Building upon RouteFlow: a SDN development experience

Allan Vidal[1 2], **Fábio Verdi**[2], **Eder Leão Fernandes**[1],
Christian Esteve Rothenberg[1], **Marcos Rogério Salvador**[1],

[1] Fundação CPqD – Centro de Pesquisa e Desenvolvimento em Telecomunicações
Campinas – SP – Brazil

[2]Universidade Federal de São Carlos (UFSCar)
Sorocaba – SP – Brazil

{allanv,ederlf,esteve,marcosrs}@cpqd.com.br, verdi@ufscar.br

Abstract. *RouteFlow is a platform for providing virtual IP routing services in OpenFlow networks. During the first year of development, we came across some use cases that might be interesting pursuing in addition to a number of lessons learned worth sharing. In this paper, we will discuss identified requirements and architectural and implementation changes made to shape RouteFlow into a more robust solution for Software Defined networking (SDN). This paper addresses topics of interest to the SDN community, such as development issues involving layered applications on top of network controllers, ease of configuration, and network visualization. In addition, we will present the first publicly known use case with multiple, heterogeneous OpenFlow controllers to implement a centralized routing control function, demonstrating how IP routing as a service can be provided for different network domains under a single central control. Finally, performance comparisons and a real testbed were used as means of validating the implementation.*

1. Introduction

Software Defined Networking (SDN) builds upon the concept of the separation of the data plane, responsible for forwarding packets, and the control plane, responsible for determining the forwarding behavior of the data plane. The OpenFlow protocol [McKeown et al. 2008], an enabling trigger of SDN, introduced the notion of programmable switches managed by a network controller / operating system: a piece of software that controls the behavior of the switches, forming a general view of the network and acting accordingly to application purposes.

The RouteFlow project [RouteFlow] aims to provide virtualized IP routing services on OpenFlow-enabled hardware following the SDN paradigm. Basically, RouteFlow links together an OpenFlow infrastructure to a virtual network environment running Linux-based IP routing engines (e.g. Quagga) to effectively run target IP routed networks on the physical infrastructure. As orchestrated by the RouteFlow control function, the switches are instructed via OpenFlow controllers working as proxies that translate protocol messages and events between the physical and the virtual environments.

The project counts with a growing user base worldwide (more than 1,000 downloads and more than 10,000 unique visitors since the project started in April, 2010). External contributions range from bug reporting to actual code submissions via the community-oriented GitHub repository. To cite a few examples, Google has contributed with an

SNMP plug-in and is currently working on MPLS support and new APIs of the Quagga routing engine. Indiana University has added an advanced GUI and run pilots with hardware switches in the US-wide NDDI testbed. UNIRIO has prototyped a single node abstraction with a domain-wide eBGP controller. UNICAMP has done a port to the Ryu OpenFlow 1.2 controller and is experimenting with new data center designs. While some users look at RouteFlow as "Quagga on steroids" to achieve a hardware-accelerated open-source routing solution, others are looking at cost-effective BGP-free edge designs in hybrid IP-SDN networking scenarios where RouteFlow offers a migration path to OpenFlow/SDN [Rothenberg et al. 2012]. These are ongoing examples of the power of innovation resulting from the blend of open interfaces to commercial hardware and open-source community-driven software development.

In this paper, we present re-architecting efforts on the RouteFlow platform to solve problems that were revealed during the first year of the public release including feedback from third party users and lessons learned from demonstrations using commercial OpenFlow switches.[1] The main issues we discuss include configurability, component flexibility, resilience, easy management interfaces, and collection of statistics. A description of our solutions to issues such as mapping a virtual network to a physical one, topology updates and network events will also be presented from the point of view of our routing application. The development experience made us review some original concepts leading to a new design that attempts to solve most of the issues raised in the first version [Nascimento et al. 2011].

One of the consequences of these improvements is that RouteFlow has been extended to support multiple controllers and virtual domains, becoming, as far as we know, the first distributed OpenFlow application that runs simultaneously over different controllers (e.g., NOX, POX, Floodlight, Ryu). Related work on dividing network control among several controllers has been proposed, for reasons of performance, manageability and scalability [Tavakoli et al. 2009, Heller et al. 2012]. We will present a solution that uses multiple heterogeneous controllers to implement a separation of routing domains from a centralized control point, giving the view of a global environment while keeping the individuality of each network and its controller.

Altogether, this paper contributes with insights on SDN application development topics that will certainly interest the vast majority of researchers and practitioners of the OpenFlow/SDN toolkit. We expect to further evolve discussions around traditional IP routing implemented upon SDN, and how it can be implemented as a service, opening new ways of doing hybrid networking between SDN and legacy IP/Eth/MPLS/Optical domains.

