a virtual node; (e) shows rules after *reduce*.

Chord example described in Section 3. The first variant allows any point on the chord ring to be chosen as the id of a virtual node. This variant, called *variable-sized RuleBricks*, results in bricks of odd sizes that may not align well with the hierarchical structure imposed by prefix-based wildcard rules. The second variant exploits RuleBricks by limiting the minimum brick size based on the number of replicas in the system. Limiting the brick size effectively limits the choice of virtual node id in Chord. The resulting rules fit well into the hierarchical structure imposed by prefix-based wildcard rules, resulting in predictable scaling in the rule set. We refer to this variant as *fixed-sized RuleBricks*.

Linear scalability in the number of active rules when using fixed-sized RuleBricks is demonstrated in Figure 9. This figure shows the number of rules used to partition the address space between replicas using 16 virtual nodes per replica. To demonstrate the role of rule reduction in RuleBricks, we compare the number of rules to a naïve tree-based strategy. Like the fixed-sized RuleBricks scenario, in the tree-based strategy, every virtual node covers part of the address space

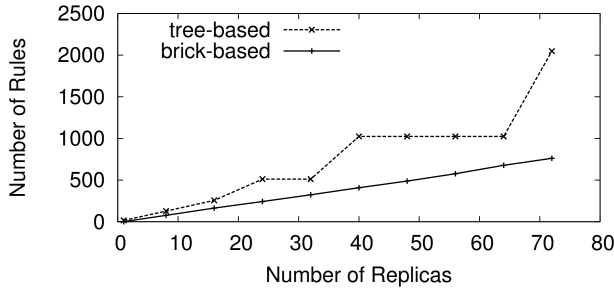

Figure 9: Brick-based rule reduction

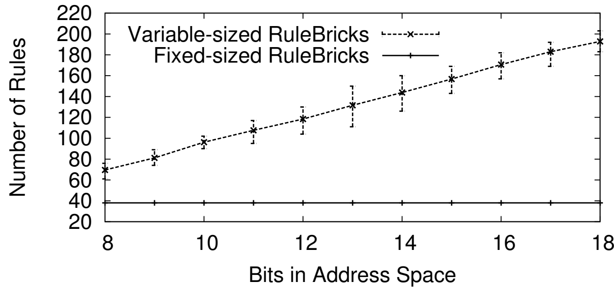

Figure 10: Limiting rule set with fixed-size RuleBricks

using a rule corresponding a fixed level of the tree, depending on the number of replicas. By applying the rule reduction transformations described in Section 2.3, RuleBricks saves 51% of the rules on average.

To demonstrate how using fixed-sized RuleBricks yields a predictable number of active rules, we experiment with 10 replicas with 4 virtual nodes each. Figure 10 shows the number of active rules to support the Chord-like flow assignment as a function of the number of bits in the address space. The use of fixed-sized RuleBricks decouples the number of active rules from the address space size. Variable-sized RuleBricks suffers, both in number of active rules and variability (the error bars depict the average, maximum, minimum for 10 trials). To cover the 32-bit range of IPv4 addresses, the active rule set may become infeasible to program into switches.

It should be noted that, in both of these strategies, the number of rules in the system are independent from the number of flows in the system. However, the effectiveness of the address space partition depends on the distribution of flows in the system.

Fixed-sized RuleBricks result in a smaller, more predictable active rule set. However, a manageable rule set is not without consequence. As shown in Figure 11, fixed-sized RuleBricks guarantee a degree of balance in the address space partitioning due to their adherence to the hierarchical structure imposed by prefix-based wildcard rules. This is optimal only when flows are uniformly distributed throughout the address space. Not surprisingly, for highly-skewed address space distributions, the more flexible variable-sized RuleBricks achieve better load balancing.

