SDN Solutions for Switching Dedicated Long-Haul Connections: Measurements and Comparative Analysis

Nageswara S. V. Rao

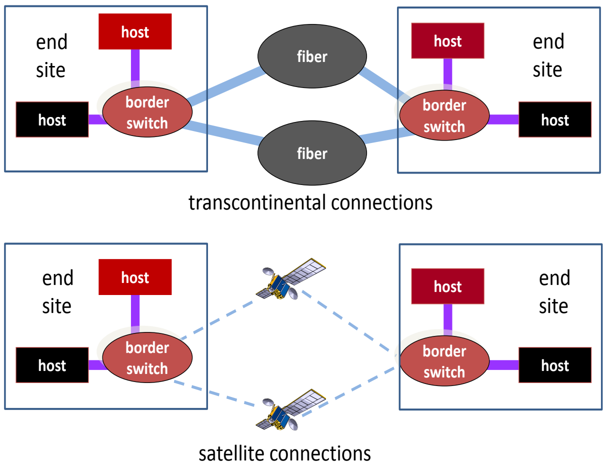
Oak Ridge National Laboratory
Oak Ridge, TN 37831, USA
Email: raons@ornl.gov

***Abstract*—We consider a scenario of two sites connected over a dedicated, long-haul connection that must quickly fail-over in response to degradations in host-to-host application performance. The traditional layer-2/3 hot stand-by fail-over solutions do not adequately address the variety of application degradations, and more recent single controller Software Defined Networks (SDN) solutions are not effective for long-haul connections. We present two methods for such a path fail-over using OpenFlow-enabled switches: (a) a light-weight method that utilizes host scripts to monitor application performance and dpctl API for switching, and (b) a generic method that uses two OpenDaylight (ODL) controllers and REST interfaces. For both methods, the restoration dynamics of applications contain significant statistical variations due to the complexities of controllers, north bound interfaces and switches; they, together with the wide variety of vendor implementations, complicate the choice among such solutions. We develop the impulse-response method based on regression functions of performance parameters to provide a rigorous and objective comparison of different solutions. We describe testing results of the two proposed methods, using TCP throughput and connection rtt as main parameters, over a testbed consisting of HP and Cisco switches connected over long-haul connections emulated in hardware by ANUE devices. The combination of analytical and experimental results demonstrate that the dpctl method responds seconds faster than the ODL method on average, even though both methods eventually restore original TCP throughput.**

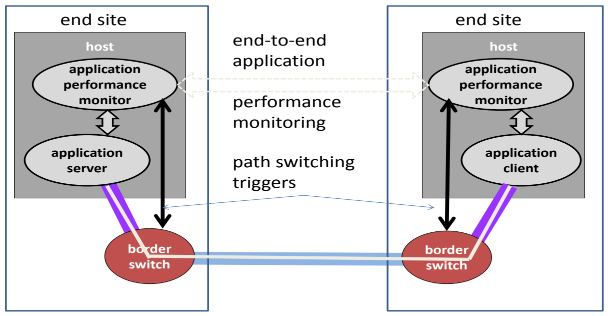
Keywords*–*Software defined networks; OpenFlow; Opendaylight; controller; long-haul connection; impulse-response; testbed.

I. Introduction

We consider scenarios where two remote sites are connected over a dedicated long-haul connection with hundreds of millisecond latency, such as a transcontinental fiber or satellite link [1], as illustrated in Figure 1(a). Different client-server application pairs are executed at different times on host systems located at the sites, which range from data transfers to on-line instrument monitoring to messaging. Applications may incorporate different methods to account for network losses and jitter; for example, they may utilize TCP or UDT for guaranteed delivery, UDP for loss tolerant cases, custom buffering methods and others at application level to account for jitter. Furthermore, their performance may be optimized or customized to connection parameters such as latency, jitter and loss rate; for example, TCP parameters may be tuned for long-haul connections and buffers of interactive codes may be tuned for long latencies and jitter of satellite links. The connection quality can degrade due to a variety factors such as equipment failures, weather conditions, and geographical events, which may be reflected in host-to-host application performance. Indeed, the client-server application pairs may respond differently to various degradations, such as decreased throughput of file transfers, increased jitter in streaming, loss



(a) host-to-host application pairs



(b) two connection modalities

Figure 1. Two sites connected over long-haul connections.

of end-to-end control in computational steering, and in some cases (such as messaging) having very little effect. As a mitigation strategy, a physically diverse and functionally equivalent *standby* path is switched to when the performance of currently running application pairs degrades.

The performances of application pairs are continuously monitored on host systems, and the current *primary* path is switched out when needed, for example, by modifying Virtual Local Area Networks (VLAN) and route tables on border switches and routers, respectively. In our use cases, human operators watch host-level performance monitors, and invoke Command Line Interface (CLI) commands or web-based interfaces of network devices for path switching. Typically, the path fail-overs are accomplished either by manual configuration or through device-specific scripts. Since triggers for path switching are dynamically generated by application pairs, they are not adequately handled by conventional hot standby layer-2/3 solutions that solely utilize connection parameters. For example, certain losses may be tolerated by messaging applications but not by monitoring and control applications of instruments and sensors. Currently, the design and operation of such application-driven fail-over schemes require a detailed knowledge of host codes, and the specialized interfaces and APIs of switches, such as custom TL1, CURL and python scripts, which currently vary significantly among vendor products. Furthermore, in our use cases such fail-over operations must be coordinated between two physically-separated operations centers located at the end sites. The combination of recent advances in host and network virtualization technologies [2] offers very promising and potentially game changing solutions to seamlessly automate the dynamic fail-over workflows that integrate diverse application monitors and network elements.

