Stemflow: Software-Defined Inter-Datacenter Overlay as a Service

Shuhao Liu, *Student Member, IEEE*, and Baochun Li, *Fellow, IEEE*

***Abstract*— Modern Internet applications are typically hosted in the public cloud, with multiple server instances running within geographically distributed datacenters. Thanks to the abundantly available bandwidth on wide-area links that interconnect these datacenters, it is conceivable that bandwidth-intensive applications may improve their performance by relaying their traffic through such an *inter-datacenter network*. However, there does not yet exist a cloud service that provides a turn-key solution to tap into such available bandwidth resources conveniently. In this paper, we design and implement *Stemflow*, a new system framework that provides *Inter-Datacenter Overlay as a Service* based on the software-defined networking principle. It offers an attractive foundation that helps an Internet application to transparently improve its scalability and performance by using inter-datacenter networks for its traffic. With *Stemflow*, all deployed server instances will construct an overlay atop an inter-datacenter network, and the routing decisions to relay application traffic are made by a centralized controller. The algorithms needed to make these decisions are customized to meet the needs of individual applications, and are cached within the data plane. We motivate and describe the design decisions, and present an extensive experimental evaluation in public cloud infrastructures, using two example applications as our case studies.**

***Index Terms*— Software defined networking, wide area networks, overlay networks, web services.**

I. Introduction

REFERRED to as Infrastructure-as-a-Service, cloud service providers, such as Amazon AWS and Microsoft Azure, have routinely provided convenient access to both computation and network resources in their geographically distributed datacenters at affordable costs. As a result, most modern Internet applications are hosted in virtual machines or lightweight containers in these datacenters. To serve users from different locations with acceptable accessibility, scalability, and availability, it is a common practice to deploy multiple instances of servers in geographically distributed datacenters, especially for network-intensive applications. For example, Twitch, a live video broadcaster, takes advantage of geo-distributed deployment in Amazon datacenters.[1]

Maintained by large cloud service providers, these datacenters are often inter-connected with high-capacity links. Some inter-datacenter links can provide 100s of Gbps to Tbps of capacity, due to the use of dedicated fiber-optic links [13]. For this reason, such *inter-datacenter networks* have recently emerged as an attractive option to serve as the "backbone" of the public Internet over the wide area.

Unfortunately, until recently such high inter-datacenter capacities over the wide area have not been efficiently utilized; the average utilization of even the busier links is 40-60% [13]. For its traffic, Google has deployed a new software-defined infrastructure for more efficient traffic engineering across its inter-datacenter links, substantially improving its link utilization [15].

But how about user-generated traffic from a wide variety of Internet and mobile applications? It is intuitively conceivable that such applications — especially those that are bandwidth-demanding such as video broadcast and multi-party conferencing — would also benefit tremendously from high inter-datacenter capacities in the cloud. Thanks to Software-as-a-Service (SaaS), modern cloud applications have convenient turn-key access to features such as databases and load balancing; but unfortunately, there does not yet exist a cloud service, in the spirit of SaaS, which provides a turn-key solution to conveniently and transparently tap into the abundantly available capacities in inter-datacenter networks.

In this paper, we present our design and implementation of *Stemflow*, a new system framework that provides *Software-Defined Inter-Datacenter Overlay as a Service*. Much in the spirit of SaaS, *Stemflow* offers a simple turn-key solution for a globally deployed application to improve its scalability and performance by using the inter-datacenter network for its traffic, yet with very minimal modifications to its source code. With a focus on simplicity, *Stemflow's* application interface is identical to that of a standard server: it interacts with applications using standard protocols, such as HTTP, HTTPS, or in the context of video streaming, the Real Time Messaging Protocol (RTMP).

Internally, however, *Stemflow* is implemented as a *software-defined* overlay atop an inter-datacenter network. The overlay consists of a number of virtual machines (or Docker containers) located at geographically distributed datacenters, used as traffic relays. With a complete separation of the control and data planes, the routing decisions to be used for relaying application traffic are fully controlled by a centralized controller. For the sake of convenience, all of these design choices and their implementation details are completely hidden from the perspective of application developers.

Although *Stemflow* is simple from the perspective of applications, it is designed to perform well, offering an attractive alternative to applications in need for *performance*.

Manuscript received April 1, 2017; revised September 12, 2017; accepted September 25, 2017. Date of publication October 5, 2017; date of current version December 1, 2017. *(Corresponding author: Baochun Li.)*

The authors are with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: shuhao@ece.toronto.edu; bli@ece.toronto.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2017.2760159

[1]https://twitchstatus.com/.

Applications based on *Stemflow* can enjoy a number of salient features:

First, flexible control over cross-datacenter traffic. Unlike traditional OpenFlow-based software-defined networks [19], *Stemflow* implements a novel approach for data plane-controller interactions. It allows developers to easily customize the routing and scheduling algorithms, and to immediately apply to their inter-datacenter traffic with little overhead. In particular, developers can define their algorithms using a concise set of Javascript APIs, which will later be interpreted by a light-weight interpreter at each relay node within the data plane, minimizing the volume of interactions with the controller. It greatly improves efficiency in the inter-datacenter overlay scenario, where network latency makes a significant difference. In addition, *Stemflow* provides real-time measurements of the overlay network, including network latencies and estimates of available bandwidth. These measurements are accessible via APIs on the controller, which allows additional room for further optimization.

Second, higher throughput in terms of wide-area data transfers. The traffic relays in *Stemflow* are designed to make full utilization of the inter-datacenter available bandwidth. Relay nodes communicate with each other via persistent, parallel TCP connections.

Last but not the least, the simplicity of building applications with *Inter-Datacenter Overlay as a Service*. For example, *Stemflow* is designed to support multicast and conferencing sessions with multiple paths and trees across datacenters. Flows can be replicated at relay nodes in a given datacenter, with a few lines of Javascript. In fact, we have developed two example bandwidth-intensive applications based on *Stemflow* — video broadcast and file sharing — in the form of both mobile applications and web clients.

We have conducted an extensive study to evaluate application performance based on *Stemflow*, using a large-scale inter-datacenter network testbed with high capacities. Our evaluation results have demonstrated the high performance of *Stemflow* as a software-defined overlay. As compared to traditional OpenFlow-based software-defined networking designs, *Stemflow* performs well even if the relay nodes are geographically distributed. Our novel scheme that facilitates interactions between control and data planes makes it feasible to avoid a significant amount of communication overhead with the controller. Consequently, flows can initiate about 90% faster at tail distribution, with the controller offloaded. At the same time, users can still enjoy the benefit of centralized routing intelligence. Furthermore, *Stemflow* is fault-tolerant, and flows can recover from link failures in several milliseconds.

