Parallel Consistent Network Updates

Nayden A. Nedev

Master's Thesis

In Partial Fulfillment of the Requirements
for the Master of Science in Engineering
Department of Computer Science
Princeton University

Adviser: David P. Walker

June 2013

Abstract

Network configuration changes are a frequent operation performed by network administrators. Unfortunately, these changes can result in a wide range of problems such as network outages, security vulnerabilities or worse overall network performance.

A recent result in this area proposes the notion of consistent network update - an update that preserves certain properties when updating from one network policy to another. The authors of the mentioned work describe a per-packet consistent update algorithm that guarantees that every packet in the network traverses either the old policy or the new one but not some mixture of the two.

With the advent of software-defined networking, development of centralized, tightly-coordinated applications with strong global correctness properties is now possible. However, it is an open problem how to bring more parallelism to these applications and how to leverage general-purpose multicore and multiprocessor machines to improve their performance. In this work, we propose two parallelized versions of the per-packet consistent update mechanism. We implement them on top of Floodlight - a new multi-threaded software-defined networking controller platform. We evaluate their performance on two example applications - a network-wide host location learning and a shortest-path routing application.

Acknowledgements

I would like to thank all the people who gave me feedback for earlier drafts of this work or attended my practice talks: Prof. David Walker, Gordon Stewart, C.J. Bell, Cole Schlesinger, Joey Dodds, Lennart Beringer, Pramod Subramanyan and all the people in WRI502. I also would like to thank Prof. David Walker for the provided funding.

Contents

Abstract iii

Acknowledgements iv

List of Tables vii

List of Figures viii

1 Introduction **1**

1.1 Background on Software-defined Networking 1

1.2 Consistent Network Updates 3

1.3 Main Contributions 5

2 Related Work **6**

3 Modified Consistent Updates **8**

3.1 Classical Algorithm 8

3.2 Parallel Version 9

3.2.1 Basic Parallel Version 9

3.2.2 Alternative Parallel Version 10

3.3 Synchronization 11

3.4 Other Possible Optimizations 12

4 Implementation **13**

4.1 Floodlight Controller 13

4.2 Basic Design of the System 14

4.3 Provided Functionality 16

5 Applications **17**

5.1 Network-wide Host Learning 17

5.2 Shortest-path Routing 20

6 Experimental Results **23**

6.1 Experimental Setting 23

6.2 Results 24

7 Future Work **27**

8 Conclusion **28**

Bibliography **29**

List of Tables

4.1 Functionality Provided by the Updates Mechanism 15

4.2 Functionality Provided by Policy 16

List of Figures

1.1 A Software-defined Network 2

3.1 Classical Consistent Updates Algorithm 9

3.2 Parallelized Consistent Updates Algorithm 10

3.3 Alternative Parallelization of Consistent Updates Algorithm 11

4.1 Main Components of the Updates System 15

5.1 *packet_in* Handler of Host-location Learning Application 18

5.2 Sequential Breadth-First Search Algorithm 19

5.3 Parallel Breadth-First Search Algorithm 20

5.4 *packet_in* Handler of Shortest-path Routing Application 21

5.5 Sequential Single-Source Shortest Path Algorithm 21

5.6 Parallel Single-Source Shortest Path Algorithm 22

6.1 Scalability of the Proposed Parallel Algorithms 25

6.2 Scalability of Parallel Breadth-First Search 26

Chapter 1

Introduction

1.1 Background on Software-defined Networking

Software-defined networking (SDN) is a new approach for building computer networks that completely redesigns the way networking devices interact with the software that controls them. This is achieved by decoupling the system that makes decisions about where traffic is sent (the control plane) from the underlying system that executes these decisions in practice (the data plane). This new paradigm evolved from ideas that appeared in previous foundational systems called Ethane [7] and 4D [13].

A software-defined network consists of two main components - a centralized general-purpose machine called *controller* and a number of *switches*. The main purpose of the controller is to install low-level forwarding and modification rules on the switches in the network. In contrast to the controller, switches are quite simple. They can only forward and modify packets according to the rules installed on them. All these design decisions make network management substantially less complicated and error-prone. Figure 1.1 shows a diagram of a traditional software-defined network. The machine in red is the controller and the devices in blue are the switches. Each switch is connected by a secure channel to the controller.