We are interested in exploiting the network and host virtualization layers to unify and automate such monitoring and path switching operations. Automated scripts for these tasks provide the following advantages over current practices: (i) improved response time, since scripts can be executed much faster than manual configurations, (ii) reductions in performance degradations due to human errors in application monitoring and path switching, and (iii) reductions in site operations costs of host systems and network devices.

A. SDN Solutions

The rapidly evolving *Software Defined Networks* (SDN) technologies [3], [4] seem particularly well-suited for automating the path switching tasks, when combined with host monitoring codes. In particular, the northbound interfaces of SDN controllers can be used to communicate the path degradations information to trigger path switching; then, the path can be switched by installing flow entries that divert traffic onto the standby path using the southbound controller interfaces [5]. Thus, SDN technologies provide two distinct advantages over current network operations:

(a) trigger modules of new applications can be "dropped in place" with no no major software changes by using communications via generic northbound interfaces, and
(b) switches from different vendors with virtual interfaces can be simply be swapped, avoiding the re-work often needed to account for custom interfaces and operating systems.

While the problem space of our use cases is somewhat straightforward, their SDN solution space is much more complex: due to the rapid developments in underlying technologies, these is a wide array of choices for controllers and switches, which in turn leads to a large number of solution combinations. Indeed, their complexity and variety requires systematic analysis and comparison methods to assess their operational effectiveness and performance, such as recovery times. In addition, compared to certain data-center and network provisioning scenarios for which SDN technologies have been successfully applied, these long-haul scenarios present additional challenges. First, single controller solutions are not practical for managing the border switches at end sites due to the large latency. Second, solutions that require a separate control-plane infrastructure between the controllers and switches are cost prohibitive, in sharp contrast to the connection-rich data-center or Internet scenarios.

B. Outline of Contributions

In this paper, we present automated software solutions for path fail-over by utilizing two controllers, one at each site, that are coordinated over a single connection through measurements. We first describe a light-weight, custom designed *dpctl method*[1] for OpenFlow border switches that uses host Linux bash scripts to: (i) monitor the connection parameters, such as rtt or TCP throughput, at the host-level and detect degradations that require a fail-over, and (ii) utilize dpctl API to install and delete flow entries on the border switches to implement path fail-over when needed. This script is about hundred lines of code, which makes it easier to analyze for its performance and security aspects. We then present a more generic *ODL method* that utilizes two OpenDaylight Hydrogen (ODL) controllers [6] located at the end sites. We use REST interface of ODL controller to communicate the trigger information for path switching in the form of new OpenFlow entries to be installed on border switches. We also utilize Linux bash scripts to monitor the connection performance to generate fail-over triggers, and invoke python REST API scripts to communicate new flow entries to ODL controllers. The executional path of this approach is more complex compared to the dpctl method since it involves communications using both northbound and southbound ODL interfaces and invoking several computing modules within ODL software stack. Thus, a complete performance and security analysis of this method requires a closer examination of much larger code base that includes both host scripts and corresponding ODL modules, including its embedded http server.

We present implementation and experimental results using a testbed consisting of Linux hosts, HP and Cisco border switches, and ANUE long-haul hardware connection emulation devices. We utilize TCP throughput as a primary performance measure [2] for the client-server applications, which is effected by the connection rtt and jitter possibly caused by path switching, and the available path capacity. Experimental results show that both dpctl and ODL methods restore the host-to-host TCP throughput within seconds by switching to the standby connection after the current connection's RTT is degraded (by external factors). However, the restoration dynamics of TCP throughput show significant statistical variations, primarily as a result of interactions between the path switching dynamics of controllers and switches, and the highly non-linear dynamics of TCP congestion control mechanisms [7]–[9]. As a result, direct comparisons of individual TCP throughput time traces corresponding to fail-over events are not very instructive in reflecting the overall performance of the two methods.

To objectively compare the performance of these two rather dissimilar methods, we propose the *impulse-response*

[1]Note that dpctl is originally intended for diagnosis purposes, which we utilize as a controller.

[2]The overall approach is applicable to other application-level performance measures such as response times, which typically degrade under connection disruptions and recover when connection is restored. Our choice of TCP is based on its widespread use for guaranteed packet delivery and its rich dynamics.

method that captures the average performance by utilizing measurements collected in response to a train of path degradation events induced externally. We establish a statistical basis for this method using the finite-sample theory [10] by exploiting the underlying monotonic properties of performance parameters during the degradation and recovery periods. This analysis enables us to objectively conclude that on the average the dpctl method restores the TCP throughput several seconds faster than the ODL method for these scenarios. This paper is an expanded version of an earlier conference paper [1] with additional explanations and details of SDN implementations, and it also provide a complete derivation of the performance equations for the impulse response method using finite sample statistical analysis.

The organization of this paper is as follows. Two-site scenarios with dedicated long-haul connections are described in Section II. A coordinated controllers approach for connection fail-over, and its implementation using dpctl and ODL methods are described in Section III. An experimental testbed consisting of Linux servers and HP and Cisco switches is described in Section IV-A, and the results of experiments using dpctl and ODL methods using five different configurations are presented in Section IV-B. The impulse response method to assess the overall fail-over performance is presented in Section V-A, and its statistical basis is presented in Section V-B. Conclusions are presented in Section VI.