II. Overview

A. Motivations and Design Objectives

There exist Internet applications — most prominently live video broadcast (such as Twitch and Periscope) or multi-party video conferencing (such as Skype) — that are *bandwidth-intensive*. They typically depend on application-specific solutions that are based on centralized servers, and video streams are relayed by one of a group of servers from the video source to the participants of the session. Referred to as server-based solutions, existing solutions heavily depended on the accuracy of selecting the best possible server to assist a live video broadcast or conferencing session. Unfortunately, for sessions involving a large number of participants around the world, it may happen that a single server may not offer satisfactory performance due to the lack of available bandwidth to some of the participants.

TABLE I

A Motivating Example for Using Inter-Datacenter Links to Transfer Bulk Data. Taiwan and Iowa Locate in the Google Cloud Datacenters. The Throughput for Data Transfers Is Measured With `Iperf3`

#	Sender	Receiver	Relay	Throughput (Mbps)
1		Toronto	*	27.0 ± 16.3
2	Taiwan	**Toronto**	**Iowa**	$\mathbf{85.4 \pm 2.7}$
3		Iowa	*	97.4 ± 5.2
4	Iowa	Toronto		86.5 ± 3.8

Stemflow is designed as a framework to assist these bandwidth-intensive applications, offering them better performances by using inter-datacenter networks, yet with minimal modifications to their source code. Intuitively, as compared to traditional server-based solutions, there are two reasons why inter-datacenter networks may offer better performances. First, inter-datacenter links often offer much higher bandwidth capacities, up to hundreds of Mbps. Second, rather than selecting a server for all the participants to connect to, each participant can connect to a datacenter that is considered its best choice regarding bandwidth or latency. From a workload perspective, this is naturally a more scalable solution.

To prove these two claims, we measure the throughput for data transfers from a server located in Taiwan to a downloader in Toronto. The measurement shows that relaying the traffic via a server in Iowa (Tab. I #2) can provide >3x of the throughput as compared to direct download (Tab. I #1). The reason is that Iowa can serve the downloader better (Tab. I #4), while the inter-datacenter network link between Taiwan and Iowa has much more bandwidth available (Tab. I #3).

One may question that relaying traffic through intermediate datacenters can increase the cost of data transfers. As cloud providers usually charge bandwidth usage by the volume of data sent *out* of datacenters, adding one relay datacenter could double the cost. However, it is a trade-off between the performance and the cost. Data transfers are typically charged at an affordable cost (*e.g.*, 1 cent per GB).[2] For many applications, especially those requiring high throughput or low jitters, it is worthwhile to pay more for the performance improvement.

Though both of these reasons are important, achieving better performance requires a level of understanding that is more than skin deep. For bandwidth-intensive applications, it may be necessary to support multiple paths from a source to its destination, and even multiple trees in the multicast case. Such support, though feasible at the transport (with MPTCP) or the

[2]Amazon Web Services pricing. https://aws.amazon.com/ec2/pricing/on-demand/

network layer (IP multicast), may be difficult to provide in practice, since off-the-shelf OS platforms or hardware may not support them. It is, therefore, a necessity to conceive and design *Stemflow* to take advantage of an application-layer overlay, offering the ability for any application to enjoy the benefits of multiple paths and trees in such an overlay. Also, for such an overlay to provide sufficient flexibility at runtime, it would be best to adopt the software-defined networking principle to separate the control plane and the data plane completely. In *Stemflow*, all runtime routing decisions are made in the centralized controller.

The good news is that our software-defined application overlay can be designed to be even more flexible than traditional OpenFlow-based software-defined networks. Without the limitation of switch NICs, the relay nodes are far more configurable. As a result, a more general way for controller-data plane interactions is designed and implemented, with the objectives of better communication efficiency, better flexibility, and easier development.

As an example, consider a live streaming course where the instructor needs to share a file with students around the globe. Students from different regions are typically served by nearby servers. While file downloading can tolerate some delays, the video streaming is sensitive to latency and throughput jitters. With *Stemflow*, operators can deploy optimal scheduling and routing algorithms to maximize user experience (*i.e.,* better video quality with best-effort downloading speed), as compared to the simple bandwidth sharing between streaming and downloading via direct TCP connections.

Better performance and flexibility can only become attractive if application developers find *Stemflow* simple to use as a framework. After all, *Stemflow* is designed to eliminate the need for developers to maintain multiple virtual machines across datacenters themselves. From the perspective of the developers, it would be ideal if *Stemflow* can be viewed as a single server, following the same protocol — such as HTTP — as traditional server-based solutions. We believe that, in practice, developers will be on board only when both simplicity and better performance can be offered.

With the objectives in mind, our design may be more aptly called *Inter-Datacenter Overlay as a Service*, since it leverages a software-defined application overlay, tapping into the excessive bandwidth capacities across geo-distributed datacenters.

B. Architectural Design

In this section, we briefly introduce the architecture of *Stemflow* by examining the deployment and the delivery of a sample user-generated flow.

As a start, *Stemflow* is deployed within VMs that are initiated across geo-distributed centers, as a number of server instances located at each datacenter. The VMs that *Stemflow* uses are typically leased from an IaaS public cloud provider, such as Amazon EC2. The number of VMs used depends on the workload at runtime and can be adjusted dynamically over time. Within each VM, the *Stemflow* runtime executable is simply referred to as a *relay server*, for lack of a better term. As is shown in Fig. 1, each application client, *e.g.,* a mobile device using an LTE network, connects to and sends its application traffic to one of the relay servers, which will relay the received data stream to all of its destinations. Following software-defined principles, the data plane in *Stemflow* consists of all the relay servers that are currently forwarding traffic, and it is completely separated from the control plane, which is the logic to decide how such traffic should be forwarded. Similar to software-defined networks, a central controller is responsible for making optimal fine-grained forwarding decisions, by using data plane statistics as they are collected by the relay servers in the data plane.

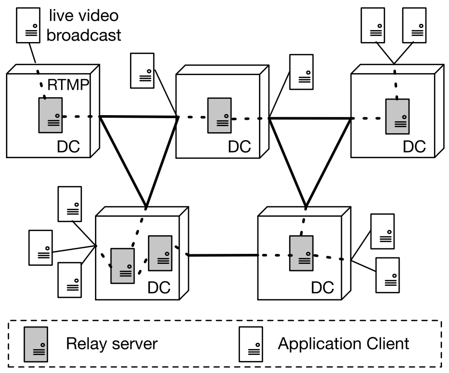


Fig. 1. *Stemflow* is internally designed as a software-defined application overlay across geo-distributed datacenters. One of the applications using the service may be live video broadcast, using RTMP as the protocol to connect to a relay server and to forward traffic using *Stemflow*.