In Section 2 we present the core principles of RouteFlow discussing the previous design and implementation as well as the identified issues. In Section 3, we revisit the objectives and describe the project decisions and implementation tasks to refactor and introduce new features in the RouteFlow architecture. Section 4 presents results from the experimental evaluation on the performance of the middleware in isolation and the RouteFlow platform in action in two possible setups, one in a multi-lab hardware testbed and

[1]Open Networking Summit I (Oct/2011) and II (Apr/2012), Super Computing Research Sandbox (Nov/2011), OFELIA/CHANGE Summer School (Nov/2011), Internet2 NDDI (Jan/2012), 7^{th} API on SDN (Jun/2012). See details on: https://sites.google.com/site/routeflow/updates

another controlling multiple virtual network domains. Section 5 discusses related work on layered SDN application development, multiple controller scenarios, and novel routing schemes. Section 6 presents our work ahead on a research agenda towards broadening the feature set of RouteFlow. We conclude in Section 7 with a summary and final remarks.

2. Core Design Principles

RouteFlow was born as a Gedankenexperiment ("thought experiment") on whether the Linux control plane embedded in a 1U Ethernet switch prototype could be run out of the box in a commodity server with OpenFlow being the solely communication channel between the data and the control plane. Firstly baptized as QuagFlow [Nascimento et al. 2010] (Quagga + OpenFlow) the experiment turned out to be viable in terms of convergence and performance when compared to a traditional lab setup [Nascimento et al. 2011]. With increasing interest from the community, the RouteFlow project emerged and went public to serve the goal of connecting open-source routing stacks with OpenFlow infrastructures.

Fundamentally, RouteFlow is based on three main modules: the RouteFlow client (RFClient), the RouteFlow server (RFServer), and the RouteFlow proxy (RFProxy).[2] Figure 1 depicts a simplified view of a typical RouteFlow scenario: routing engines in a virtualized environment generate the forwarding information base according to the configured routing protocols (e.g., OSPF, BGP) and ARP processes. In turn, the routing and ARP tables are collected by the RFClient daemons and then translated into OpenFlow tuples that are sent to the RFServer, which adapts this FIB to the specified routing control logic and finally instructs the RFProxy, a controller application, to configure the switches using OpenFlow commands.

Matching packets on routing protocol and control traffic (e.g., ARP, BGP, RIP, OSPF) are directed by the RFProxy to the corresponding virtual interfaces via a software switch. The behavior of this virtual switch [3] is also controlled by the RFProxy and allows for a direct channel between the physical and virtual environments, eliminating the need to pass through the RFServer and RFClient, reducing the delay in routing protocol messages and allowing for distributed virtual switches and additional programmability.

2.1. Architectural issues

We identified the most pressing issues in the old architecture (see Figure 2(a)) as being:

Too much centralization. Most of the logic and network view was implemented and stored in the RFServer, without the help of third-party database implementations. The centralization of this design raised concerns about the reliability and performance of a network controlled by RouteFlow. It was important to relieve the server from this burden while providing a reliable storage implementation and facilitating the development of new services like GUI or custom routing logic (e.g. aggregation mode).

[2]As a historical note, the first QuagFlow prototype implemented RFServer and RFProxy as a single NOX application. After the separation (in the first RouteFlow versions) RFProxy was named RouteFlow controller. This caused some confusion, since it actually an application on top of an OpenFlow controller, so we renamed it. Its purpose and general design remain the same.