II. Long-Haul Connection Switching

We consider scenarios consisting of two sites connected over a long-haul connection such as transcontinental fiber routes or satellite connections as shown in Figure 1(a). The sites house multiple workstations on which server and client applications are executed, which form the client-server application pairs over the long-haul connection as shown in Figure 1(b). The client-server application performance depends on the quality of the connection specified by parameters such as latency, loss rate, and jitter. These connection parameters may degrade due to external factors such as equipment failures, fiber cuts and weather events. Such factors will be reflected in the degradation in client-server performance, which may be detected by the performance monitoring software implemented at application-level, for example, using Linux scripts. To protect against such factors, a parallel standby path is provisioned that can be switched-over to when performance degradations are detected. The border switches or routers at the sites are connected to both the primary long-haul connection that carries the traffic, and the stand-by connection whose traffic can be activated as needed. In our case, these connections are implemented as layer-2 VLANS at the border switches, which can be modified to implement the fail-over. All traffic between the sites is carried by the single long-haul connection, including client-server communications and other traffic needed for coordinating network operations; in particular, it is not cost-effective to provision a separate “control” connection for supporting the network configuration operations unlike in other cases, such as in UltraScienceNet [11] a decade ago and more recently in SDN based solutions.

Application codes on host systems continually monitor the client-server performance, such as iperf for TCP throughput and UDP loss rate. Under path degradations, these parameters would be out of normal range, and such events are detected

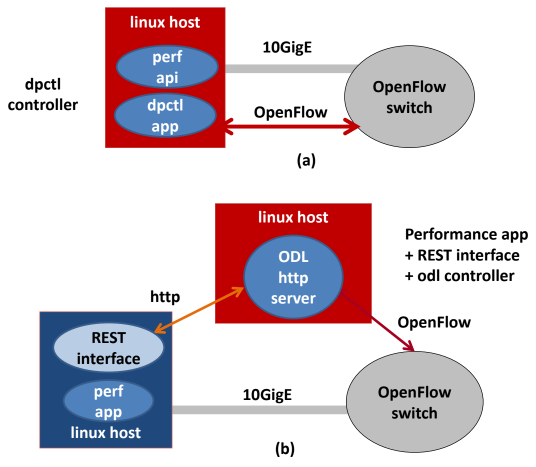


Figure 2. Configurations of dpctl and ODL controllers.

and alerts are sent to network operations. Typically, human operators receiving the alerts modify the VLANS on border switches to implement path switching, for example, by invoking CLI or TL1 or curl scripts, or manually making the changes through CLI or web interfaces. Due to different organizational zones at the sites and the long separation between them, it is not practical for a single network operations center to handle connection switching at both sites, particularly, if the same “degraded” connection is used for these communications as well. Instead, such tasks are typically coordinated between the two site organizations using other means such as telephone calls. Due to the multi-step process needed here, the fail-over operation can take anywhere between few minutes to hours. Thus, it is highly desirable to automate the entire fail-over work flow that includes application-level monitoring and network-level switching as described in Introduction section.

III. Coordinated SDN Controllers

For the long-haul scenarios considered here, a single controller solution is not effective, although such approaches with stable control-plane connections have been effective in path/flow switching over local area networks using OpenFlow [12], [13] and cross-country networks using customized methods [11], [14]. Since the controller has to be located at a site, when primary connection degrades, it may not be able to communicate effectively with the remote site. Our approach is to utilize two controllers, one at each site, which are “indirectly” coordinated based on the monitored application-level performance parameters. When path degradation is inferred by a host script, the controller at that site switches to the fail-over path by installing the appropriate flow entries on its border switch. If the primary path degrades, for example, resulting in increased latency or loss rate, its effect is typically detected at both hosts by the monitors, and both border switches fail-over to the standby path approximately at the same time. If border switch at one site fails-over first, the connection loss will be detected at the other site which in turn triggers the fail-over at the second site. Also, one-way failures lead to path switching at one site first, which will be seen as a connection loss at the other site, thereby leading to path switching at that site as well. Due to recent developments in SDN technologies, both in open software [3], [4], [15] and specific vendor implementations [16], [17], there are many different ways such a solution can be implemented. We restrict here to OpenFlow solutions based on open standards and software [12].

A. dpctl Method

As a part of OpenFlow implementation, some vendors support dpctl API which enables hosts to communicate with switches to query the status of flows, insert new flows and delete existing flows. It has been a very useful tool primarily for diagnosing flow implementations by using simple host scripts; however, some vendors such as Cisco do not provide dpctl support. We utilize dpctl API in a light-weight host script that constantly monitors the connection rtt and detects when it crosses a threshold and invokes dpctl to implement the fail-over as shown in Figure 2(a). The OpenFlow entries for switching to the standby path are communicated to the switch upon the detection of connection degradation. This script consists of under one hundred lines of code and is flexible in that the current connection monitoring module can be replaced by another one such as TCP throughput monitor using iperf. Compared to methods that use separate OpenFlow controllers, this method compresses both performance monitoring and controller modules into one script, and thereby avoids the northbound interface altogether; for ease of reference, we refer to this host code as the *dpctl controller*. This small footprint of the code makes is easier to analyze both from performance and security perspectives. Also, the executional path of this code is very simple since it involves application monitoring directly followed by communications with the switch; in particular, this method does not require a separate controller that is constantly running at the end sites.

B. OpenDaylight Method

We now present a method that utilizes two ODL Hydrogen controllers and REST interfaces to implement fail-over functionality using OpenFlow flows as shown in Figure 2(b). ODL is an open source controller [6] that communicates with OpenFlow switches [3], and is used to query, install and delete flow entries on them using its southbound interface. The applications communicate with ODL controller via the northbound interface to query, install and delete flows. ODL software in our case runs on linux workstations called the controller workstations, and the application monitoring codes can be executed on the same workstation in the *local* mode or can be executed on a different workstation in the *remote* mode.

The same performance monitoring codes of the dpctl method above are used in this case to detect path degradations but are enhanced to invoke python code to communicate new flows for switching paths to ODL controllers via REST interfaces; the content of these flow entries are identical to previous case. Thus, both the software and executional paths of this method are much more complicated compared to previous case, and also the ODL controllers are required to run constantly on the servers at end sites. Also, this Hydrogen

[3]ODL provides interfaces to much broader classes of switches and applications, but we limit our discussions to the functionalities that are directly related to long-haul scenarios described in Section II.