How does a client connect to one of the *Stemflow* relay servers? From the perspective of application developers, and for the sake of simplicity when using *Stemflow* as a cloud service, we opt to support standard protocols — such as HTTP, HTTPS, and RTMP (designed for live video streaming and conferencing) — at each of the relay servers. As a result, applications can directly connect to a relay server without additional work modifying their source code.

For example, live video broadcast is one of the example applications we have built to evaluate *Stemflow*. As shown in Fig. 1, it uses RTMP to connect to one of the relay servers as standard RTMP clients. From the perspective of applications, the service provided by *Stemflow* is semantically and functionally no different from a single RTMP server, used by traditional server-based approaches. This implies that it is possible to publish to or playback from the same URL, without any knowledge of *Stemflow*'s internal design as a scalable inter-datacenter application overlay.

One of the important advantages of using standard protocols is that the type of clients using *Stemflow* as a cloud service does not need to be limited to specific platforms. For example, in addition to mobile applications, it is equally feasible to use a standard QuickTime player, for example, as a client to playback video, as it supports RTMP natively. Alternatively, we have even designed web pages that use Adobe Flash plugins in standard web browsers to send or receive live video via *Stemflow*. Such flexibility is afforded by the *simplicity* of using standard protocols as APIs, one of *Stemflow*'s important design choices.

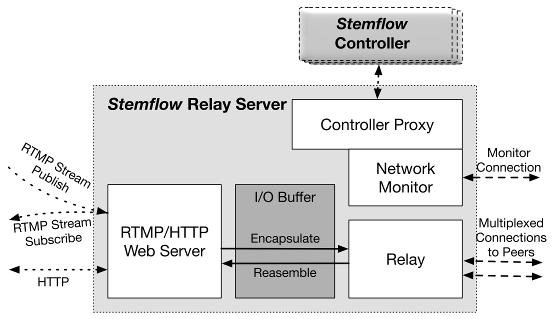


Fig. 2. The architecture of a relay server in *Stemflow*.

To improve performance, we have further optimized the mechanism used for the controller to interact with a large number of relay nodes in the data plane, as the overlay scales up. Each relay node is designed to maintain a local *cache* that stores the forwarding decisions that the controller has made. Rather than the traditional way of storing these decisions as lookup table entries on a network switch, we have chosen to store the control logic directly and explicitly, expressed in a scripting language, such as Javascript used in *Stemflow*. The relay server is then capable of executing such control logic locally at runtime with a Javascript interpreter, with a significantly reduced volume of interactions with the controller.

III. Application Interfaces and the Data plane

The relay servers in *Stemflow* are launched in the VMs across geo-distributed datacenters, and they are responsible for two important tasks: *interacting with application clients* with standard protocols as the interface, as well as *forwarding traffic* in the data plane. In this section, we will present our design choices and implementation highlights in both aspects.

The overall design of each relay server is illustrated in Fig. 2. The application interface for interacting with application clients is designed and implemented as a standard RTMP/HTTP server. It receives traffic from the clients and pushes all received traffic to the *relay* component via internal I/O buffers. The relay component maintains network connections to a subset (or all) of the other relay servers — its neighbors — in the inter-datacenter application-layer overlay. *Stemflow* supports both TCP connections and UDP flows to relay application traffic, but maintains only one connection (or flow) to each of the neighboring relay servers. Traffic from all the applications will be multiplexed and share the same connection to each relay server. At the same time, a network monitor periodically measures network conditions to each neighboring relay server.

A. Interactions With Application Clients

As a cloud service, the relay servers are responsible for interacting with applications clients directly, using standard protocols. In our current implementation, RTMP (for real-time video streaming and conferencing) and HTTP/HTTPS have been supported. Any relay server can parse queries from clients (such as HTTP requests and RTMP stream publish

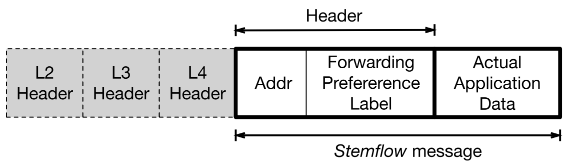


Fig. 3. The structure of a message in *Stemflow*.

messages), as well as receive and send application data, precisely as if it is a simple RTMP or HTTP server.

The RTMP/HTTP server we have so far implemented interacts with the relay component using an internal buffer. In the case of HTTP servers, all the traffic from applications will be written to the inbound buffer of the relay, and via an internal interface, specify the desired destinations to the relay component. In the case of RTMP servers, since RTMP involves exchanges of a number of control messages between the client and the server, we have implemented an adaptor — in about 500 lines of C++ code — to extract the raw video streams and to forward them to the relay component. Note that such a design is sufficiently flexible to easily accommodate future support of other protocols (e.g., RTP/RTSP) or cryptographic mechanisms, if such a need arises.

B. The Data Plane

As its name suggests, the *relay* component in a relay server is responsible for forwarding application traffic, from/to either the RTMP/HTTP server within the same relay server or other neighboring relay servers via the inter-datacenter network.

Fig. 3 shows the structure of a message in *Stemflow*. As soon as application data arrives at the relay component using the inbound buffer shared with the RTMP/HTTP server, a special header is attached to specify its corresponding destination and forwarding policies. Then, the relay component will forward it to its desired destinations, or a set of next-hop relay servers towards the destinations.

1) Performance: Whenever the relay component receives a message from one of its neighboring relay servers, it serves as a message forwarding switch if the message needs to be forwarded to the other relay servers. Naturally, it is up to the controller to make forwarding decisions in the control plan, but the relay component will be responsible for implementing all these forwarding decisions, *i.e.,* forwarding incoming messages to either the internal RTMP/HTTP server for the final delivery, or to the next-hop relay servers as needed.

For the sake of *high-performance*, the entire message forwarding mechanism at the data plane is implemented in C++ in *Stemflow*. As system optimizations, the need for copying data among different buffers is minimized with the use of smart pointers and reference counting. Furthermore, with asynchronous network I/O backed by the `Boost.asio` library, *Stemflow* performs rate control automatically, avoiding overflow on any individual relay server.

Moreover, persistent, parallel TCP connections are established among all relay servers and are multiplexed among data deliveries. This technique significantly improves the throughput achieved for inter-datacenter transfers, maximizing the utilization of the available bandwidth.

2) Monitoring the Inter-Datacenter Network: To provide sufficient statistical information for the controller to make the best possible forwarding decisions, each relay server is able to measure a number of performance metrics across the inter-datacenter network in real time, with respect to both latencies and available bandwidth, and report them to the controller periodically. With respect to latencies, a relay server is able to measure both the Round-Trip Time (RTT) and one-way delays on an inter-datacenter link, with the latter measured by implementing a clock synchronization protocol that conforms to the Network Time Protocol (NTP) [1]. With respect to available bandwidth, we have implemented Initial Gap Increasing (IGI) [14], which uses packet trains to estimate end-to-end available bandwidth.