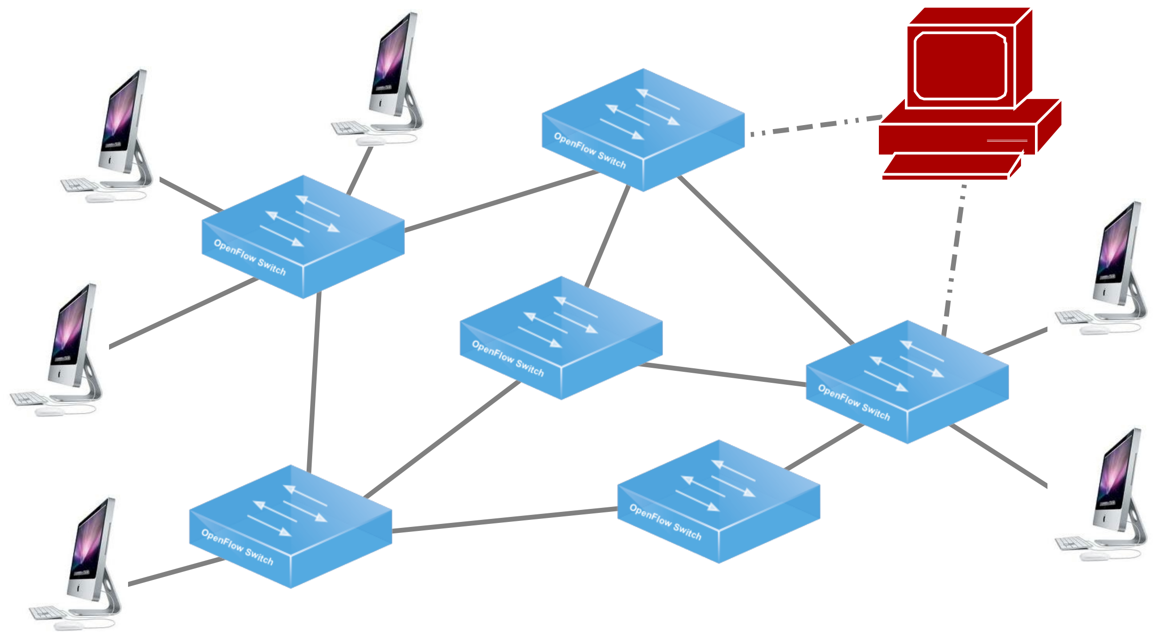

Figure 1.1: A Software-defined Network

Each packet flying through the network can be processed either by a switch or by the controller. A packet is processed by a switch if there is a installed rule on the switch that matches the packet. Then the packet is forwarded or modified according to the matching rule. If there is no such rule on the switch, the packet is sent to the controller. The controller has a complete knowledge about the whole network and the network policy, so it can decide where the packet should go. However, processing a packet by the controller is orders of magnitude slower than processing it by a switch. So the main aim in designing a software-defined network is that most of the packets will be processed by the switches and only some special packets, such as the first packet of a flow, will be processed by the controller.

There has been an increasing interest in software-defined networking, both from industry and academia, for the last several years. It has become hot research topic in many systems and programming languages groups at top universities around the world. Google and Facebook have already deployed SDN-based technologies in their data centers. Moreover, several companies, including Facebook, Google, Yahoo! and

Microsoft, founded the Open Networking Foundation. The main aim of the foundation is to promote and widely adopt software defined networking through open standards development. Software-defined networking has also been commercialized by a couple of successful startup companies. In addition, many switch vendors support OpenFlow [20, 4], a concrete realization of a protocol for communication between an SDN controller and switches. All these facts suggest that software-defined networking will become even more widely adopted in both industrial and academic environments.

1.2 Consistent Network Updates

Nowadays, different kinds of network updates are a very common operation performed by network operators. Updates are an integral part of many important network applications such as stateless and stateful firewalls, traffic monitoring and engineering, virtual machine migrations in data centers or planned network maintenance. For example, in a stateless firewall changing access control list can be viewed as an update operation. In traffic engineering adjusting a link weight is such an operation. While doing virtual machine migration, moving a server from one location to another is a type of configuration change. These are only a few examples among many others that show the high frequency of different update operations in modern network administration.

Certain behaviors, while transitioning from one configuration to another, should be preserved. Unfortunately, it is tremendously hard to maintain these behaviors during updates because the network administrator should consider all different interactions between the update operation and the packets flowing through the network. A recent post [2] in a popular networking blog very clearly shows these difficulties when performing a large-scale firewall upgrade. It also proposes a solution based on software-defined networking. This application is one of many such examples which need a general solution. Finding an efficient general solution for this task would reduce the amount of network instability, security vulnerabilities and lower overall network performance.

Researchers have proposed a number of solutions for these problems that are limited to a concrete network protocol. In order to come up with a more general solution, Reitblatt *et al.* [22] formalize the notion of a network update and propose an algorithm for *per-packet consistent network update.* That is, given two network policies P_1 and P_2 and a per-packet consistent update from the first to the second, then every packet in the network is forwarded according to either P_1 or P_2. Moreover, no packet is forwarded according to any mixture of P_1 or P_2.

Although the algorithm of Reitblatt *et al.* is a huge improvement over previous algorithms, there are still some problems with it. More specifically, it can be quite slow for bigger networks with a large number of switches. It goes through all the switches in the network and updates each of them one by one. This operation can take a substantial amount of time. Given that network updates are a quite frequent operation, this problem can limit the flexibility of applications that use consistent network updates by several orders of magnitude. Moreover, the algorithm is specified in terms of low-level rules and switch ports. It is more convenient to think of a network policy as a set of higher-level abstractions such as paths. So there is quite a lot of room for improvement over the classical per-packet consistent update algorithm.

The usage of general-purpose machines as controllers in a software-defined network allows many general approaches for building software to be applied in the context of networks. With the advent of commodity multicore and multiprocessor computer systems, it becomes possible to exploit thread-level parallelism in ordinary applications, such as software-defined networking ones. Unfortunately, parallelism naturally arises in problems from domains such as scientific computing or graphics, but rarely in systems. The consistent updates application is not an exemption from this rule. The approach of our work is to come up with a new way of expressing network updates that increases the amount of parallelism.

1.3 Main Contributions

The main contributions of this thesis are the following:

- A modification of the classical consistent updates algorithm that allows manipulation of whole paths in the network, rather than low-level rules.
- Two parallel algorithms for consistent network updates - one direct extension of the classical consistent updates algorithms and one based on the new path abstraction.
- Prototype implementation of the proposed algorithms in a real system on top of the Floodlight controller platform.
- Evaluation of the proposed parallel algorithms on two sample applications - a network-wide host location learning and a shortest-path routing applications.

The rest of this thesis is organized as follows. We review the related work in Chapter 2. We present the algorithms that we derived in Chapter 3 and our prototype implementation in Chapter 4. We describe two applications based on the invented algorithms in Chapter 5. The results of our experiments with these applications on a popular network topology are described in Chapter 6. We discuss several future directions in Chapter 7 and conclude in Chapter 8.

Chapter 2

Related Work

The algorithm for per-packet consistent network update was initially proposed by Reitblatt *et al.* [22]. Along with the algorithm for per-packet consistency, the authors present an analogous per-flow consistent update mechanism, a mathematical model that captures the essential behavior of software-defined networks, a tool for formal verification of network configurations, a prototype implementation of the proposed algorithms on top of NOX [14] and evaluation of the prototype on a number of different benchmarks.