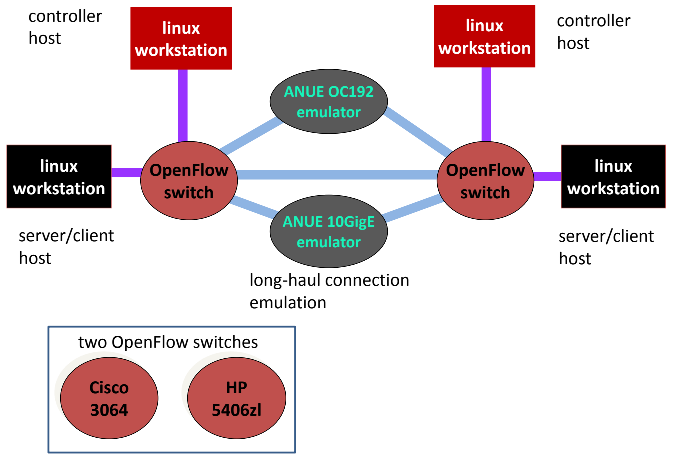


Figure 3. Testbed of two sites connected over local fiber and emulated connections.

ODL code[4] is much more complex to analyze since it involves not only the REST scripts but also the ODL stack which by itself is a fairly complex software. The execution path is more complex since it involves additional communication over both northbound and southbound interfaces of ODL controllers.

The dpctl and ODL methods represent two different fail-over solutions, and the choice between them depends on their recovery performance in response to triggers. In the next section, we describe a set of experiments using HP and Cisco switches that highlight the performances of controllers and switches. However, a direct comparison of the measurements between various configurations is complicated by their statistical variations, which we account for using the impulse response method described in Section V.

IV. Experimental Results

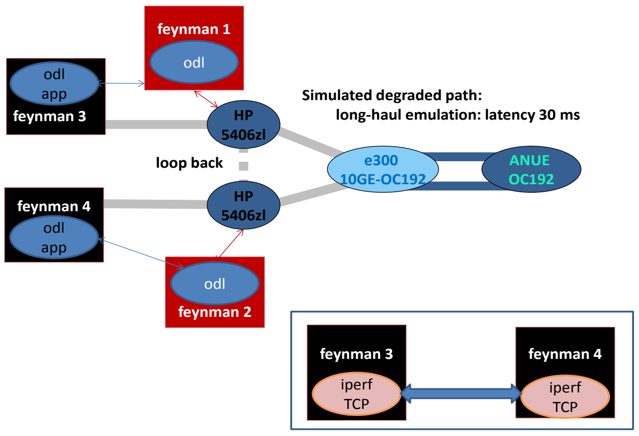
In this section, we first describe the details of testbed and tests using dpctl and ODL controllers, and then describe the experimental results.

A. Emulation Testbed

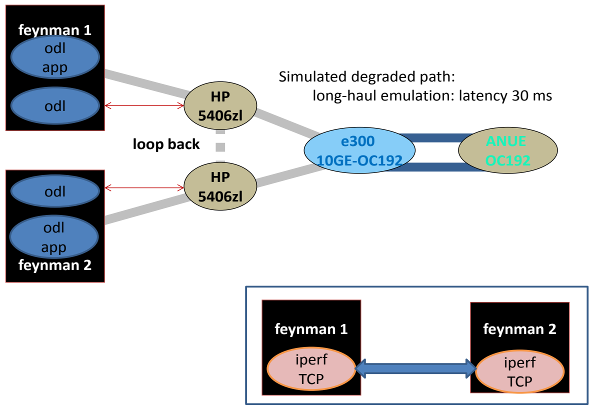
The experimental testbed consists of two site LANs each consisting of multiple hosts connected via 10GigE NICs to the site's border switch. The border switches are connected to each other via local fiber connection of few meters in length and ANUE devices that emulate long-haul connections in hardware, as shown in Figure 3. Tests are performed in configurations that use pairs of HP 5064zl and Cisco 3064 devices as border switches. These switches are OpenFlow-enabled but only HP switches support dpctl interface. We only utilize OC192 ANUE device in our tests, which can emulate rtts in the range of [0-800] milliseconds with a peak capacity of 9.6 Gbps. These devices are utilized primarily to emulate the latencies of long-haul connections, both transcontinental

[4]Later ODL releases starting with Helium, including Lithium and Beryllium, support more agile code builds wherein only the needed service modules are loaded (as karaf features [6]), thereby significantly reducing its size. However the execution path remains the same as in Hydrogen.

fiber and satellite connections, to highlight the overall recovery dynamics. However, no efforts are made to highlight their capacity differences, for example, by limiting the latter to typical satellite connection capacities, which could have resulted in somewhat muted dynamics. The conversion between 10GigE LAN packets from the border switches and long-haul OC192 ANUE packets is implemented using a Force10 E300 switch, which provides 10GigE LAN-PHY and WAN-PHY conversion as shown in Figure 4.



(a) Configurations C-E: remote



(b) Configuration B: local

Figure 4. Remote and local modes of ODL controller configurations.