3) Fault Tolerance: The relay component is also designed to be *fault-tolerant*, in the sense that *Stemflow* can recover from the failures of any online relay server gracefully. In particular, each relay server is connected to its neighbors and consistently monitoring their status. Once a relay server becomes offline, the failure will be discovered immediately and broadcast to all peer relay servers in the network.

IV. Control Plane

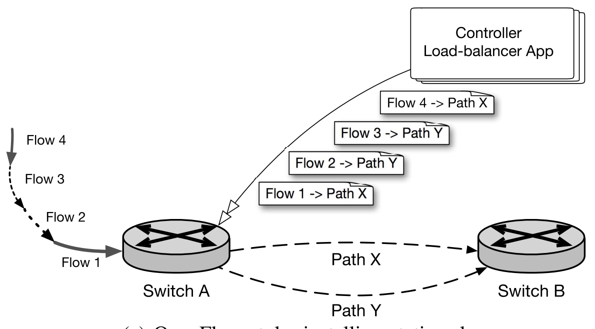
With software-defined networking as its underlying principle, the control and data planes in *Stemflow* are completely separated. The control plane is implemented with a centralized controller in Python, and it communicates — either proactively or reactively — with all the relay servers, collecting real-time measurement statistics and deploying its control decisions.

Traditional ways of scaling the control plane cannot work smoothly in *Stemflow*, whose relay servers are distributed in wide-area networks, being far from each other geographically. *Stemflow* implements a novel controller-data plane interaction scheme. By caching dynamic forwarding logic instead of static flow tables on the relay servers, our scheme greatly improves the scalability of the software-defined networking architecture.

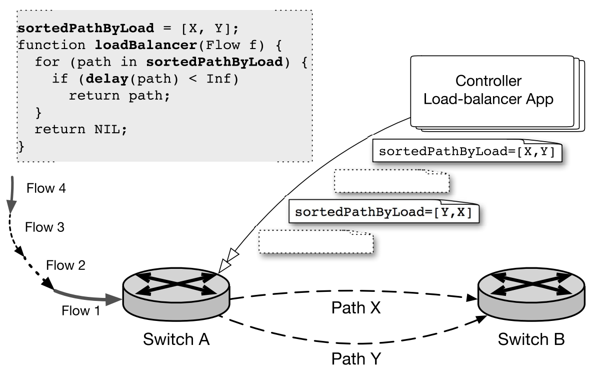
A. Motivation

The OpenFlow-based software-defined networking architecture is problematic while being deployed at large-scale. Since the control plane has to handle all data plane events and ensuring decision integrity at the same time, the single centralized controller approach suffers from high workload and reliability issues. In this case, OpenFlow switches will query the controller about the forwarding decisions for *each* new flow. This approach is quite brute-force but fine-grained because the controller can make flow-specific or even packet-specific decisions in a straight-forward manner.

However, four major issues limit the scalability of this design: i) There will be a significant flow initiation delay, since its head packets will be forwarded to the controller, during which the communication latency adds to the delay; ii) The controller workload is high, and it might be congested by bursty network messages because of the all-to-one traffic pattern; iii) Flow rerouting is necessary but expensive, which may encounter severe consistency issues across the network;



(a) OpenFlow-style: installing static rule.



```
sortedPathByLoad = [X, Y];
function loadBalancer(Flow f) {
  for (path in sortedPathByLoad) {
    if (delay(path) < Inf)
      return path;
  }
  return NIL;
}
```

(b) *Stemflow*: caching dynamic forwarding logic.

Fig. 4. An example to show the benefits of programmable data plane in software-defined networks. There exist two available shortest paths with the same capacity between ingress Switch A and egress Switch B. The network application tries to balance the load between the two paths. The rate of Flow 1 is 3 units, while the rate of Flow 2, 3 and 4 is 2 units, respectively.

iv) Flow table will bloat as the network scales, requiring more expensive TCAM spaces.

Existing approaches to scale an OpenFlow control plane, such as hierarchical or partitioning solutions, either sacrifice the granularity of control or limit the generality of possible network applications (see Sec. VI). Recent progress made on the programmable data plane, as is introduced in OpenFlow 2.0 [2], seems a promising cure for the scalability problem.

In this paper, we implement a novel scalable and readily-deployable scheme for controller-data plane interactions in *Stemflow*. Inspired by the idea of a reconfigurable data plane, the central *Stemflow* controller caches control logic closer to the data plane by *injecting script code to the interpreter runtime on data plane nodes.* Whenever the *Stemflow* data plane node is processing a new message, it directly dispatches the message header information to the provided handler in the script code, to perform the desired computation and actions based on locally cached controller intelligence.

An example of a load-balancing network application is shown in Fig. 4. For a traditional OpenFlow network (Fig. 4a), a rule has to be installed on Switch A *reactively* for each new flow. In the depicted example, four messages are required to achieve globally optimal load-balancing. Note that the response time for rule installation will add to the overhead of flow initiation.

In *Stemflow* (Fig. 4b), on the contrary, the `loadBalancer` logic is cached on Switch A beforehand by the central controller. When a new flow comes in, the path selection is completely handled by the cached `loadBalancer` function, without querying the controller. At the same time, the controller will *proactively* update Switch A with measurement statistics of path load, ensuring the global optimality of forwarding decisions. Only two updating messages are necessary in this case. Moreover, `loadBalancer` takes local measurement statistics into consideration. Notably, a `delay()` function returns the monitoring delays of a given path. Its call on a path with failure will return an `Inf`. Therefore, failed paths will be avoided automatically, without the slow reaction of the controller.

As a conclusion, the benefits of dynamic logic caching are three-fold: i) Time overhead of flow initiation is reduced because the inevitable communication latency is avoided; ii) Both computation and network load on the controller are very likely to be reduced, since statistics are updated on-demand; and iii) All route selection can react to network failures quickly, so as to enable fast recovery.

B. Caching Dynamic Forwarding Logic

Generally speaking, message processing in software-defined networks involves three stages: header matching, computing and action applying. Messages from different flows are identified through header matching. Then, computation is required to get the desired list of actions (*e.g.,* header modification, forwarding to a given next-hop node). Finally, the actions will be applied to the message.

In software-defined networking, the computing stage is completely migrated to the control plane. The core concept of our approach is to allow most of the messages to be processed without consulting the central controller.

To this end, the *Stemflow* controller preemptively caches the control logic (instead of static forwarding rules) on the data plane. As follow-up actions, it frequently compiles and updates the required global statistics. In *Stemflow*, the communication messages between the controller and relay servers are encoded in JavaScript Object Notation (JSON) strings, which naturally enables the logic caching via Javascript code injection. The *Stemflow* relay servers are embedded with a lightweight Javascript interpreter to execute the code.