Algorithms for different kinds of updates, which preserve certain types of properties, have been proposed in the literature. Most of them are designed for specific network protocols. Francois *et al.* [12] present technique for avoiding transient micro loops that can occur during the convergence of link-state interior gateway protocols such as IS-IS or OSPF. Again Francois *et al.* [11] describe a solution for the problem of Loss of Connectivity when an eBGP peer link is shut down by an operator during maintenance. Raza *et al.* [21] leverage dynamic programming and ant colony optimization techniques in order to find an optimal sequence of update operations that minimizes the overall performance disruptions. Recent work by Vanbever *et al.* [24] proposes a solution for elimination of forwarding loops that are caused by various IGP migration scenarios. Such scenarios include protocol replacement, hierarchy modifications and route summarization.

Consensus routing [16] proposes a solution for elimination of errors such as routing loops and blackholes due to BGP updates. The key insight in consensus routing is to sacrifice responsiveness in return for consistency. It improves overall availability when combined with several existing heuristics. Another notable work in the context of BGP is BGP-LP [18], which is a consistency model for BGP updates. This solution can be viewed as a per-packet update mechanism designed particularly for the BGP protocol.

Ajmani *et al.* [5] discuss the problem of general software updates in a distributed system. The authors propose a methodology for reasoning about the correctness of a distributed system with installed software with several different versions. Their approach also aims to limit service disruptions while doing a software update. This work considers the update of general-purpose machines and not the update of network switches, as we and Reitblatt *et al.* do.

Zhang *et al.* [26] formalizes and perform a theoretical analysis of various safety properties in firewall policy deployment. The authors also present several ways to make efficient firewall policy updates while preserving certain secure behaviors, such as constantly rejecting illegal traffic. This work is limited to only one device which is a general-purpose machine.

A number of different controller platforms [14, 17, 25, 10, 1, 23, 3] have been released over the last several years. Most of them have well-expressed drawbacks that make them hard to use. Such drawbacks include low-level languages for programming their platform and lack of multi-threading support. This thesis works with Floodlight [3] which is one of the few multi-threaded platforms along with its predecessor Beacon [1] and NOX-MT [23].

Chapter 3

Modified Consistent Updates

3.1 Classical Algorithm

The idea of consistent updates was initially proposed by Reitblatt *et al.* [22]. In order to make this thesis self-contained, we will describe their algorithm here. First, we will define the needed terminology. An *ingress port* is defined as a switch port that is connected to another switch port which is not part of the network (*e.g.* a switch port from another network) or to a host. An *internal port* is defined as a switch port that is connected to another *internal port*. That is, an internal port is a switch port connected to another switch port in the same network. Moreover, an internal port is never connected to a host.

Basically, the per-packet consistent update mechanism works as follows. First, the controller installs the rules representing the new network policy on the internal ports. These newly installed rules are enabled only for packets with a different version number. After that, it installs rules representing the new configuration on the ingress ports which also mark the incoming packets with a new version number. This action makes the new policy visible to the incoming packets. The complete algorithm can be described with the pseudocode shown on Figure 3.1.

```
Network n = /* read network topology information */
Policy policy = /* read next network configuration */
unsigned v = getNextVersion();
for (Switch s : n.getSwitches()) {
  for (Port p : s.getInternalPorts()) {
    for (Rule r : policy.getPortRules(p)) {
      r = enableRule(r, v);
      n.install(s, r);
} } }
for (Switch s : n.getSwitches()) {
  for (Port p : s.getIngressPorts()) {
    for (Rule r : policy.getPortRules(p)) {
      r = mark(r, v);
      n.install(s, r);
} } }
```
Figure 3.1: Classical Consistent Updates Algorithm

3.2 Parallel Version

The classical algorithm described in the previous section is entirely sequential. In this section, we describe two parallel algorithms for the same problem: one completely based on the already described sequential algorithm and one that divides the network policy into paths and uses them as a basis for efficient parallelization.

3.2.1 Basic Parallel Version

Looking at the classical algorithm more closely, we can see that it basically goes through all of the switches and updates each one of them. It first goes through the internal and then through the ingress ports of each switch. It can be also seen from the pseudocode on Figure 3.1 that while updating the internal or ingress ports, the controller never works on two switches simultaneously. These two observations suggest a potential improvement of the algorithm by using multiple threads (or processes) to work on different switches at the same time. Pseudocode of the resulting algorithm is shown on Figure 3.2.

```
Network n = /* read network topology information */
Policy policy = /* read the next network configuration */
unsigned v = getNextVersion();
for (Switch s : n.getSwitches()) {
  fork {
    for (Port p : s.getInternalPorts()) {
      for (Rule r : policy.getPortRules(p)) {
        r = enableRule(r, v);
        n.install(s, r);
} } } }
/* wait for all threads to finish */
for (Switch s : n.getSwitches) {
  fork {
    for (Port p : s.getIngressPorts()) {
      for (Rule r : policy.getPortRules(p)) {
        r = mark(r, v);
        n.install(s, r);
} } } }
/* wait for all threads to finish */
```

Figure 3.2: Parallelized Consistent Updates Algorithm

3.2.2 Alternative Parallel Version

The main problem with the parallel algorithm shown on Figure 3.2 is that it involves a fair amount of waiting. That is, if we have a really short path between two hosts, it can be used after the whole network update has been done. Using this observation, we have parallelized the algorithm on the basis of paths, rather than on the basis of switches and low-level rules. These paths are tuples of a predicate describing the packets that flow along the path and an ordered list of switches that represent the path. It results in the parallel algorithm whose pseudocode is shown of Figure 3.3.

```
Network n = /* read network topology information */
Policy policy = /* read the next network configuration */
unsigned v = getNextVersion();
for (Path path : policy.getPaths()) {
  fork {
    for (Switch s : path.getSwitches()) {
      for (Port p : s.getInternalPorts()) {
        for (Rule r : policy.getPortRules(path, p)) {
          r = enableRule(r, v);
          n.install(s, r);
    } } }
    Switch s1 = path.getFirstSwitch();
    Switch s2 = path.getLastSwitch();
    for (Switch s : {s1, s2}) {
      for (Port p : s.getIngressPorts()) {
        for (Rule r : policy.getPortRules(path, p)) {
          r = mark(r, v);
          n.install(s, r);
} } } } }
/* wait for all threads to finish */
```