[3]We employ Open vSwitch for this task: http://openvswitch.org/

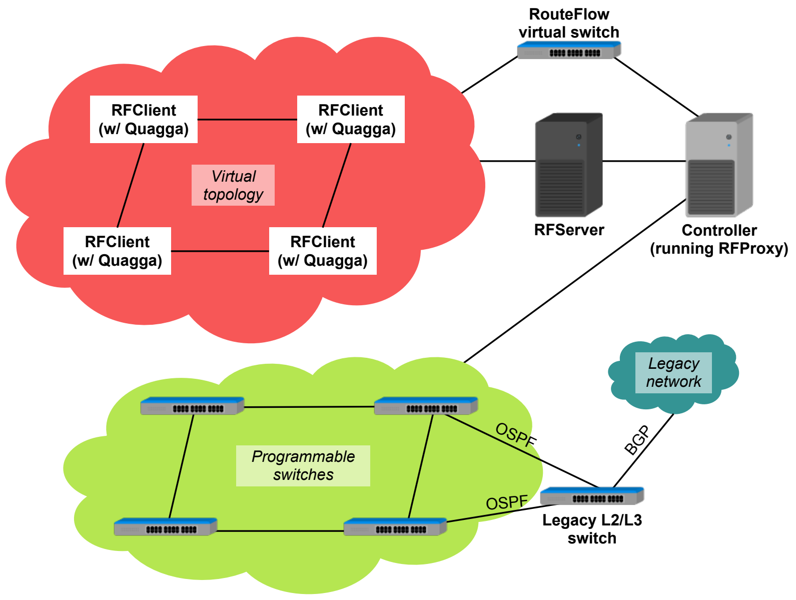


Figure 1. A typical, simplified RouteFlow scenario

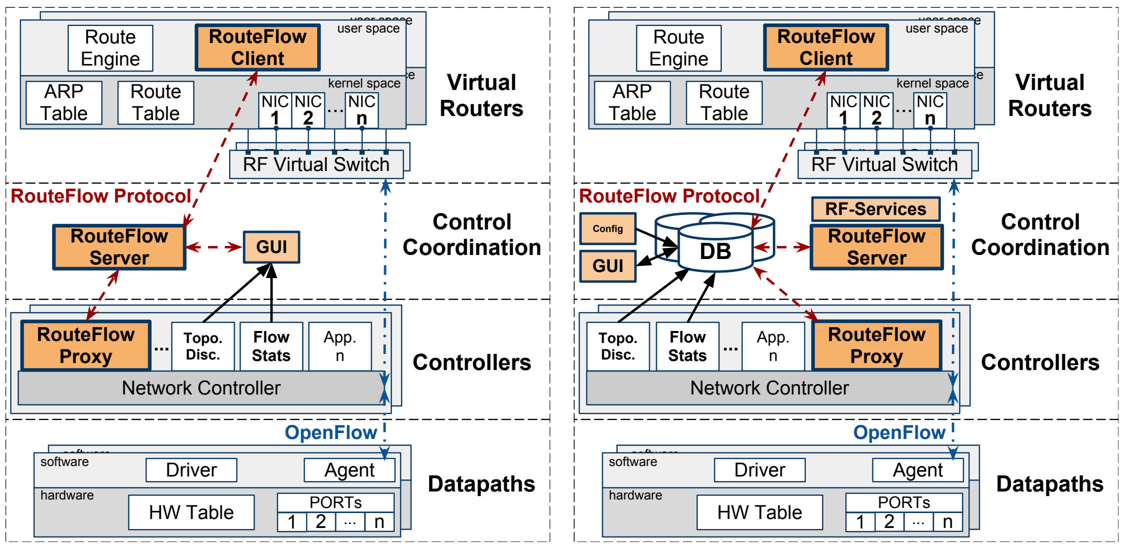
Deficits of inter-module communication. There was no clear and direct communication channel between the RFServer and the RFClients, and also the RFServer and the RFProxy application in the controller. An uniform method of communication was desired, that was extensible, programmer-friendly, and allowed to keep a convenient history of the messages exchanged by the modules to ease debugging and unit testing.

Lack of configurability. The most pressing issue was actually an implementation limitation: there was no way of telling the RouteFlow server to follow a defined configuration when associating the clients in the virtual environment with the switches in the physical environment. This forced the user to start the clients and connect the switches in a certain order without allowing for arbitrary component restart. A proper configuration scheme was needed to instruct the RFServer on how to behave whenever a switch or client joined the network under its control, rather than expect the user to make this match manually.

3. Project Decisions and Implementation

The new architecture, illustrated in Figure 2(b), retains the main modules and characteristics of the previous one. A central database that facilitates all module communication was introduced, as well as a configuration scheme and GUI tied to this database. While tackling the issues in the previous version, we also introduced new features, such as:

Make the platform more modular, extensible, configurable, and flexible. Anticipating the need for updating (or even replacing) RouteFlow components of the RouteFlow architecture, we have followed well-known principles from systems design that allow architectural evolvability [Ghodsi et al. 2011]: layers of indirection, system modularity, and interface extensibility. Meeting these goals involved also exposing configuration to the users in a clear way, reducing the amount of code, building modules with clearer purposes, facilitating the port of RFProxy to other controllers and enabling different services to be implemented on top of RFServer. The result is a better layered, distributed system, flexible enough to accommodate different virtualization use cases ($m : n$ map-



(a) First RouteFlow architecture (b) Redesigned RouteFlow architecture

Figure 2. Evolution of the RouteFlow architecture (as implemented)

ping of routing engine virtual interfaces to physical OpenFlow-enabled ports) and ease the development of advanced routing-oriented applications by the users themselves.

Keep network state history and statistics. One of the main advantages of centralizing network view is that it enables the inspection of its behavior and changes. When dealing with complex routing scenarios, this possibility is even more interesting, as it allows the network administrator to study the changes and events in the network, allowing to correlate and replay events or roll-back configurations.

Consider multi-controller scenarios. We have independently arrived at a controller-filtering architecture that is similar to the one proposed by Kandoo (as we will discuss later in Section 5). The hierarchical architecture allows for scenarios in which different networks (or portions of it) are controlled by different OpenFlow controllers. We can implement this new feature with slight changes to the configuration. Furthermore, the higher-level RouteFlow protocol layer abstracts most of the differences between OpenFlow versions 1.0/1.1/1.2/1.3, making it easier to support heterogeneous controllers.

Enable future works on replication of the network state and high availability. Originally, the RFServer was designed to be the module that took all the decisions regarding the network management, and we want to keep this role so that all routing policy and information can be centralized in a coherent control function. However, centralizing the server creates a single point of failure, and it is important that we consider possibilities to make it more reliable. By separating the network state from its responsibilities now, we can enable future solutions for achieving proper decentralization, benefiting from the latest results from the distributed systems and database research communities.