Specifically, the *Stemflow* controller programs the message processing logic in a serial of Javascript objects. Each of the objects has a key `processMessage()` method, defining the logic to process messages of a given forwarding preference label in the header. Each object is then serialized into a JSON message, thus to be sent to relay servers. The message is then parsed and rebuilt into the origin Javascript objects. These objects that contain the message processing logic, namely the *Message Processing Objects*, are thus registered to the Javascript interpreter on the data plane node. Note that all above tasks can be completed even before the first flow in the network initiates.

The processing of a single message is illustrated in Fig. 5. Whenever there is a new incoming message, the header matcher will unpack the message headers, reading the specified destination information and forwarding preference labels. Such information will be used by the message dispatcher, to select an appropriate Message Processing Object stored to process the corresponding message.

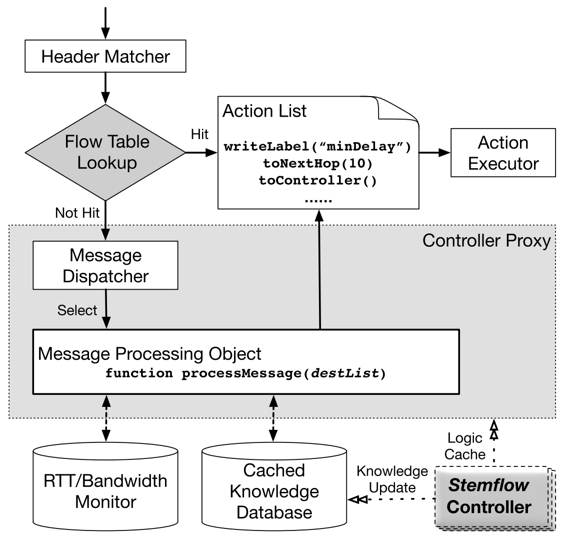


Fig. 5. The message processing diagram in *Stemflow*.

The member function `processMessage()` of the selected Message Processing Object will then be called, with an argument indicating the list of destinations. The return value of this function is a list of pre-defined actions, *e.g.,* rewriting the *Stemflow* message header, relaying to a given next hop node or further consulting the central controller. Finally, the action executor deploys the actions, which concludes a message process cycle.

The above process seems simple; however, it cannot be completed without a particular slice of global network knowledge. At the time a new Message Processing Object being installed on a relay server, it can also *subscribe* the updates of some customized variables (*i.e. global knowledge*) from the controller. This global knowledge can be some intermediate variables used by the algorithm inside `processMessage()`, which cannot be locally obtained.

Moreover, the measurement results of local network delays and bandwidth statistics can be used directly by `processMessage()`. As a result, the forwarding decision making can react directly from the data plane events, such as link failure, without involving the controller.

C. Customizing the Message Processing Logic

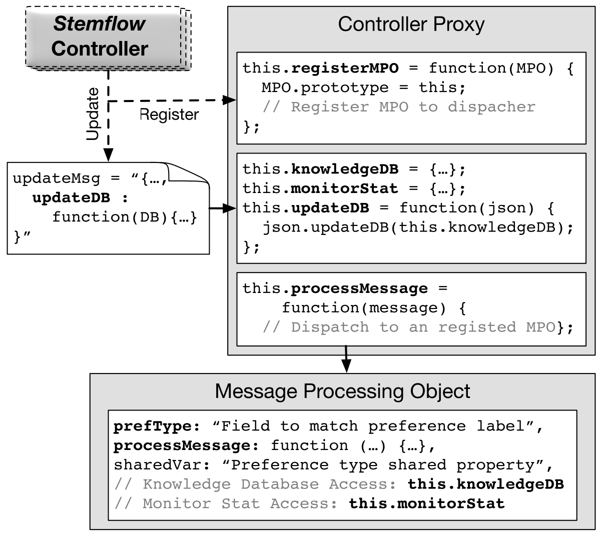
A software-defined networking architecture allows network operators to implement a variety of novel network applications in pure software, customizing the control logic over different network flows. In *Stemflow*, programming a network application is as simple as in OpenFlow. To implement a customized *Stemflow* network application, there are two steps in general: defining a Message Processing Object and subscribing a variable to be cached as global knowledge.

Message Processing Objects are the core objects that define the control logic. Different Message Processing Objects can be used to process messages with different labels. On data plane nodes, new Message Processing Objects are registered to the controller proxy, and inherit the database and monitor access.

Thanks to the prototype inheritance mechanism in Javascript, customized Message Processing Objects can

TABLE II
SUMMARY OF PROTOTYPE NETWORK APPLICATIONS IMPLEMENTED IN *Stemflow*

Network App	Summary	Global Knowledge to Subscribe
"min-delay"	Choose the next hop which directs to the path with the minimum possible one-way delay. Desired bandwidth can be specified as a prerequisite.	Best 3 paths with the least latency of each possible destination. The corresponding available bandwidth is given, too.
"max-bandwidth"	Choose the next hop which directs to the path with most available bandwidth.	Best 3 paths with the most available bandwidth of each possible destination.
"bw-allocation"	Allocate a fraction of the available bandwidth to a single flow, according to the desired relative weight	None.
"load-balancer"	Choose the next hop for a message to balance the load among all available paths. Note that either per-message or per-flow load balancing can be specified.	Same as above.



```
this.registerMPO = function(MPO) {
  MPO.prototype = this;
  // Register MPO to dispacher
};
```

```
updateMsg = "{…,
  updateDB :
    function(DB){…}
}"
```

```
this.knowledgeDB = {…};
this.monitorStat = {…};
this.updateDB = function(json) {
  json.updateDB(this.knowledgeDB);
};
```

```
this.processMessage =
    function(message) {
  // Dispatch to an registed MPO};
```

```
prefType: "Field to match preference label",
processMessage: function (…) {…},
sharedVar: "Preference type shared property",
// Knowledge Database Access: this.knowledgeDB
// Monitor Stat Access: this.monitorStat
```

Fig. 6. Prototype inheritance of Message Processing Objects, illustrating the programming interfaces.

program with simple interfaces shown in Fig. 6. The `prefType` property defines the type of messages to be handle, whose value matches the preference label in message headers. `processMessage()` function programs the message processing logic, as is addressed previously. At the meantime, variables which are shared among the flows of the same `prefType` can be defined or modified as property `sharedVar`. For example, a counter can be added as a shared variable to get the number of processed messages. Access to the global knowledge database and monitor statistics is made via `this.knowledgeDB` and `this.monitorStat` interfaces directly.

The second step is to define the subscribed data from the controller. Together with the Message Processing Object installation, the result of some user-defined functions can be subscribed to update the knowledge database.

We implemented several prototype network applications, in the form of two Message Processing Objects. Tab. II summarizes their implementation. Typically, these three applications are enough to satisfy the general traffic engineering needs to operate *Stemflow* in inter-datacenter networks. Note that multicast is automatically supported by these applications. Also, per-message multipath routing can be enabled by the options specified in the preference label field of a message header.