Two classes of Linux hosts are connected to the border switches. The controller hosts (feynman1 and feynman2) are utilized to run ODL controllers, and client and server hosts (feynman3 and feynman4) are used to execute monitoring and trigger codes along with client server codes, for example iperf clients and servers. Five different configurations are employed in the tests as shown in Table I. The *dpctl* tests utilize only the client and server hosts to execute both monitoring and switching codes. In these tests, dpctl is used to communicate flow modification messages between the hosts and HP 5064zl switches, and Cisco switches are not used. Configuration A corresponds to these tests with the monitoring and dpctl

Table I. Five test configurations with two controllers, two connection degradation methods and two switch vendors.

test configuration	controller method	path degradation	switch vendor
A	dpctl	path switch	HP
B	ODL local	path swith	HP
C	ODL remote	path switch	HP
D	ODL remote	rtt extension	HP
E	ODL remote	rtt extension	Cisco

scripts running on server/client hosts, and it uses HP border switches. For *ODL remote mode* tests, the monitoring codes on client/server hosts utilize REST interface to communicate flow message needed for fail-over; both HP and Cisco switches are used in these tests. Configurations C-E implement these tests, which employ ODL controllers running on controller hosts and monitoring scripts running on server/client hosts. In *ODL local mode* tests, the monitoring and client/server codes are executed directly on control hosts. Configuration B implements these tests, and it is identical to Configuration C except its scripts are executed on controller hosts. The measurements in Configurations B and C are quite similar, and hence we mostly present results of the latter.

In the experiments, connection degradation events are implemented by external codes using two different methods:

(a) *Path switching using dpctl:* The current physical path with a smaller rtt is switched to a longer emulated path, whose rtt is sufficient to trigger the fail-over. This switching is accomplished by using dpctl or ODL by installing OpenFlow entries on the border switches to divert the flow from the current path to longer path. The packets enroute on the current path will be simply be dropped and as a result the short-term TCP throughput becomes zero. After the fail-over, the path is switched back to the original path, and the TCP flow recovers gradually back to previous levels.

(b) *RTT extension using curl scripts:* The current connection's rtt is increased by changing the setting on ANUE device to a value above the threshold to trigger the failover. This is accomplished using http interface either manually or using curl scripts. Unlike the previous case, the packets enroute on the current path are not dropped but are delayed; thus, the instantaneous TCP throughput does not always become zero but is reduced significantly. After the fail-over, the original rtt is restored, and TCP throughput recovers gradually to previous levels.

The first degradation method using dpctl to switch the paths is only implemented for configurations with HP border switches in Configurations A - C. The second method is used for both HP and Cisco system in Configurations D and E, and since the curl scripts are used here to change delay settings on ANUE devices, and border switches are not accessed.

B. Controller Performance

TCP throughput measurements across the long-haul connection are constantly monitored using iperf. The default CUBIC congestion control module [18] for Linux hosts is used in all tests. The rtt between end hosts is also constantly monitored using ping, and path switching is triggered when it crosses a set threshold; this module can be replaced by a more general application-based trigger module, for example, to detect when throughput falls below or jitter exceeds thresholds. The path degradations are implemented as periodic impulses and the responses are assessed using the recovery profiles of TCP throughput captured at one second intervals. Also, the ANUE dynamics in extending the rtt affect the TCP throughput recovery, and we obtain additional baseline measurements by utilizing a direct fiber connection that avoids packets being routed through ANUE devices. Thus, TCP throughout traces in our tests capture the performances of: (a) controllers, namely, dpctl and ODL, in responding to fail-over triggers from monitoring codes, and in modifying the flow entries on switches, typically by deleting the current flows and inserting the ones for standby path, and (b) border switches in modifying the flows entries in response to controller's messages and re-routing the packets as per new flow entries.

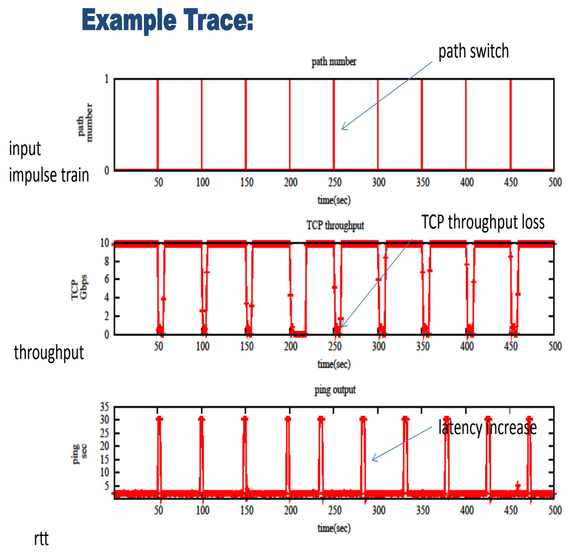


Figure 5. Trace of impulse response of TCP throughput for dpctl method with local primary path and path switching degradation.

An example TCP throughput trace of a test run in Configuration A is shown in Figure 5 for the dpctl method, with fiber connection as the primary path, and using the path switching degradation method. The connection rtt is degraded at the periodicity of 50 seconds by externally switching to the longer ANUE path, and the change is detected as shown in the bottom plot, which in turn triggers the fail-over. TCP throughput parameters on the hosts are tuned to achieve 10Gbps for the default rtt, and it degrades once the connection rtt is increased to 30ms after path switching. Upon the detection of increased rtt, the default path is restored, which in turn restores TCP throughput to the original value as shown in the middle plot of Figure 5. Note that throughput trace shows significant variations during the recovery periods following the fail-over, even in this simplest among the test configurations.