Figure 3.3: Alternative Parallelization of Consistent Updates Algorithm

3.3 Synchronization

One of the most important things in the design of an efficient parallel algorithm is the correct placement of synchronization in it. The really good thing about the first proposed algorithm is that it does not require adding a lot of explicit synchronization. At each moment of its execution, only one thread is modifying the state of a certain switch. All the rules and ports are associated with a concrete switch, so a rule or a port is accessed by exactly one thread at a time. These facts mean that explicit synchronizations are not needed to protect the modification of the state of a given switch. However, a programmer should synchronize the concurrent accesses to the global network policy which can be concurrently modified by another thread that adds new rules to the policy. That is, synchronization is needed for lines 7 and 15 in the code on Figure 3.2.

In contrast to the first method of parallelization, the algorithm shown on Figure 3.3 requires bigger amount of extra synchronization. The implementation should maintain consistent the state of the switches, while they are modified by different threads, because switches are shared between different paths. This should be done on lines 10 and 18 in the code on Figure 3.3. It could be done by maintaining a lock for each switch. An alternative place to synchronize the access to each switch is in the controller mechanism that installs rules on the switches. This approach is dependent on the implementation of this mechanism and allows a wider use of lock-based and lock-free synchronization techniques.

3.4 Other Possible Optimizations

A number of optimizations can be applied in some special cases of the new configuration. One such case is when the new configuration only adds new flows or paths to the old one. Reitblatt *et al.* [22] call this case *pure extension.* When updating to *pure extension*, the only needed thing is to install the newly added rules on the switches. Moreover, the same version number can be used for them.

Another case, in which an optimization is possible, is when a number of rules have been removed from the policy. Reitblatt *et al.* [22] call it *pure reduction.* An update to *pure reduction* can be implemented by first updating the ingress ports, waiting for all packets in flight to reach their destinations and then update the internal ports.

These techniques can be applied at the level of paths instead at the level of whole policies. The updates mechanism finds all paths that are added, removed or modified and updates each one of them. Unfortunately, in the general case the update transforms into an ordinary network-wide update and the benefits of optimizations are lost.

Chapter 4

Implementation

4.1 Floodlight Controller

The whole implementation of the algorithms described in this thesis is on top of Floodlight [3]. Floodlight is a Java-based OpenFlow controller platform. There are three main reasons for this choice:

- By the time of the beginning of this work, Floodlight was one of the few multi-threaded controller platforms along with Beacon [1] and NOX-MT [23]. We needed such feature, because one of the main objectives of this thesis is to bring parallelism to the consistent updates algorithm and to software-defined networking in general. NOX-MT was not publicly available then and also Floodlight has many improvements over Beacon.

- Floodlight is produced and supported by an industrial vendor. It gives us the comfort of using well-tested, well-documented and stable source code, rather than an unstable prototype implementation.

- All Floodlight libraries are written in Java, which brings the opportunity of using standard Java features such as object-oriented programming paradigm, generics, sophisticated development environments, *etc.*

4.2 Basic Design of the System

Figure 4.1 shows a diagram of all the components of the built system and their interactions. Its first main component is the Policy class, whose functionality is shown on Figure 4.2. The Policy class maintains a global network policy, visible to all the applications in the controller. It also can be modified and queried by both applications running on the controller and clients out of the controller. It provides an interface, via Floodlight's RestAPI, by which outside clients can invoke all the methods shown on Figure 4.2. As in the classical algorithm, low-level rules can be added and removed from the policy. In addition, the Policy component provides functionality for adding and removing whole paths. As mentioned in Chapter 3, the paths can be specified as a tuple of a predicate describing the packets flying on the path and a list of the nodes in the path. Predicate can include information like source and destination IP or MAC addresses and input switch port.

Given this description of policy as a set of paths, it is much easier for the client to specify its policy. Moreover, it is easier to do the optimizations described in the previous chapter.

Another main component is the Updates class which executes the consistent updates algorithm on the network. It reads the current state of the Policy class and installs it on the network. The functionality of the Updates class is shown on Figure 4.1. As can be seen, it can execute both the sequential and parallel versions of the consistent updates mechanism.

All new threads are started and managed by a *newFixedThreadPool* instance of the *Executors* class in the Java concurrent library. The reason for choosing this thread pool is twofold. First, it immediately assigns a thread for a task after its submission to the thread pool. Second, it allows creating a thread pool with a explicitly specified number of threads, which is needed for our experiment, described in Chapter 6.

The synchronization mechanisms used in the implementation are based on locking schemes. An intrinsic lock using the *synchronized* language construction in Java is used for synchronization of the concurrent accesses to each switch in the path-based parallel consistent update algorithm. Also, the same method is used for synchronization of the concurrent accesses to the policy class. There is a lock on the getRules() method in class Policy. It could be improved by providing synchronization inside the Policy class instead of putting locks on the method, but as mentioned in Chapter 7, we leave it as a future work.