D. Discussion

One may think caching control logic on the relays is a violation of the software-define network principle. Indeed, in some cases, the relay servers in *Stemflow* seem to work in the same way as a traditional routing. However, it is not true because the control logic is *cached* by the central controller. In other words, the controller is free the delete or update the cached logic at any time. Eventually, the forwarding decisions are still under full provisioning of the central controller.

Caching forwarding logic is compatible with the OpenFlow-style of control, that is, caching static forwarding rules. Directly consulting the controller is the default action to be applied to a *Stemflow* message. In particular, the message will be forwarded to the controller under the following conditions: i) there is no matching Message Processing Object installed; ii) `processMessage()` raises an exception due to a lack of global knowledge.

Another concern about this design is that because the forwarding decisions can be made locally, it might be unfeasible to ensure the correctness of message forwarding. For example, the forwarding decisions might result in a loop. *Stemflow* solves this problem by consistently updating the knowledge cached on relay servers. Since the cached knowledge on different relays is drawn from the same database, a loop can be avoided easily even with greedy path selection. Moreover, the locality of forwarding decision making guarantees the consistency during a network update. There will not be any inconsistent rules installed on the relay servers.

V. EXPERIMENTAL EVALUATION

In this section, we conduct several sets of real-world experiments to evaluate the performance and flexibility of *Stemflow* from a variety of different perspectives. Our experimental results have shown that:

1) The overlay network in *Stemflow* can achieve up to 8.8x throughput as compared to direct TCP connections.
2) Customized routing, flow scheduling, and bandwidth allocation algorithms can be correctly implemented.
3) The data plane-controller interaction scheme that we have implemented can reduce the tail flow initiation delays by almost 10 seconds, and significantly reduces the controller inbound/outbound traffic.

A. Flexibility for Controller Algorithms

1) Multicast Routing and Inter-Datacenter Throughput: First, we test the throughput of a sample multicast session

TABLE III
THE MULTICAST SESSION THROUGHPUT AND WORST MESSAGE DELIVERY LATENCY COMPARISON

Scheme	Throughput (Mbps)	Delivery Latency
min-delay	608.7 ± 9.1	574.3 ± 21.6
max-bandwidth	823.1 ± 11.1	896.3 ± 33.7
direct	92.0 ± 10.3	320.8 ± 18.3

TABLE IV
FLOW WEIGHT SETTINGS DURING THE TEST

Session	Time Range (s)				
	0-10	10-20	20-30	30-40	40-50
1	1	**3**	3	3	**1**
2	1	1	**9**	9	**1**
3	1	1	1	**0.1**	**1**

in Google Cloud. In this experiment, three relay servers are deployed in Taiwan, Iowa, and Belgium, respectively. The server located in Taiwan is the data source, while the other servers work as receivers in the multicast session.

With different network applications deployed, *i.e., "min-delay" and "max-bandwidth,"* we compare the achieved session throughput and the worst message delivery delay with TCP (*i.e.,* "direct"). The results are shown in Table III. "min-delay" choose the direct path between the source and receiver datacenters, while "max-bandwidth" further relays the traffic destined to Belgium via Iowa, which provides $>$200 Mbps of throughput at the cost of $\sim$320 ms of latency in message delivery. As compared to the baseline, *Stemflow* significantly improves the multicast session throughput, because of the high-bandwidth inter-datacenter links and parallel TCP connections.

2) Bandwidth Allocation: Here we verify the effectiveness of the implemented bandwidth allocation algorithm. We launch three flows between Taiwan and Belgium, while they share the overlay link. The weights of these flows have been changed four times during the test, and the relative weights have been listed in Table IV. As a result, the fluctuations of the flow sending rate are illustrated in Fig. 7, which correctly reflect the desired bandwidth allocation rate.

B. The Efficiency of Log Caching

We deploy *Stemflow* in the wide-area network, with relay servers disseminated in datacenters and testbed regions around the globe. Running with two sample mobile applications, we evaluate the performance of *Stemflow* upon real inter-datacenter networks. Particular attention has been paid to the dynamic rule caching mechanism for data plane-controller interactions. The experimental results show that *Stemflow* benefits significantly from this design regarding performance, scalability, and reliability.

1) Experimental Setup: In our experiments, 24 relay servers in total are deployed in the form of virtual machines. The geographical locations of these deployed relay servers are illustrated in Fig. 8, and they are acquired from three different domains.

Specifically, 10 relay servers are deployed in the Amazon EC2 cloud, distributed in 8 different regions (Singapore and

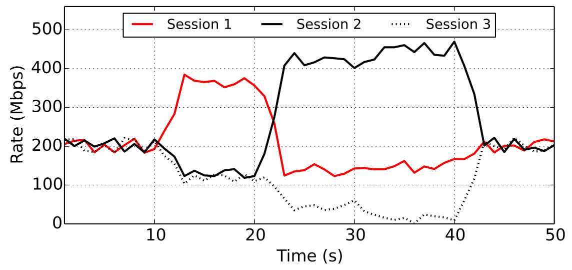


Fig. 7. Bandwidth allocation test results.

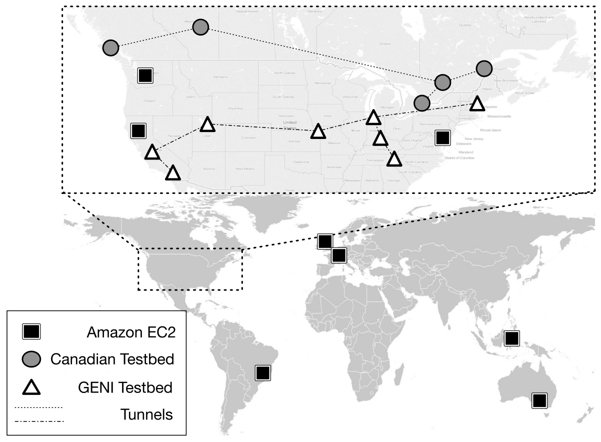


Fig. 8. Global deployment topology.

Sau Paulo hold 2 for each). More relay nodes are launched in North America, with 8 in GENI testbed [7] and 6 in a Canadian testbed. Note that both GENI and the Canadian testbed have tunneled or dedicated links different interconnecting regions, which can be utilized by *Stemflow* as high-performance inter-datacenter links. The controller locates on a dedicated virtual machine near Toronto.