5. RELATED WORK

While high availability is a well established research area, its implication on Software Defined Networks (SDN) and

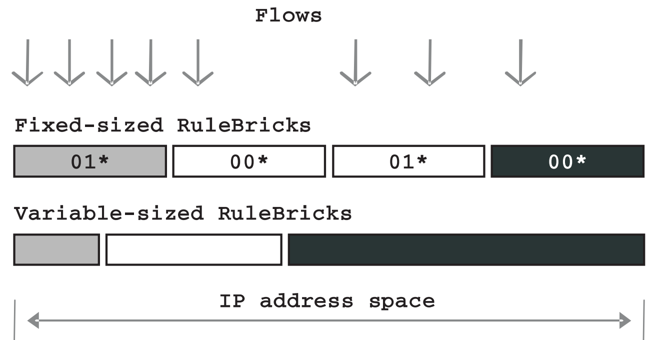

Figure 11: Effects of fixed vs. variable brick sizes on flow assignment

OpenFlow has been limited to implementing highly available controllers [7]. This paper looks at HA from a different perspective. We investigate how OpenFlow can support HA policies of end applications.

Broadly, there are two areas of related work. The first studies the use of OpenFlow to implement middlebox functionality, like load balancers [5, 13] or random host mutations [6]. Load balancers are interesting because they distribute flow assignments across server replicas. Wang et al. [13] focus on exploiting the wildcard prefix characteristics of OpenFlow to minimize flow rules. RuleBricks focuses on the construction of arbitrary HA policies, while similarly trying to exploit OpenFlow's wildcards (and precedence rules) to minimize flow rules.

The second area looks at creating high level language constructs to simplify the programmability (encoding) of network policies. Examples of such languages include Frenetic [4], its extension, Pyretic [9], and Hierarchical Flow Tables [3]. In particular, Pyretic, which focuses on the composability of SDNs, offers a compelling approach to building network policies. Unlike RuleBricks, it does not focus on offering a simple approach to encode backup plans, especially in existing forwarding policies.

6. CONCLUSION

Planning for failure is as important as planning for elasticity in today's network environments. RuleBricks addresses the need for failure-planning in OpenFlow networks through an expressive brick-based data structure. To date, we have implemented the Chord assignment policy and begun to explore the implications of brick size and the reduce primitive. As more flow assignment policies are implemented in RuleBricks, we expect RuleBricks to: (1) expose the potential for rule explosion through brick-size restrictions, and (2) offer a "toolbox" of transformations from which optimal active rule sets and backup plans can be automatically derived. Ultimately, we hope RuleBricks leads to the use of increasingly flexible and scalable flow assignment policies.

7. REFERENCES

[1] Auto Scaling. `http://aws.amazon.com/autoscaling/`.

[2] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. *SIGOPS OSR 41*, 6 (2007), 205–220.

[3] Ferguson, A. D., Guha, A., Liang, C., Fonseca, R., and Krishnamurthi, S. Hierarchical Policies for Software Defined Networks. In *Proc. of ACM HotSDN* (Helsinki, Finland, Aug. 2012).

[4] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. Frenetic: A Network Programming Language. In *Proc. of ACM ICFP* (Tokyo, Japan, Sept. 2011).

[5] Handigol, N., Seetharaman, S., McKeown, N., and Johari, R. Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow. In *Proc. of ACM SIGCOMM* (Spain, Aug. 2009). (Demo).

[6] Jafarian, J. H., Al-Shaer, E., and Duan, Q. OpenFlow Random Host Mutation: Transparent Moving Target Defense using Software Defined Networking. In *Proc. of ACM HotSDN* (Helsinki, Finland, Aug. 2012).

[7] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., , Hama, T., and Shenker, S. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. of USENIX OSDI* (Vancouver, Canada, Oct. 2010).

[8] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR 38*, 2 (Apr. 2008).

[9] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing Software Defined Networks. In *Proc. of USENIX NSDI* (Lombard, IL, Apr. 2013).

[10] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. of USENIX NSDI* (Lombard, IL, Apr. 2013).

[11] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM* (San Diego, CA, Aug. 2001).

[12] Wang, G., Ng, T. S. E., and Shaikh, A. Programming Your Network at Run-time for Big Data Applications. In *Proc. of ACM HotSDN* (Helsinki, Finland, Aug. 2012).

[13] Wang, R., Butnariu, D., and Rexford, J. OpenFlow-based Server Load Balancing Gone Wild. In *Proc. of USENIX HotICE* (Boston, MA, Mar. 2011).

[14] Wundsam, A., Levin, D., Seetharaman, S., and Feldmann, A. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *Proc. of USENIX Annual Technical Conf.* (Portlan, OR, June 2011).