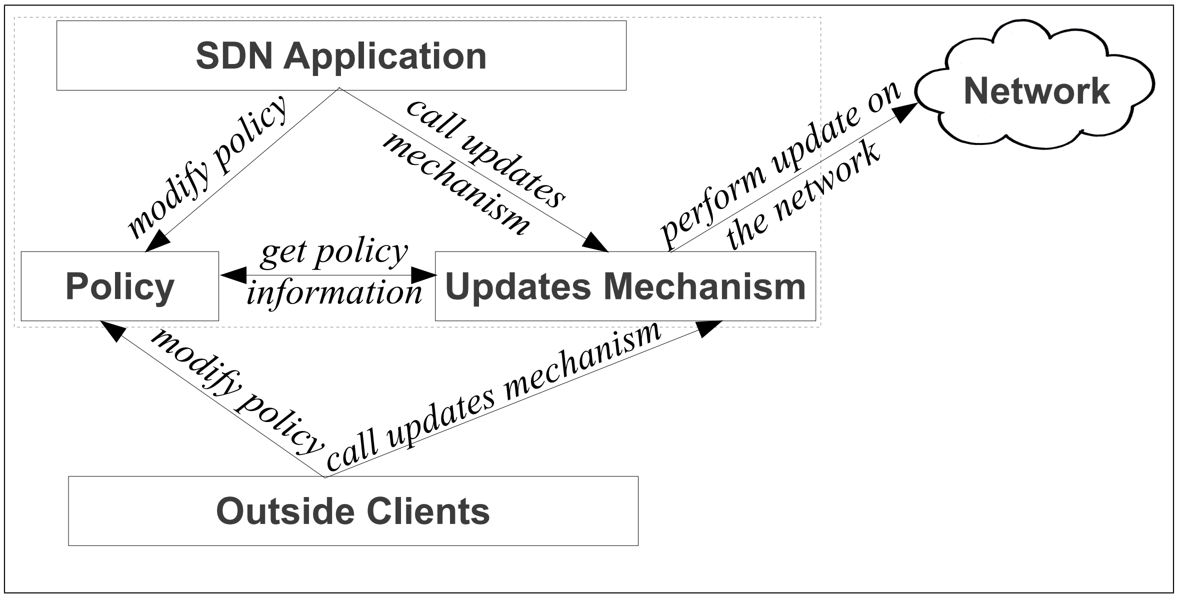

Figure 4.1: Main Components of the Updates System

Function	Action
void sequentialUpdate()	Performs classical consistent update algorithm
void basicParallelUpdate()	Performs the first parallel algorithm
void alternativeParallelUpdate()	Performs the path-based parallel algorithm

Table 4.1: Functionality Provided by the Updates Mechanism

Function	Action
void addRule(switchId, ruleName, rule)	Adds a rule with name ruleName on a given switch
void addPath(pathName, path)	Adds a path with a given name
void removeRule(switchId, ruleName)	Removes a rule from the policy
void removePath(pathName)	Removes a path by its name
List<Rule> getRules(switchId)	Retrieves all rules of a given switch
void clearPolicy()	Removes all rules and paths from the policy

Table 4.2: Functionality Provided by Policy

4.3 Provided Functionality

In order to summarize the details from the previous sections, the built system supports the following functionality:

- Manipulation of the current network policy in terms of whole paths as well as low-level OpenFlow rules.
- Ability to perform concurrent modification of the current network policy by clients out of the controller or applications inside the controller.
- Implementation of the sequential and parallel consistent updates algorithms described in Chapter 3.
- Ability to execute network updates on a real or a simulated network.

Chapter 5

Applications

In order to apply and evaluate the algorithms proposed in Section 3, we developed two software-defined networking applications - a network-wide host location learning and shortest-path routing applications. Both of them work inside the controller and extensively use the system functionality described in Section 4.

5.1 Network-wide Host Learning

This application has goals similar to that of the traditional learning switch application. The main difference with the traditional learning switch application is that, instead of learning a switch port for each host, it computes whole paths between hosts and install them on the switches in a per-packet consistent way. It basically works in the following way. Suppose that we have a network with a controller, a number of switches and a number of hosts connected to them. Suppose that host H_1 pings host H_2. Since there are no rules that bring connectivity between these two hosts on the switches and they have not communicated so far, H_1 sends an ARP packet. When this packet arrives at the switch to which H_1 is connected, it is automatically sent to the controller because there is no rule that it can match on. When the packet arrives at the controller, the controller processes it by sending a *packet_out* message to the switch which tells the switch to flood the packet. Eventually, the ARP packet reaches H_2 and an ARP reply is returned to H_1. Then H_1 sends an IP packet, which again goes to the controller because there is no rule for it at the switch connected to H_1. Now this packet contains much more information than the previously sent ARP packet. It has information about the IP and MAC addresses of the destination host H_2. The controller extracts this information and based on it calculates a path between H_1 and H_2. After that, it gives this path to the updates mechanism which installs it per-packet consistently in the network. After the installation, all new packets going from H_1 to H_2 and vice versa will find rules on the switches on which they can match. So the connectivity between H_1 and H_2 is provided. Figure 5.1 shows an elaborate pseudocode of controller's *packet_in* handler for the described application.

```
/* receive packet pi in the controller from switch sw */
Match m = pi.getMatchFromPacket();
if (m.getDataLayerType() != ARP &&
      m.getDataLayerType() != IP) {
   return;
}
if (m.getDataLayerType() == ARP ||
      isPathComputed(m)) {
   writePacketOut(sw, pi, FLOOD);
}
sourceMAC = m.getSourceMAC();
destMAC = m.getDestMAC();
Path path = computePath(sourceMAC, destMAC);
policy.addPath(path);
perPacketUpdate();
```

Figure 5.1: *packet_in* Handler of Host-location Learning Application

If we look at the pseudocode on Figure 5.1, we can notice that it would be possible to further parallelize this application. In this code, the computation of path between H_1 and H_2 is entirely sequential. It can be done by a traditional breadth-first search in the network graph. Figure 5.2 provides a pseudocode of this algorithm.

```
 1: Graph g = /* read information about the graph */
 2: Queue q;
 3: q.add(root);
 4: for (Node n : q) {
 5:   int level = n.getLevel() + 1;
 6:   for (Node m : g.getNeighbors(n)) {
 7:     if (m.getLevel() == INF) {
 8:       m.setLevel(level);
 9:       q.add(m);
10: } } }
```

Figure 5.2: Sequential Breadth-First Search Algorithm

The code shown on Figure 5.2 contains limited opportunities for effective parallelization. A new thread (or process) can be started between lines 4 and 5. This thread will perform all the work on lines 5-9. However, in order for a subsequent thread for a new node to be started, the previous one should have already ended. This limits the number of nodes that can be processed in parallel.