Two types of flows with distinct forwarding preferences are generated to test the performance of *Stemflow* inter-datacenter overlay. One kind of flows is delay-sensitive streams to simulate workload from mobile applications that require data transfer. Live video broadcast, for example, requires traffic to be delivered at the lowest possible latency but sends out traffic at a limited rate that can be pre-specified. In our experiment, the desired streaming rate is randomly selected between 96 to 2400 Kbps, which covers the rate of standard network video streams. Flows of this kind last for a fixed period, even if the desired sending rate can be achieved. The other kind is bandwidth-demanding flows, which is the typical behavior of applications such as large file transfers. These flows are fixed in size but have no requirement on per-message latency or the message arrival order. In this case, multipath routing might be applied to achieve best possible throughput.

The prior two prototype applications listed in Table II are used to handle different types of flows, respectively. With the same trace of traffic on the overlay network, which generates 7200 flows in about 1 hour, we evaluate the *Stemflow* performances under different controller-relay server interaction schemes. We compare *Stemflow* to the OpenFlow-style of

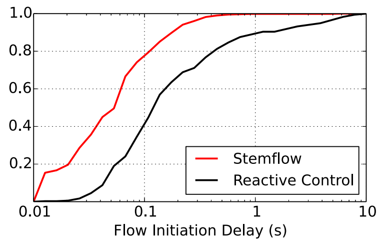


Fig. 9. CDF of flow initiation delays.

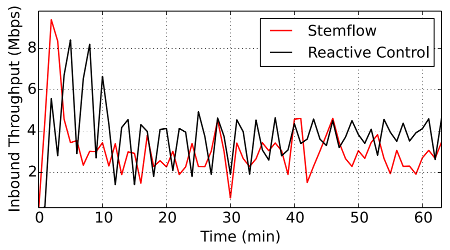


Fig. 10. Inbound traffic throughput log.

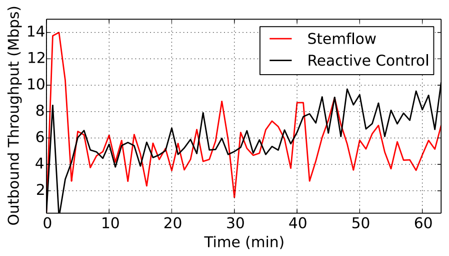


Fig. 11. Outbound traffic throughput log.

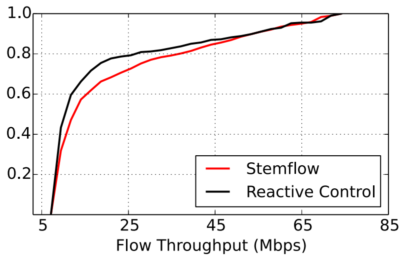


Fig. 12. CDF of flow throughput achieved by bandwidth-demanding flows.

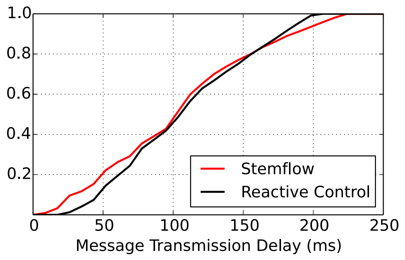


Fig. 13. CDF of flow throughput achieved by delay-sensitive flows.

interaction, which is the *reactive control* scheme based on query and responses. *Stemflow* is feasible to mimic the reactive control scheme by caching no pre-defined message processor objects on relay servers, because the default action to deal with an unknown *Stemflow* message is to query the controller.

Note that we compare *Stemflow* to the reactive control scheme only. Other scalable control plane solutions, especially the distributed controller designs in the software-defined network literature, are not feasible to be implemented since the forwarding decisions of both types of flows should be made with global knowledge.

2) Flow Initiation Delays: The flow initiation delay is one of the most significant overheads in OpenFlow-based networks. In *Stemflow*, the delay of a flow is calculated at the sender by measuring the time difference between sending of the first message and receiving of its acknowledgment from all desired destinations.

Fig. 9 illustrates the Cumulative Distribution Function (CDF) of the initiation delays of all generated flows. Note that the x-axis is in log scale. It shows that *Stemflow* significantly reduces the communication overhead at the time of flow initiation. In particular, nearly 80% flows in *Stemflow* are initiated within 100 ms, as compared to 40% without logic caching. The tail distribution of flow initiation delays is even more impressive. The 99th-percentile delay is less than 1 second in *Stemflow*. However, with reactive control, 1% of the flows will take nearly 10 seconds to initiate.

3) Controller Traffic Load: Throughout the experiment, we record the network traffic load on the controller to see how well the logic caching strategy can offload the control plane. Fig. 10 and 11 depict the inbound and outbound traffic throughput fluctuation during the experiment.

In both figures, *Stemflow* controller works at a relatively lower network traffic load except for the first 2 minutes after the system launches. It is reasonable because the controller has to transfer additional messages to cache control logic onto all relay servers. Once the logic caching is completed, the controller can work persistently at a lower workload.

It is interesting to see that the outbound traffic load on the *Stemflow* controller shows less benefit as compared to the inbound traffic. The reason for this phenomenon is that the *Stemflow* controller is proactively updating the relay servers with the latest global network knowledge.

4) Quality of Selected Routes: One may doubt that, in *Stemflow*, the relay servers might make less optimal forwarding decisions than the central controller would do. In Fig.12 and 13, we evaluate the achieved throughput for bandwidth-demanding flows and average message latency for delay-sensitive ones.

Without a doubt, *Stemflow* and the traditional control plane design achieve similar performances in terms of traffic engineering. The two CDFs of message latency is almost identical, while *Stemflow* can even achieve slightly higher throughput. It is because *Stemflow* can react to the local network events quickly: whenever a competing flow terminates, the relay server can reroute the traffic for multipath transmission quickly.

5) Fault Tolerance: We further experiment to test the fault tolerance of the system. In this test, we start five flows from a sender server and select the same route for data relay. At a particular time, we intentionally terminate one of the relay

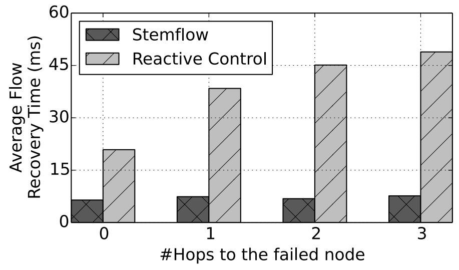


Fig. 14. Flow recovery time comparison.

servers on the path, and calculate the flow recovery time at the receiver. The RTT between the sender and the controller is about 10 ms.

The corresponding results are depicted in Fig. 14. It shows a similar trend for a different number of hops between the sender and the failed relay. In *Stemflow*, since each relay server is capable of rerouting immediately after the next-hop relay fails, it takes nearly no time for the flow to recover. Though the fallback path selection might not be optimal, it will be eventually rerouted to an optimal path after the controller takes over. However, without cached logic, all relay servers have to wait until the controller reacts. Also, some of them will have to hold the buffered messages at the mean time. Consequently, the flow cannot recover quickly from the failure.