The restoration profiles of TCP throughput in these tests reflect the detection of connection degradation and fail-over, followed by the recovery response of the non-linear dynamics of CUBIC congestion control mechanism. Three different TCP recovery profiles from the tests are shown in Figure 6: (a)

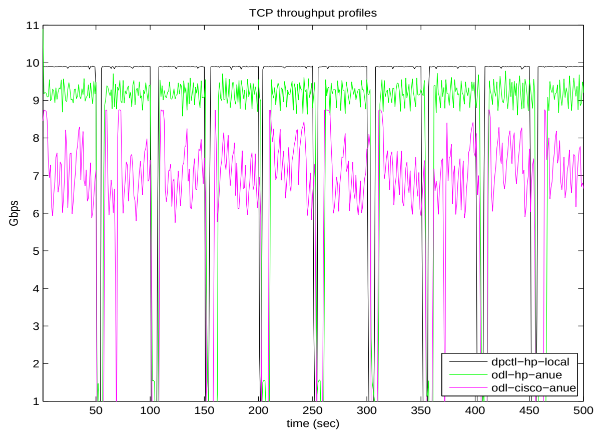
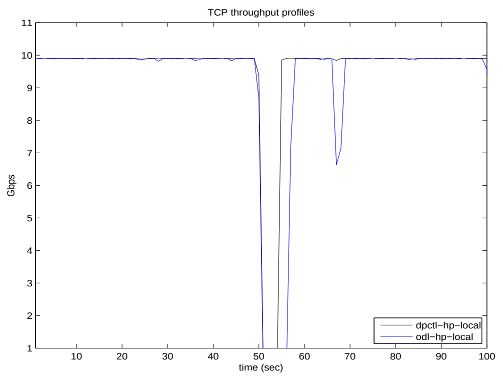


Figure 6. TCP throughput for dpctl method for configuration A and ODL methods for configuration D and E.

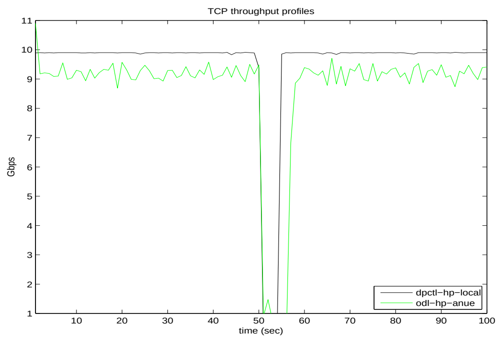
Configuration A: dpctl method using HP switches with connection degradation by path switching, (b) *Configuration D*: ODL method using HP switches with connection degradation by rtt extension, and (c) *Configuration E*: ODL method using Cisco switches with connection degradation by rtt extension. As seen in these plots, TCP response dynamics contain significant variations for different degradation events of the same configuration as well as between different configurations.

The individual TCP throughput recovery responses to single path degradation events reveal more details of the difference between configurations as shown in Figure 7. The delayed response of ODL method compared to dpctl method can be seen in Figure 7(a) for Configurations A and B. Since the packets in transit during the switching are simply lost during path switching, the instantaneous TCP throughput rapidly drops to zero for Configuration A. On the other hand, some of the packets in transit when rrt is extended are delivered, and as a result TCP throughput may be non-zero in some cases, as shown in Figure 7(b). Another aspect is that, TCP throughput recovers to 10Gbps when the direct fiber connection is used between the switches, but only peaks around 9 Gbps when packets are sent via ANUE connection with zero delay setting as shown in Figure 7(b). Also, the recovery profiles are different between HP and Cisco switches in otherwise identical Configurations E and F as shown in Figure 7(c). Thus, TCP dynamics depend both on the controller primarily in terms of recovery times, and on the switches in terms of the peak throughput achieved and its temporal stability.

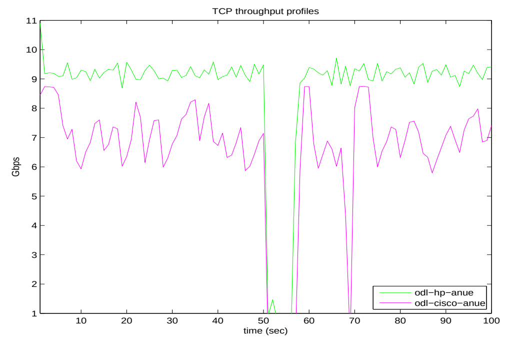
We now consider more details of the dpctl and ODL methods using HP switches with path switching degradation shown in Figure 7(a) with the primary fiber connection. Here, the TCP throughput becomes zero and rtt crosses the threshold immediately following the path switching degradation, and both controllers respond to the fail-over triggers and switch the path back to original fiber connection. Since these configurations are identical except the controllers, the recovery time of ODL method is a few seconds slower than dpctl method. However, the complex dynamics and statistical variations of TCP profiles make it harder to draw general conclusions about their comparative performance based on such single degradation events.



(a) Configurations A and B



(b) Configurations A and D



(c) Configurations D and E

Figure 7. Trace of impulse response of TCP throughput for dpctl method with local primary path and path switching degradation.

C. Switch Performance

TCP performance is effected by the path traversed by the packets between the border switches, in addition to its dependence on dpctl and ODL methods as described in the previous section (Figure 7(b)), thereby indicating the effects of the connection modality on client-server performance. In configurations A and B, the primary connection is few meters of fiber between the switches, and TCP throughput is restored to around 10 Gbps after the fail-over as shown in Figure 7(a). In Configurations D and E, the packets are sent through the emulated connection, which consists of long-haul Force10 E300 switch and OC192 ANUE emulator with the peak capacity of 9.6 Gbps. In this case, both peak value and the dynamics of TCP throughput are effected as shown in Figure 7(b); as expected, the peak is below 9.6 Gbps but there are significant variations in the throughput. Furthermore, the connection modality effects HP and Cisco switches differently as shown in Figure 7(c) in that the latter reached somewhat lower peak throughput and exhibited larger fluctuations [5].

Although we focussed on TCP throughput measurements in this section, the overall approach is applicable to other performance parameters such as latency in reporting sensor measurements, response times of remote control operations, and loss of quality in voice and video transmissions. In general, we consider that the chosen performance parameter degrades when the current connection properties degrade, and it recovers when stand-by connection is restored. This overall characterization is used to develop a method to systematically compare the measurements under different configurations.