Hassaan *et al.* [15] describe a parallel breadth-first search algorithm that provides more parallelism than the algorithm shown on Figure 5.2. The main observation is that the level of each node found by the breadth-first search algorithm is actually the minimum across the levels of all its neighbors plus one. Using this fact, an algorithm, based on fixpoint computation, can be derived. We initialize the node levels in the following way:

level(root) = 0; level(i) = INF, for all nodes i different from root

Then, we can proceed with the following computation until reaching a fixpoint:

level(n) = min(level(m) + 1), for all nodes m that are neighbors of n

This allows us to design an algorithm that uses an unordered concurrent set in which nodes can be added and taken in any order. This fact substantially increases the number of nodes that can be processed in parallel. Complete pseudocode of the resulting algorithm is shown on Figure 5.3.

```
 1: Graph g = /* read information about the graph*/
 2: WorkSet s;
 3: s.add(root);
 4: for (Node n : s) {
 5:   fork {
 6:     for (Node m : g.getNeighbors(n)) {
 7:       int level = n.getLevel() + 1;
 8:       if (level < m.getLevel()) {
 9:         m.setLevel(level);
10:         s.add(m);
11: } } }
```

Figure 5.3: Parallel Breadth-First Search Algorithm

Synchronization should be added when updating the level of each neighbor. This happens on line 9. It can be easily done by using a lock on the node m while performing the operation on line 9. However, only one memory location is updated, so a lock-free synchronization primitive such as Compare-And-Swap (CAS) could be used.

5.2 Shortest-path Routing

The second application that we develop on top of our system performs shortest-path routing computations. In principle, it works in a similar way as the network-wide host location learning application. The main difference is that the second one does not compute *any* path between H_1 and H_2 but the *shortest* one with respect to the weights of the links. Figure 5.4 shows an elaborate pseudocode of controller's *packet_in* handler for this application.

The shortest path between two hosts can be computed using the classical Dijkstra's algorithm [8]. Pseudocode of this algorithm is shown on Figure 5.5.

The code shown on Figure 5.5 contains limited opportunities for parallelization. A new thread (or process) can be started on line 7. However, in order to start a

```
/* receive packet pi in the controller from switch sw */
Match m = pi.getMatchFromPacket();
if (m.getDataLayerType() != ARP &&
     m.getDataLayerType() != IP) {
  return;
}
if (m.getDataLayerType() == ARP ||
     isPathComputed(m)) {
  writePacketOut(sw, pi, FLOOD);
}
sourceMAC = m.getSourceMAC();
destMAC = m.getDestMAC();
Path path = computeShortestPath(sourceMAC, destMAC);
policy.addPath(path);
perPacketUpdate();
```

Figure 5.4: *packet_in* Handler of Shortest-path Routing Application

```
Graph g = /* read information about the graph */
PriorityQueue q; // ordered by distance
for (Node m : g.getNeighbors(root)) {
  q.push()
}
for (Pair <node, dist> : q) {
  if (node.getDist() == INF) {
    node.setDist(dist);
    for (Node m : g.getNeighbors(node)) {
      if (m.getDist == INF) {
        q.push(<m, dist + g.edgeWeight(node, m)>);
} } }
```

Figure 5.5: Sequential Single-Source Shortest Path Algorithm

thread for a new node, the previous thread should have already ended its execution. This facts drastically limits the number of nodes that can be processed in parallel.

Fortunately, we can follow the idea with the effective parallelization of the breadth-first search algorithm. Here, a fixpoint computation algorithm is also possible. It can be observed that the minimum distance to a node is a the minimum across the sums of minimum distances to its neighbors plus the weight of the link between the node and its neighbor. The minimum distances are initialized in the following way:

dist(root) = 0; d(k) = INF, for all nodes k different from root

And then the following fixpoint computation can be used:

dist(n) = min(dist(m) + weight(m, n), for all nodes m that are neighbors of n)

The resulting parallel shortest-path algorithm is shown of Figure 5.6. It is basically a parallel version of the Ford-Bellman single source shortest-path graph algorithm [9, 6].

```
Graph g = /* read information about the graph*/
WorkSet s;
s.add(root);
for (Node n : s) {
  fork {
    for (Node m : g.getNeighbors(n)) {
      int dist = n.getDist() + g.edgeWeight(n, m);
      if (dist < m.getDist()) {
        m.setDist(dist);
        s.add(m);
} } }
```

Figure 5.6: Parallel Single-Source Shortest Path Algorithm

Additional synchronization should be added when updating the minimum distance of a node in analogous way to updating the node level in the parallel breadth-first search algorithm. It again can be done either by using a lock or a lock-free synchronization primitive such as Compare-And-Swap (CAS).

Chapter 6

Experimental Results

6.1 Experimental Setting

In order to evaluate the effectiveness of the proposed parallel algorithms, we performed an experiment with the applications described in Chapter 5 on a modern multicore server machine and a network emulated with Mininet [19].

We ran the experiment on a Fujitsu RX200 S6 server with a dual, six-core 3.06GHz Intel X5675 processor with 48GB RAM running the PUIAS distribution of Linux. The controller application (*i.e.* the network-wide host learning or the shortest-path routing applications) was running on the mentioned machine. A Mininet instance was running on a separate Intel-based 2.7GHz four-core machine with 6GB RAM. The controller and the Mininet instance communicated through the Princeton COS Department Network.

There are several different factors that can change slightly the results of our computations. Such factors include thread scheduling fluctuations in the server operating system, network delay and disruptions, other applications concurrently running on the server or other applications concurrently running on the host where Mininet runs. In order to minimize the effects of all these factors, each run of our programs was repeated four times and the average execution time of the four runs is presented in the results here.