VI. Related Work

Inter-datacenter Networks are critical network resources in the WAN. Preliminary experimental measurements in [10], [18] suggest inter-datacenter be an approachable new "backbone" in wide-area networks. Most of these networks are built upon dedicated links [15]. As traditional traffic engineering technologies result in sub-optimal routing patterns [22], Google [15] and Microsoft [13] resort to the software-defined networking technology.

Software-Defined Networking is a hot research topic in recent years. Starting from a campus network [19], it has already developed as a promising technology in next-generation networks [17]. The principle of software-defined networking is to decouple the packet forwarding intelligence from the hardware. OpenFlow [3] is the first standardized protocol designed for communications between the controller and the data plane. Static rules are installed and cached on the data plane switches, to perform longest prefix matches at runtime.

Despite the fine granularity of control, OpenFlow-based control plane suffers from *scalability problems*. Some workaround solutions attempt to offload the controller by applying the label-switching [4]–[6], [9] technology or a distributed control plane [12], [20]. However, they are till far from effective in dealing with traffic in production [8].

Reconfigurable Dataplanes have emerged as the future of OpenFlow 2.0 [8]. With new advancements in the hardware, switches today are able to process packets using reconfigurable logic, instead of static rules, at the line rate [25]. Following this direction, network programming primitives (*e.g.,* FAST [21], P4 [8], Probabilistic NetKAT [11], Domino [24]) and compilers (*e.g.,* SNAP [16]) are proposed.

Our design of *Stemflow* is greatly inspired by this line of work. However, due to hardware constraint, sophisticated algorithms are difficult to be implemented with the limited number of operations [23]. In *Stemflow*, taking advantage of the overlay data plane which is more easily programmable and reconfigurable, the controller-data plane interaction scheme is designed to be more flexible. It adapts better to the inter-datacenter overlay environment, getting rid of the unnecessary complexity incurred by processing multiple-layer packet headers.

VII. Concluding Remarks

We present the concept, design, implementation, and evaluation of *Stemflow*, a new system framework that provides easy access to the inter-datacenter overlay resources, designed specifically for bandwidth-intensive applications. Easily accessible via standard protocols, *Stemflow* features a software-defined overlay network architecture, and is designed to significantly improve application performance over inter-datacenter networks, while ensuring the flexibility of centralized control. For a real-world experimental evaluation, we deployed *Stemflow* across geographically distributed datacenters, and showed that Stemflow helps achieve up to 8.8x throughput improvements as compared to direct TCP connections, and allows for customized routing, flow scheduling, and bandwidth allocation algorithms to be deployed. We believe that the controller-data plane interaction scheme in *Stemflow* represents a further step towards a flexibly programmable data plane in software switches, complementing current standards for hardware switches such as OpenFlow 2.0.

References

[1] *Network Time Protocol (Version 3) Specification, Implementation and Analysis*. Accessed: Mar. 27, 2017. [Online]. Available: https://tools.ietf.org/html/rfc1305

[2] *Open Network Foundation Official Website*. Accessed: May 6, 2015. [Online]. Available: https://www.opennetworking.org/

[3] *OpenFlow White Paper*. Accessed: Mar. 27, 2017. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[4] K. Agarwal, C. Dixon, E. Rozner, and J. Carter, "Shadow MACs: Scalable label-switching for commodity ethernet," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw.-Defined Netw. (HotSDN)*, 2014, pp. 157–162.

[5] D. O. Awduche, "MPLS and traffic engineering in IP networks," *IEEE Commun. Mag.*, vol. 37, no. 12, pp. 42–47, Dec. 1999.

[6] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus, *Requirements for Traffic Engineering Over MPLS*, document IETF RFC 2702, Sep. 1999. Accessed: Oct. 10, 2017. [Online]. Available: http://www.ietf.org/rfc/rfc2702.txt

[7] M. Berman *et al.*, "GENI: A federated testbed for innovative network experiments," *Comput. Netw.*, vol. 61, pp. 5–23, Mar. 2014.

[8] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[9] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: A retrospective on evolving SDN," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw.-Defined Netw. (HotSDN)*, 2012, pp. 85–90.

[10] Y. Feng, B. Li, and B. Li, "Airlift: Video conferencing as a cloud service using inter-datacenter networks," in *Proc. IEEE 20th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2012, pp. 1–11.

[11] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic NetKAT," in *Proc. Eur. Symp. Programm. Lang. Syst.*, 2016, pp. 282–309.

[12] S. H. Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw.-Defined Netw. (HotSDN)*, 2012, pp. 19–24.
[13] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 15–26.
[14] N. Hu and P. Steenkiste, "Evaluation and characterization of available bandwidth probing techniques," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 6, pp. 879–894, Aug. 2003.
[15] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 3–14.
[16] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *Proc. 12th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2015, pp. 103–115.
[17] D. Kreutz, F. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.
[18] Z. Liu, Y. Feng, and B. Li, "Bellini: Ferrying application traffic flows through geo-distributed datacenters in the cloud," in *Proc. IEEE GLOBECOM*, Dec. 2013, pp. 1753–1759.
[19] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.
[20] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman, "Application-Aware Data Plane Processing in SDN," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw.-Defined Netw. (HotSDN)*, 2014, pp. 13–18.
[21] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw.-Defined Netw. (HotSDN)*, 2014, pp. 61–66.
[22] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. Maltz, "Latency inflation with MPLS-based traffic engineering," in *Proc. ACM Internet Meas. Conf. (IMC)*, 2011, pp. 463–472.
[23] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *Proc. USENIX NSDI*, 2017, pp. 67–82.
[24] A. Sivaraman *et al.*, "Packet transactions: High-level programming for line-rate switches," in *Proc. ACM SIGCOMM*, 2016, pp. 15–28.
[25] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "DC.p4: Programming the forwarding plane of a data-center switch," in *Proc. ACM Symp. Softw. Defined Netw. Res. (SOSR)*, 2015, Art. no. 2.

Shuhao Liu (S'17) received the B.Eng. degree from Tsinghua University in 2012. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Toronto. His current research interests include software-defined networking and big data analytics.

Baochun Li (F'15) received B.E. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995, the M.S. and Ph.D. degrees from the Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, in 1997 and 2000. Since 2000, he has been with the Department of Electrical and Computer Engineering, University of Toronto, where he is currently a Professor. His research interests include cloud computing, large-scale data processing, computer networking, and distributed systems. He is a member of ACM. In 2000, he was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the Field of Communications Systems. In 2009, he was a recipient of the Multimedia Communications Best Paper Award from the IEEE Communications Society, and a recipient of the University of Toronto McLean Award. He holds the Nortel Networks Junior Chair in Network Architecture and Services from 2003 to 2005, and the Bell Canada Endowed Chair in Computer Engineering since 2005.