All the proposed changes are directly related to user and developer needs identified during the cycle of the initial release, some in experimental setups, others in real testbeds. In order to implement them, we went through code refactoring and architectural changes to introduce the centralized database and IPC and a new configuration scheme. The code refactoring itself involved many smaller tasks such as code standardization, proper modularization, reduction of the number of external dependencies, easier testing, rewriting of the web-based graphical user interface and other minor changes that do not warrant detailed description in the scope of this paper. Therefore, we will focus on the newly introduced database and flexible configuration scheme.

3.1. Centralized database with embedded IPC

We first considered the issue of providing an unified scheme of inter-process communication (IPC) and evaluated several alternatives. Message queuing solutions like RabbitMQ or ZeroMQ,[4] were discarded for requiring a more complex setup and being too large and powerful for our purposes. Serializing solutions like ProtoBuffers and Thrift,[5] were potential candidates, but would require additional logic to store pending and already consumed messages, since they provide only the message exchange layer. When studying the use of NoSQL databases for persistent storage, we came across the idea to use the database itself as the central point for the IPC and natively keep a history of the RouteFlow workflow allowing for replay or catch-up operations. A publish/subscribe semantic was adopted for this multi-component, event-oriented solution.

After careful consideration of several popular NoSQL options (MongoDB, Redis, CouchDB)[6], we decided to implement the central database and the IPC mechanism upon MongoDB. The factors that lead to this choice were the programming-friendly and extensible JSON orientation plus the proven mechanisms for replication and distribution. Noteworthy, the IPC implementation (e.g., message factory) is completely agnostic to the DB of choice, should we change this decision.[7]

At the core of the RouteFlow state is the mapping between the physical environment being controlled and the virtual environment performing the routing tasks. The reliability of this network state in RFServer was questionable and it was difficult to improve this without delegating this function to another module. An external database fits this goal, allowing for more flexible configuration schemes. Statistics collection performed by the RFProxy could also be stored in this central database, based on which additional services could be implemented for data analysis or visualization.

The choice of delegating the core state responsibilities to an external database allows for better fault-tolerance, either by replicating the database or separating RFServer in several instances controlling it. The possibility of distributing RFServer takes us down another road: when associated with multiple controllers, it effectively allows for routing to be managed from several points, all tied by a unifying distributed database.

To wrap up, the new implementation is in line with the design rationale and best practices of cloud applications, and includes a scalable, fault-tolerant DB that serves as IPC, and centralizes RouteFlow's core state, the network view (logical, physical, and protocol-specific), and any information base used to develop routing applications (e.g., traffic histogram/forecasts, flow monitoring feedback, administrative policies). Hence,

[4]RabbitMQ: http://www.rabbitmq.com/; ZeroMQ: http://www.zeromq.org/.

[5]Thrift: http://thrift.apache.org; ProtoBuffers https://developers.google.com/protocol-buffers/.

[6]MongoDB: http://www.mongodb.org/; Redis: http://redis.io/; CouchDB: http://couchdb.apache.org/.

[7]While some may call Database-as-an-IPC an antipattern (cf. http://en.wikipedia.org/wiki/Database-as-IPC), we debate this belief when considering NoSQL solutions like MongoDB acting as a messaging and transport layer (e.g. http://shtylman.com/post/the-tail-of-mongodb/).

the DB embodies so-called Network Information Base (NIB) [Koponen et al. 2010] and Knowledge Information Base (KIB) [Saucez and et al. 2011].

3.2. Flexible configuration scheme

In the first implementation of RouteFlow, the association between VMs (running RF-Clients) and the OpenFlow switches was automatically managed by the RFServer with the chosen criteria being the order of registration: the nth client to register would be associated with the nth switch to join the network. The main characteristic of this approach is that it does not require any input from the network administrator other than taking care of the order in which switches join the network.

While this approach works for experimental and well-controlled scenarios, it posed problems whenever the switches were not under direct control. To solve this issue, we devised a configuration approach that would also serve as the basis for allowing multiple controllers to manage the network and ease arbitrary mappings beyond 1:1. In the proposed configuration approach, the network administrator is required to inform RouteFlow about the desired mapping. This configuration is loaded and stored in the centralized database. Table 1 details the possible states a mapping entry can assume. Figure 3 illustrates the default RFServer behavior upon network events.

Whenever a switch[8] joins the network, RFProxy informs the RouteFlow server about each of its physical ports. These ports are registered by the server in one of two ways explicited by Table 1: as (i) *idle datapath port* or (ii) *client-datapath association*. The former happens when there is either no configuration for the datapath port being registered or the configured client port to be associated with this datapath port has not been registered yet. The latter happens when the client port that is to be associated with this datapath (based on the configuration) is already registered as idle.

When a RFClient starts, it informs the RouteFlow server about each of its interfaces (ports). These ports are registered by the RFServer in one of two states shown in Table 1: as an idle client port or an client-datapath association. The association behavior is analogous to the one described above for the datapath ports.