V. Impulse Response Method

We present the *impulse response method* in this section that captures the overall recovery response by "aggregating" generic (scalar) performance measurements (such as TCP throughput as in previous section) collected in response to periodic connection degradations. It enables us to objectively compare the performances of different methods and devices by extracting the overall trends from measurement traces of multiple fail-over events. While our discussion is centered around TCP measurements described in the previous section, the overall approach is more generically applicable for comparing configurations with different controllers and switches, and it is particularly useful when the recovery throughput traces are not amenable to simple, direct comparisons.

A. Response Regression Function

A configuration X that implements the fail-over is specified by its SDN controller and switches that implement the fail-over, and also the monitoring and detection modules that trigger it. Let $\delta(t-iT), i=0,1,\ldots,n$ denote the input impulse train that degrades the connection at times $iT+T_D$, where t represents time, T is the period between degradations and $T_D < T$ is the time of degradation event within the period. Let $\mathbf{T}^X(t)$ denote the parameter of interest at time t, such as TCP throughput, as shown in Figure 6 for configurations X=A,D,E. Let $R^X(t) = \mathbf{B} - \mathbf{T}^X(t)$ denote the response that captures the "unrealized" or "unused" portion of the peak performance level $\mathbf{B}$. For example, over a connection with capacity $\mathbf{B}$ it is the residual bandwidth at time t above TCP throughput $\mathbf{T}^X(t)$; it is close to zero when throughput is close to the peak and it is close to $\mathbf{B}$ during the fail-over period when throughput is close to zero. We define the *impulse response function* $R^X(t)$ such that

$$R_i^X(t) = R^X(t-iT), t \in [0,T)$$

is the response to ith degradation event $\delta(t-iT)$, for $i = 0,1,\ldots,n$. An ideal impulse response function is also an

[5]No connection-specific TCP optimizations are performed in these tests, and it is quite possible that different optimizations may be needed to achieve comparable throughput in these two configurations.

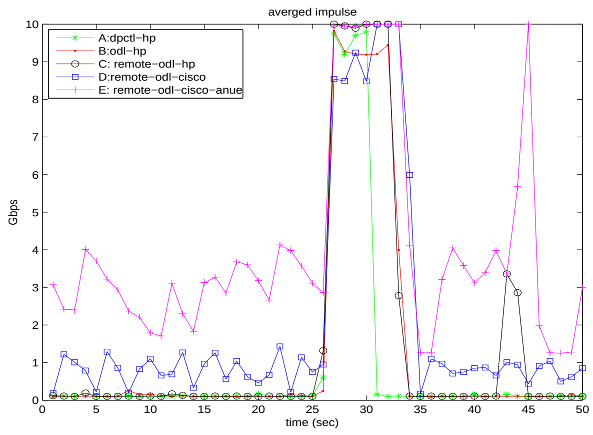


Figure 8. Examples of impulse response functions for Configurations A-E for a single path degradation.

impulse train that matches the input, wherein each impulse represents the instantaneous degradation detection, fail-over and complete recovery the parameter. But in practice, each $R_i^X(t)$ is a "flattened" impulse function whose shape is indicative of the effectiveness of the fail-over. In particular, its leading edge represents the effect of degradation and its trailing edge represents the recovery, and the narrower this function is the quicker is the recovery. Examples of $R_1^X(.)$ are shown Figure 8 for configurations A-E; these TCP measurements show significant temporal variations that persist across the different degradation events, which make it difficult to objectively compare these single-event time plots. In general, such variations are to be expected in other performance parameters such as latency and response time associated with sensor and control applications. Nevertheless, certain general trends seem apparent such as the faster TCP response of the dpctl method compared to the ODL method.

We define the *response regression* of configuration X as

$$\bar{R}^X(t) = E\left[R_i^X(t)\right] = \int R_i^X(t) d\mathbf{P}_{R_i^X(t)},$$

for $t \in [0,)$, where the underlying distribution $\mathbf{P}_{R_i^X(t)}$ is quite complex in general since it depends on the dynamics of controllers, switches, end hosts, application stack and monitoring and detection modules that constitute X. It exhibits an overall decreasing profile for $t \in [0, T_D + T_I]$ followed by an increasing profile for $t \in (T_D + T_I, T]$, where T_I is the time needed for the application to react to connection degradation. After the fail-over, TCP measurements exhibit an overall increasing throughput until it reaches its peak as it recovers after becoming nearly zero following the degradation. We consider that a similar overall behavior is exhibited by the general performance parameters of interest.

We define the *response mean* $\hat{R}_i(t)$ of $\bar{R}_i(t)$ using the discrete measurements collected at times $t = j\delta$, $j = 0, 1, ..., T/\delta$, as

$$\hat{R}^X(j\delta) = \frac{1}{n} \sum_{i=1}^{n} \left(R_i^X(j\delta)\right)$$

which captures the average profile. Examples of $\hat{R}_i^X(.)$ for TCP throughput are shown Figure 9 for different configurations