6.2 Results

We measure the scalability of the proposed parallel algorithms on a big number of processor cores. We emulated a full binary tree topology with four levels inside Mininet. Two of the nodes made a ping with all other nodes in the network, *i.e.* the number of pings was equal to two times the number of hosts. The current implementation of our system computes and installs unidirectional paths. That is, if we have a ping from A to B, a separate paths from A to B and from B to A will be computed and installed. Moreover, it will happen for different protocols. For example, separate paths for IP and ARP are installed between the same hosts. So the number of installed paths is several times bigger than the number of pings.

There were several main reasons for choosing the tree topology for this experiment. First, it has relatively big amount of switches compared to the number of host. For example, in a fat tree topology, the number of hosts is quite bigger than the number of switches. Also, in a tree topology most of the paths have big number of switches in common. This will allow us to see how the algorithms perform under higher contention on the lock protecting each switch. The result of this experiment is shown on Figure 6.1.

As it can be seen from the diagram, both algorithms do not scale very well beyond four threads. We explain this observation with the following facts. First, in our current implementation, there is excessive amount of locking introduced by the applications, in order to synchronize thread-unsafe Floodlight libraries. Before installing a rule on a switch, each thread waits for a lock on the switch. The implementation of the Policy class also uses a fair amount of locking especially in the functions used

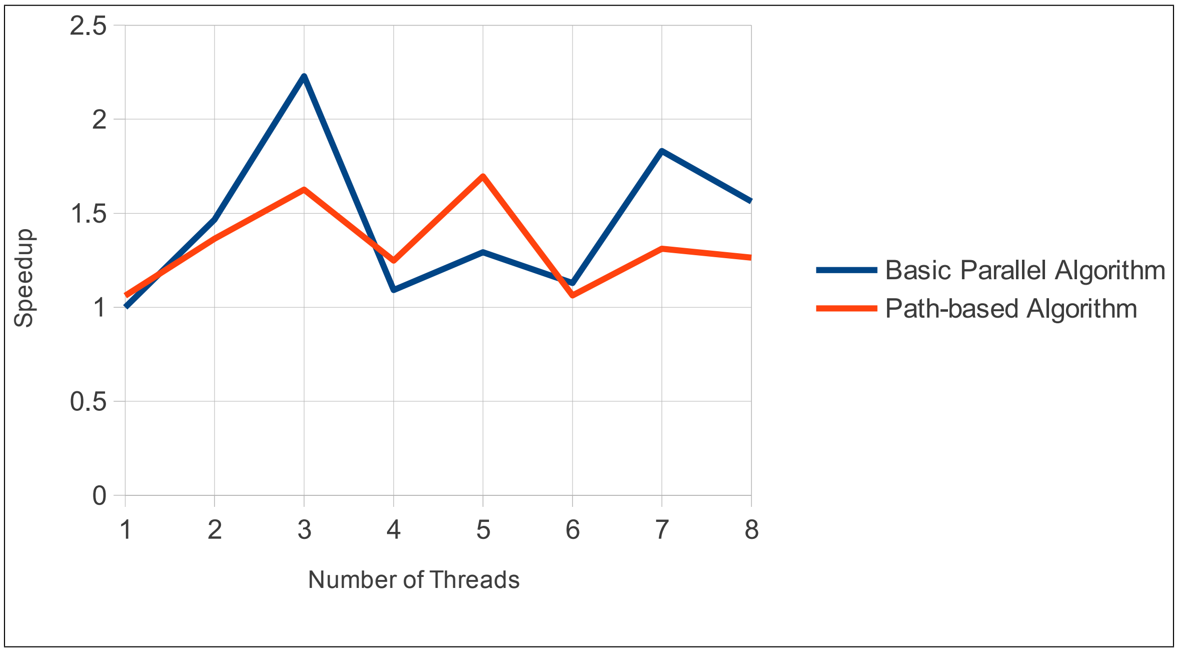

Figure 6.1: Scalability of the Proposed Parallel Algorithms

by the path-based parallel algorithm. Another reason, for the low scalability of the path-based algorithm beyond four threads is that most of the paths have more than one switch in common. This leads to higher contention on the locks that protect the access to the switches. The last potential reason could be the use of *newFixedThread-Pool* in the current implementation. This thread pool, in contrast to other ones, does not apply any intelligent scheduling to the threads. It just assigns a new thread for each task immediately after it is submitted to the pool. It is the most convenient one for this experiment because it allows us to specify the exact number of threads that it will use.

Although none of the algorithms scales in an excellent way beyond four threads, both of them achieve very good performance with up to three threads. This is a good improvement over the sequential per-packet consistent update algorithm. Moreover, the single-threaded path-based algorithm has a slightly better performance than the classical sequential one.

The second part of the experiment aims to investigate the scalability limits of the described in Chapter 5 parallel breadth-first search algorithm on the tree topology. We measured the total time of all executions of the parallel breadth-first search algorithm in the network-wide host learning application. We did this with different number of threads and compared the results based on their speedup over the single-threaded version. Figure 6.2 shows the results.