After the association, the RFServer asks the RFClient to trigger a message that will go through the virtual switch to which it is connected and reach the RFProxy. When this happens, the RFProxy becomes aware of the connection between the RFClient and its virtual switch, informing the RFServer. The RFServer then decides what to do with this information. Tipically, the RFProxy will be instructed to redirect all traffic coming from the a virtual machines to the physical switch associated with it, and vice-versa. In the event of a switch leaving the network, all the associations involving the ports of the

Table 1. Possible association states

Format	Type
vm_id, vm_port, -, -, -, -, -	idle client port
-, -, -, -, dp_id, dp_port, ct_id	idle datapath port
vm_id, vm_port, dp_id, dp_port, -, -, ct_id	client-datapath association
vm_id, vm_port, dp_id, dp_port, vs_id, vs_port, ct_id	active client-datapath association

[8]Terms datapath and switch are used interchangeably.

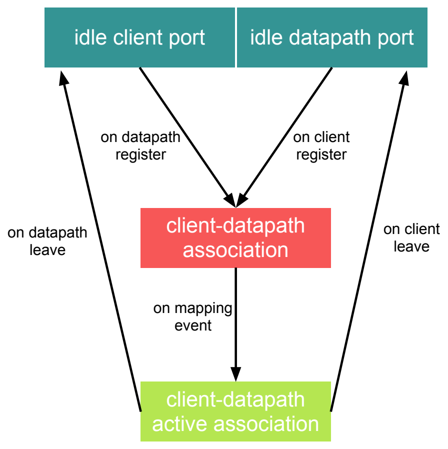


Figure 3. RFServer default association behavior

switch are removed, leaving idle client ports in case there was an association. In case the datapath comes back, RouteFlow will behave as if it were a new datapath, as described above, restoring the association configured by the operator.

An entry in the configuration file contains a subset of the fields identified in Table 1: `vm_id`, `vm_port`, `dp_id`, `dp_port`, `ct_id`. These fields are enough for the association to be made, since remaining fields related to the virtual switch attachment (`vs_*`) are defined at runtime. The `ct_id` field identifies the controller to which the switch is connected. This mechanism allows RouteFlow to deal with multiple controllers, either managing parts of the same network or different networks altogether.

Considering that the virtual environment can also be distributed, it becomes possible to run several routing domains on top of a single RFServer, facilitating the management of several routed networks under a single point. This segmentation presents a possible solution for provisioning of routing as a service [Lakshminarayanan et al. 2004]. In this sense, our solution is capable of controlling independent networks, being different ASes, subnetworks, ISPs or a combination of them, a pending goal of the original RouteFlow paper [Nascimento et al. 2011] to apply a PaaS model to networking.

4. Evaluation

To validate the new developments, we conducted a number of experiments and collected data to evaluate the new architecture and exemplify some new use cases for RouteFlow. The code and tools used to run these tests are openly available.[9] The benchmarks were made on a Dell Latitude e6520 with an Intel Core i7 2620M processor and 3 GB of RAM.

Simple performance measurements were made using the *cbench* tool [Tootoonchian et al. 2012], which simulates a number of OpenFlow switches generating requests and listening for flow installations. We adapted *cbench* to fake ARP requests (inside 60 bytes `packet-in` OpenFlow messages). These requests are handled by a modified version of the RFClient so that it ignores the routing engine. This way, we are effectively eliminating the influences of both the hardware and software which are not under our control, measuring more closely the specific delay introduced by RouteFlow.

[9]https://github.com/CPqD/RouteFlow/tree/benchmark

4.1. How much latency is introduced between the data and control planes?

In latency mode, *cbench* sends an ARP request and waits for the `flow-mod` message before sending the next request. The results for RouteFlow running in latency mode on POX and NOX are shown in Figure 4.

Each test is composed by several rounds of 1 second in duration, in which fake packets are sent to the controller and then handled by RFProxy that redirects them to the corresponding RFClient. For every test packet, the RFClient is configured to send a flow installation message. By doing this, we are testing a worst-case scenario in which every control packet results in a change in the network. These tests are intended to measure the performance and behavior of the new IPC mechanism.[10]

Figure 4 illustrates the cumulative distribution of latency values in three tests. Figure 4a shows the latency distribution for a network of only 1 switch. In this case, the IPC polling mechanism is not used to its full extent, since just one message will be queued every time. Therefore, the latency for the majority of the rounds is around the polling timeout. Figure 4b shows the accumulated latency, calculated considering all 4 switches as one. When compared to Figure 4c, which shows the average latency for all the 4 switches, the scales differ, but the behavior is similar. The accumulated latency shows that the IPC performs better in relation to the case in Figure 4a, mostly because the IPC will read all messages as they become available; when running with more than one switch, it is more likely that more than one message will be queued at any given time, keeping the IPC busy in a working cycle, not waiting for the next poll timeout.

Another comparison based on Figure 4 reveals that RouteFlow running on top of NOX (RFProxy implemented in C++) is more consistent in its performance, with most cycles lasting less than 60 ms. The results for POX (RFProxy implemented in Python) are less consistent, with more cycles lasting almost twice the worst case for NOX.

4.2. How many control plane events can be handled?

In throughput mode, *cbench* keeps sending as many ARP requests as possible in order to measure how many flow installations are made by the application. The throughput test stresses RouteFlow and the controller, showing how many flows can be installed in a single round lasting for 1 second. The results in Table 2 show how many flows can

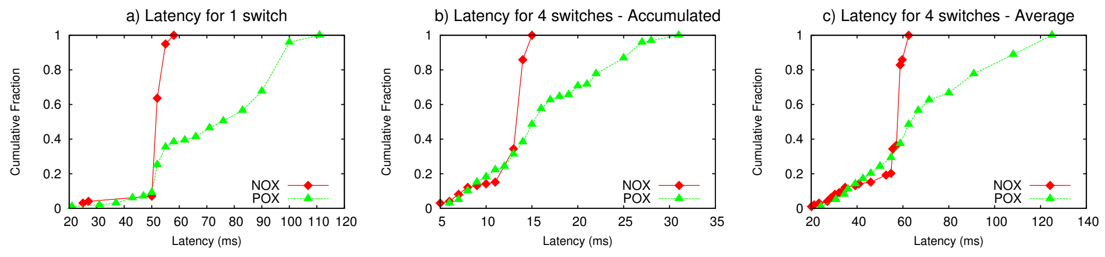


Figure 4. Latency CDF graphs for NOX and POX controlling a single network with 1 and 4 switches (taken from 100 rounds)

[10]The IPC mechanism uses a 50 ms polling time to check for unread messages. This value was chosen because it optimizes the ratio of DB access to message rate when running in latency mode. Whenever a polling timeout occurs, the IPC will read all available messages.

be installed in all of the switches in the network during a test with 100 rounds lasting 1 second each. The results show that the number of switches influence the number of flows installed per second, more than the choice of the controller.

Table 2. Total number of flows installed per second when testing in throughput mode (Average, standard deviation and 90% percentile taken from 100 rounds).

Controller	1 switch		4 switches	
	# Flows$_{avg}$	# Flows$_{90\%}$	# Flows$_{avg}$	# Flows$_{90\%}$
POX	915.05 $\pm$62.11	1013.0	573.87 $\pm$64.96	672.0
NOX	967.68 $\pm$54.85	1040.0	542.26 $\pm$44.96	597.0

4.3. What is the actual performance in a real network?

Test on a real network infrastructure were performed using the control framework of the FIBRE project,[11] with resources in two islands separated by 100 km (the distance between the CPqD lab in Campinas and the LARC lab at USP in São Paulo). To evaluate the delay introduced by the virtualized RouteFlow control plane, we measured the round-trip time from end-hosts when sending ICMP (*ping*) messages to the interfaces of the virtual routers (a LXC container in the RouteFlow host). This way, we are effectively measuring the compound delay introduced by the controller, the RouteFlow virtual switch, and the underlying network, but not the IPC mechanism. The results are illustrated in Table 3 for the case where the RouteFlow instance runs in the CPqD lab with one end-host connected in a LAN, and the second end-host located at USP. The CPqD-USP connectivity goes through the GIGA network and involves about ten L2 devices. The end-to-end delay observed between the hosts connected through this network for *ping*) exchanges exhibited line-rate performance, with a constant RTT around 2 ms. The results in Table 3 also highlight the performance gap between the controllers. The NOX version of RFProxy introduces little delay in the RTT, and is more suited for real applications

Table 3. RTT (milliseconds) from a host to the virtual routers in RouteFlow (average and standard deviation taken from 1000 rounds)

Controller	host@CPqD	host@USP
POX	22.31 $\pm$16.08	24.53 $\pm$16.18
NOX	1.37 $\pm$0.37	3.52 $\pm$0.59

4.4. How to split the control over multiple OpenFlow domains?

In order to validate the new configuration scheme, a simple proof-of-concept test was carried to show the feasibility of more than one network being controlled by RouteFlow. This network setup is illustrated in Figure 5, and makes use of the flexibility of the new configuration system. A central RFServer controls two networks: one contains four OpenFlow switches acting as routers being controlled by a POX instance, and the other contains a single OpenFlow switch acting as a learning switch being controlled by a NOX instance. In this test, RouteFlow was able to properly isolate the routing domains belonging to each network, while still having a centralized view of the networks.