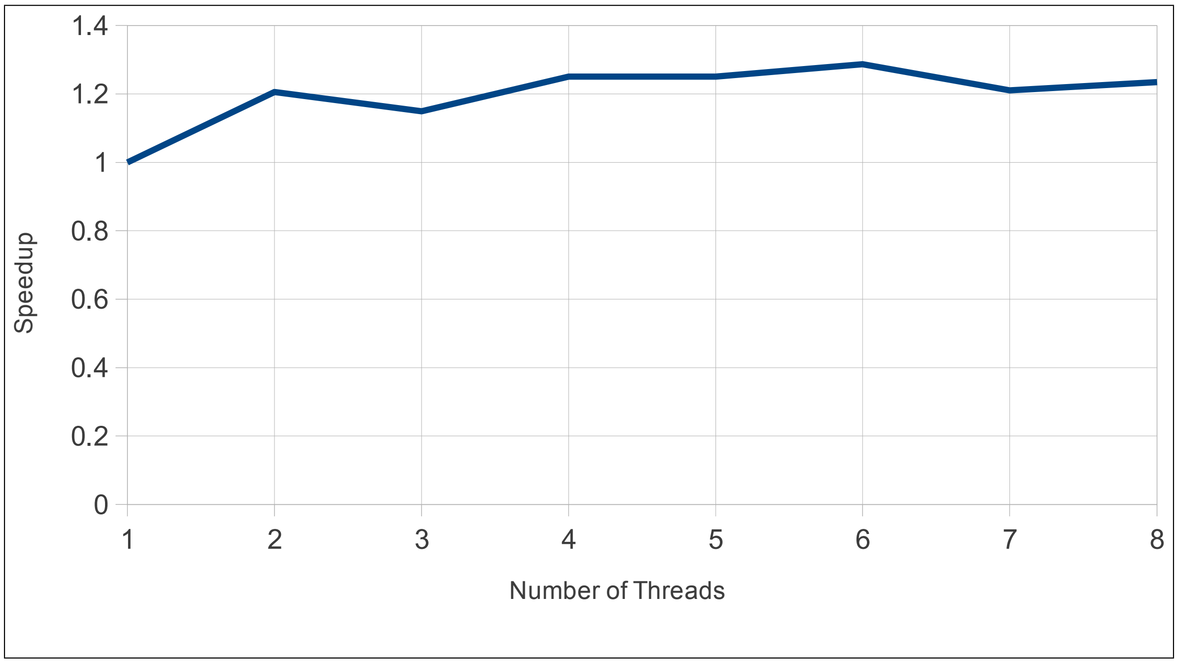

Figure 6.2: Scalability of Parallel Breadth-First Search

The parallel breadth-first search algorithm and the parallel shortest-path one, can exploit more parallelism when there is a bigger number of potential paths. However, in the tree topology there is only one path between each two nodes and each node has a limited number of neighbors. These facts limits the amount of parallelism that the two algorithms could exploit.

Chapter 7

Future Work

There is a number of possible directions for future improvements of the proposed techniques and algorithms. The first and most important one is to investigate more efficient ways for synchornization of the described parallel algorithms. Currently, the scalability of our implementation is limited by a substantial amount of locking. Coming up with an implementation that extensively uses fast lock-free synchronization mechanisms, instead of locks, would definitely increase performance.

In section 3.4, we described two possible optimizations of the classical per-packet consistent updates algorithm. It would be interesting to see how well the parallel algorithms described in this work do, when performing the mentioned optimizations. Likely, the use of *pure extension* optimization would not lead to substantial improvement of the path-based parallel algorithm's performance, because it will limit the parallelism that the algorithm would be able to exploit.

Currently, the proposed abstraction is able to express routing applications. It can be easily extended to applications that modify packets or routes. These new actions in the applications open many new opportunities for parallelization but also raise new problems. Such problems include correctness, synchronization and performance ones. We leave tackling these problems for future work.

Chapter 8

Conclusion

In this work, we proposed two different new parallelizations of the per-packet consistent network update mechanism that was initially invented by Reitblatt *et al.* [22]. We implemented them on top of Floodlight [3] - a modern Java-based multi-threaded software-defined networking controller platform. The implementation was used in two real applications - a network-wide host location learning and shortest-path routing applications. The evaluation was done on an emulated network with a controller running on a modern multicore machine. At the end, we mentioned the most notable directions for future improvement of the proposed techniques.

With the increasing interest on software-defined networking, new ways for optimizing the performance of SDN controllers should be explored. Exploiting parallelism in controller applications is a promising option for achieving this goal. However, coming up with and effective parallelization of network algorithms can be a challenging task. Achieving good performance of a parallel network application involves the addition of efficient non-trivial synchronization and a sophisticated implementation. This work showed how this can be done in the context of network updates algorithms that also bring additional consistency guarantees.

Bibliography

[1] Beacon OpenFlow Controller. See http://openflow.stanford.edu/display/Beacon/Home.

[2] Firewall Migration in the Enterprise. See http://etherealmind.com/sdn-use-case-firewall-migration-in-the-enterprise/.

[3] Floodlight OpenFlow Controller. See http://www.projectfloodlight.org/floodlight/.

[4] OpenFlow Protocol Specification. See http://www.openflow.org/.

[5] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 452–476, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[7] Martín Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *IEEE/ACM Trans. Netw.*, 17(4):1270–1283, August 2009.

[8] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[9] L. R. Ford. Network flow theory. Technical Report P-923, RAND Corporation, Santa Monica, CA, 1956.

[10] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[11] P. Francois, O. Bonaventure, B. Decraene, and P. A. Coste. Avoiding Disruptions During Maintenance Operations on BGP Sessions. *IEEE Trans. on Netw. and Serv. Manag.*, 4(3):1–11, December 2007.

[12] P. Francois, M. Shand, and O. Bonaventure. Disruption Free Topology Reconfiguration in OSPF Networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 89–97, 2007.

[13] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, October 2005.

[14] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[15] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 3–12, New York, NY, USA, 2011. ACM.

[16] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus routing: the internet as a distributed system. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 351–364, Berkeley, CA, USA, 2008. USENIX Association.

[17] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[18] Nate Kushman, Dina Katabi, and John Wroclawski. A Consistency Management Layer for Inter-Domain Routing. Technical report, MIT CSAIL, 2006.

[19] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[20] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[21] Saqib Raza, Yuanbo Zhu, and Chen-Nee Chuah. Graceful network state migrations. *IEEE/ACM Trans. Netw.*, 19(4):1097–1110, August 2011.

[22] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM.

[23] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. In *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[24] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. Seamless network-wide IGP migrations. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 314–325, New York, NY, USA, 2011. ACM.

[25] Andreas Voellmy and Paul Hudak. Nettle: taking the sting out of programming network routers. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 235–249, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] Charles C. Zhang, Marianne Winslett, and Carl A. Gunter. On the Safety and Efficiency of Firewall Policy Deployment. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 33–50, Washington, DC, USA, 2007. IEEE Computer Society.