[11] http://www.fibre-ict.eu/

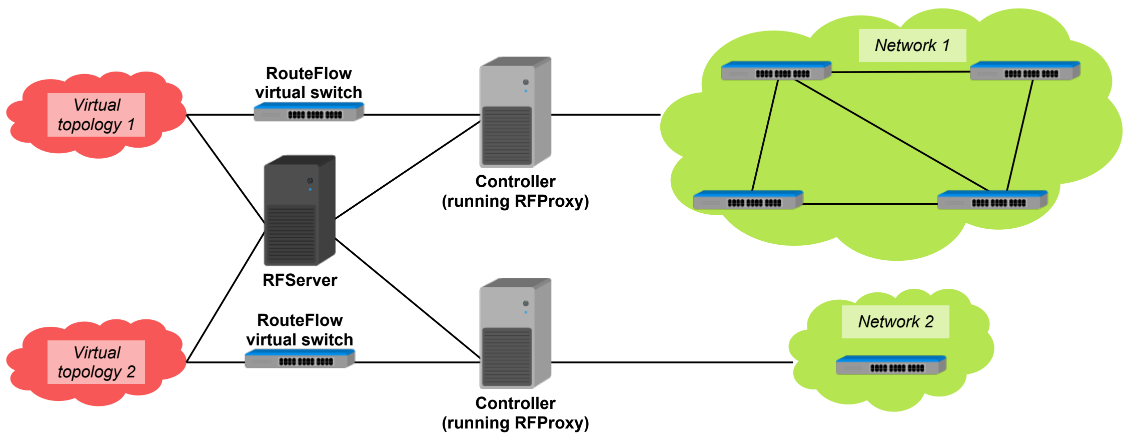


Figure 5. Test environment showing several controllers and networks

5. Related work

Layered controller application architectures. Our architectural work on RouteFlow is very similar to a recent proposal named Kandoo [Hassas Yeganeh and Ganjali 2012]. In Kandoo, one or more local controllers is directly connected to one or more switches. Messages and events that happen often and are better dealt with less latency when treated in these local (first hop) controllers. A root controller (that may be distributed), treats less frequent application-significant events, relieving the control paths at higher layers. Comparing RouteFlow and Kandoo, a notable similarity is adopting a division of roles when treating events. In RouteFlow, the RFProxy is responsible for dealing with frequent events (such as delivering `packet-in` events), only notifying the RFServer about some network-wide events, such as a switch joining or leaving. In this light, RFServer acts as a root controller in Kandoo. A key difference is the inclusion of a virtual environment on top of RFServer. This extra layer contains much of the application logic, and can be easily modified and distributed without meddling with the actual SDN application (RouteFlow). We also differ in message workflow because routing packets are sent from the RFProxy directly to the virtual environment, as determined by RFServer but without going through it. This creates a better performing path, partially offsetting the introduction of another logic layer in the architecture.

Trade-offs and controller placement. Though we do not directly explore performance and related trade-offs, some other works have explored the problem of controller placement [Heller et al. 2012] and realizing a logically centralized control functions [Levin et al. 2012]. Both lines of work may reveal useful insights when applying multiple controllers to different topologies using RouteFlow.

Network slicing. Flowvisor [Sherwood et al. 2010] bears some resemblance in that the roles of several controllers are centralized in a unique point with global view. In this case, the several instances of RFProxy behave as controllers, each with a view of their portion of a network, while RFServer centralizes all subviews and is a central point to implement virtualization policies. However, our work has much more defined scope around IP routing as a service, rather than serve as a general-purpose OpenFlow slicing tool.

Routing-as-a-Service. By enabling several controllers to be managed centrally by RouteFlow, we have shown a simple implementation towards routing as a service [Lakshminarayanan et al. 2004] based on SDN. OpenFlow switches can be used to implement routers inside either Routing Service Provider (RSP) or directly at the ASes (though this would involve a considerably larger effort). These routers could be logically separated in different domains, while being controlled from a central interface. One of the key points of RouteFlow is its integration capabilities with existing routing in legacy networks. The global and richer view of the network facilitated by SDN may also help implement some issues related to routing as a service, such as QoS guarantees, conflict resolution and custom routing demands [Kotronis et al. 2012].

Software-defined router designs. Many efforts are going on into software routing designs that benefit from the advances of general-purpose CPU and the flexibility of open-source software routing stacks. Noteworthy examples include XenFlow [Mattos et al. 2011] that uses a hybrid virtualization system based on Xen and OpenFlow switches following the SDN control plane split but relying on software-based packet forwarding. An hybrid software-hardware router design called Fibium [Sarrar et al. 2012] relies on implementing a routing cache on the hardware flow tables while keeping the full FIB in software.

6. Future and ongoing work

There is a long list of ongoing activities around RouteFlow, including:

High-availability. Test new scenarios involving MongoDB replication, stand-by shadow VMs, and datapath OAM extensions (e.g. BFD triggers). While non-stop-forwarding is an actual feature of OpenFlow split architectures in case of controller disconnection, further work in the routing protocols is required to provide graceful restart. Multi-connection and stand-by controllers introduced in OpenFlow 1.x[12] will be pursued as well. The fast fail-over group tables in v1.1 and above allow to implement fast prefix-independent convergence to alternative next hops.

Routing services. A serious push towards a central routing services provider on top of RouteFlow can be made if we build the capabilities, improvements in the configurability and monitoring in the graphical user interface in order to provide more abstract user interfaces and routing policy languages to free users from low-lvel configuration tasks and experience a true XaaS model with the benefits of outsourcing [Kotronis et al. 2012]. In addition, router multiplexing and aggreagation will be furthered developed. New routing services will be investigated to assist multi-homing scenarios with policy-based path selection injecting higher priority (least R$ cost or lowest delay) routes.

Hybrid software/hardware forwarding. To overcome the flow table limits of current commercial OpenFlow hardware, we will investigate simple virtual aggregation techniques (IETF SimpleVA) and hybrid software/hardware forwarding approaches in spirit of smart flow caching [Sarrar et al. 2012].

Further testing. Using the infrastructure being built by the FIBRE project, we will extend the tests on larger-scale setups to study the impact of longer distances and larger networks. We intend to extend the federation with FIBRE islands from UFPA and UFSCar, and even internationally to include resources from the European partners like i2CAT. Further work

[12]More benefits from moving to the newest versions include IPv6 and MPLS matching plus QoS features. Currently, Google is extending RouteFlow to make use of Quagga LDP label info.

on unit tests and system tests will be pursued including recent advances in SDN testing and debugging tools [Handigol et al. 2012].

7. Conclusions

RouteFlow has been successful in its first objective: to deliver a software-defined IP routing solution for OpenFlow networks. Now that the first milestones have been achieved, our recent work helps to position RouteFlow for the future, enabling the introduction of newer and more powerful features that go much beyond its initial target. The lessons we learned developing RouteFlow suggest SDN practitioners to pay attention to issues such as the (i) amount of centralization and modularization, (ii) the importance of IPC/RPC/MQ, and (iii) flexible configuration capabilities for diverse practical setups.

While OpenFlow controllers often provide means for network view and configuration, their APIs and features often differ, making it important to speak a common language inside an application, making it much easier to extend and port to other controllers by defining so sought northbound APIs. It was also invaluable to have a central messaging mechanism, which provided a reliable and easy-to-debug solution for inter-module communication. As an added value to these developments, our recent work in providing graphical tools, clear log messages, and an easy configuration scheme are vital to allow an SDN application going "into the wild", since these tasks can become quite difficult when involving complex networks and real-world routing scenarios.

As for the future, we are excited to see the first pilots going live in operational trials and further advance on the missing pieces in a community-based approach. Building upon the current architecture and aiming for larger scalability and performance, we will seek to facilitate the materialization of Routing-as-a-Service solutions, coupled with high-availability, better configurability and support for more routing protocols such as IPv6 and MPLS. This will help make RouteFlow a more enticing solution to real networks, fitting the needs of highly virtualized environments such as data centers, and becoming a real alternative to closed-source or software-based edge boxes in use at IXP or ISP domains.

8. References

References

Ghodsi, A., Shenker, S., Koponen, T., Singla, A., Raghavan, B., and Wilcox, J. (2011). Intelligent design enables architectural evolution. In *HotNets '11*.

Handigol, N., Heller, B., Jeyakumar, V., Maziéres, D., and McKeown, N. (2012). Where is the debugger for my software-defined network? In *HotSDN '12*.

Hassas Yeganeh, S. and Ganjali, Y. (2012). Kandoo: a framework for efficient and scalable offloading of control applications. In *HotSDN '12*.

Heller, B., Sherwood, R., and McKeown, N. (2012). The controller placement problem. In *HotSDN '12*.

Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., et al. (2010). Onix: A distributed control platform for large-scale production networks. *OSDI'10*.

Kotronis, V., Dimitropoulos, X., and Ager, B. (2012). Outsourcing the routing control logic: better internet routing based on sdn principles. In *HotNets '12*.

Lakshminarayanan, K., Stoica, I., Shenker, S., and Rexford, J. (2004). Routing as a service. Technical Report UCB/EECS-2006-19.

Levin, D., Wundsam, A., Heller, B., Handigol, N., and Feldmann, A. (2012). Logically centralized?: state distribution trade-offs in software defined networks. In *HotSDN '12*.

Mattos, D., Fernandes, N., Duarte, O., and de Janeiro-RJ-Brasil, R. (2011). Xenflow: Um sistema de processamento de fluxos robusto e eficiente para migraç ao em redes virtuais. In *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC)*.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74.

Nascimento, M., Rothenberg, C., Denicol, R., Salvador, M., and Magalhaes, M. (2011). Routeflow: Roteamento commodity sobre redes programáveis. *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC)*.

Nascimento, M. R., C. Esteve Rothenberg, Salvador, M. R., and Magalhães, M. F. (2010). QuagFlow: partnering Quagga with OpenFlow. *SIGCOMM CCR*, 40:441–442.

Rothenberg, C. E., Nascimento, M. R., Salvador, M. R., Corrêa, C. N. A., Cunha de Lucena, S., and Raszuk, R. (2012). Revisiting routing control platforms with the eyes and muscles of software-defined networking. In *HotSDN '12*.

RouteFlow. Documentation. `http://go.cpqd.com.br/routeflow`. Acessado em 04/10/2012.

Sarrar, N., Uhlig, S., Feldmann, A., Sherwood, R., and Huang, X. (2012). Leveraging Zipf's law for traffic offloading. *SIGCOMM Comput. Commun. Rev.*, 42(1):16–22.

Saucez, D. and et al. (2011). Low-level design specification of the machine learning engine. EU FP7 ECODE Project. Deliverable D2.3.

Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. (2010). Can the production network be the testbed? In *OSDI'10*.

Tavakoli, A., Casado, M., Koponen, T., and Shenker, S. (2009). Applying nox to the datacenter. *Proc. HotNets (October 2009)*.

Tootoonchian, A., Gorbunov, S., Ganjali, Y., Casado, M., and Sherwood, R. (2012). On controller performance in software-defined networks. In *Hot-ICE '12*.