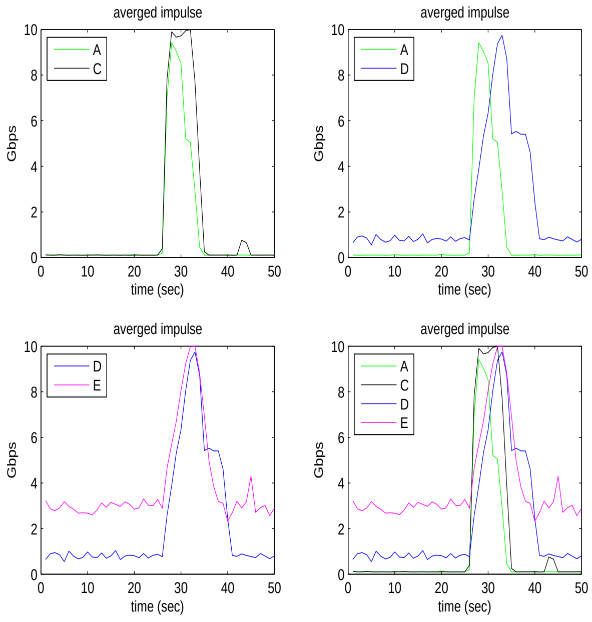
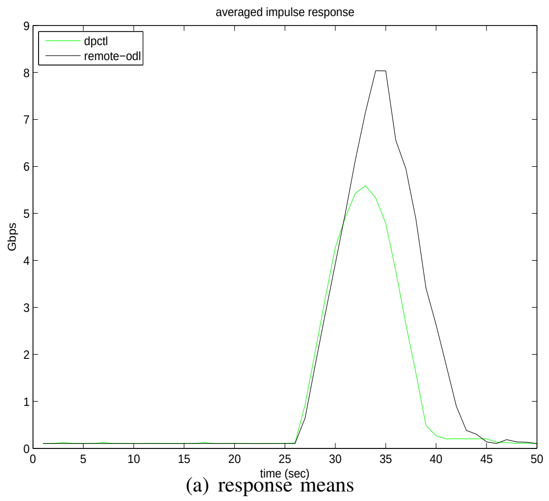
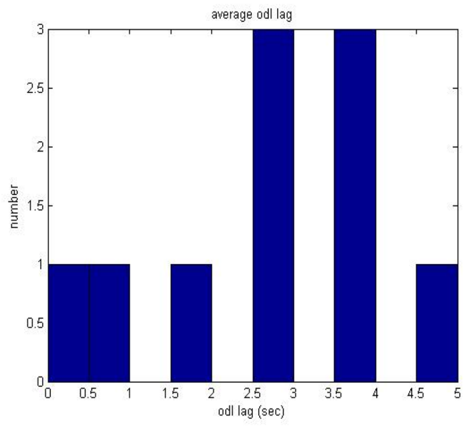


Figure 9. Impulse response regressions for $n = 10$ and $T = 50$ sec: (a) top-left: A and C, (b) top-right: A and D, (c) bottom-left: D and E , and (d) bottom-right: all.



(a) response means



(b) delay histogram

Figure 10. Comparison of response means of dpctl (configuration A) and ODL (configuration C) methods using 100 path degradations with $T = 50$ seconds.

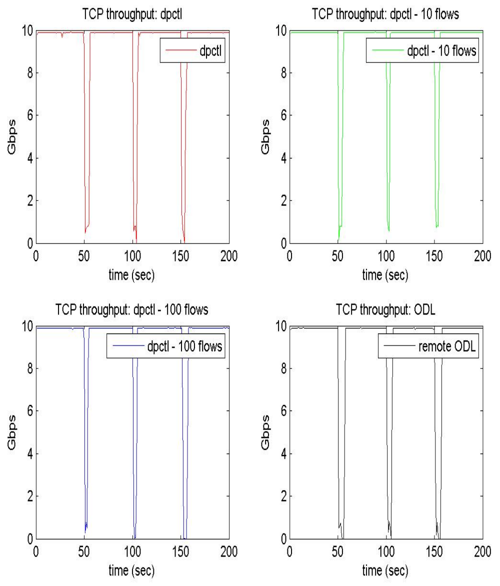
based on 10 path degradations with $T = 50$ seconds between them, which show the following general trends.

- The dpctl method responds seconds faster than ODL method as indicated by its sharper shape although their leading edges are aligned as shown in Figure 9(a).
- The connection degradation implemented by the rtt extension has a delayed effect on reducing the throughput compared to the path switching degradation method as indicated by its delayed leading edge in Figure 9(b).
- The dynamic response of regression profiles of HP 5604zl and Cisco 3064 switches are qualitatively quite similar as shown in Figure 9(c), but the latter achieved somewhat lower peak throughput overall; note that the larger throughput variations at individual switching events of the latter case are "averaged" in these plots.

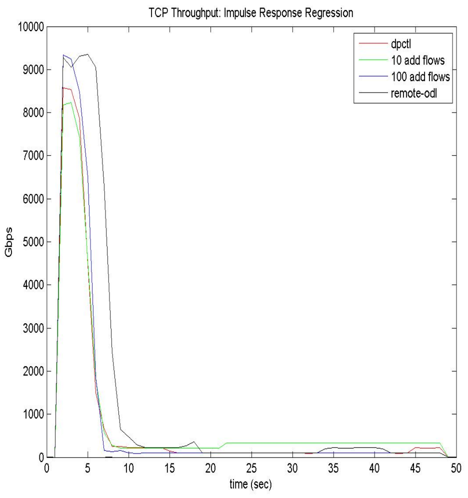
In view of the faster response of dpctl method, we collected additional measurements in configurations A and C using 100 path degradations, and the resultant response means are somewhat smoother compared to 10 degradations as shown in Figure 10 (a) for both dpctl and ODL methods. Furthermore, the response mean of ODL method remained consistent with more measurements, and a histogram of the relative delays of ODL method compared to dpctl method is plotted in Figure 10(b), and they are in the range of 2-3 seconds in majority of cases. These measurements clearly show the faster response of the dpctl method for these scenarios. Such performance is expected because of its much simpler code and execution path, both in terms of computation and communication.

In general, dpctl is primarily intended for diagnostic purposes and has not been used for control, and as a result its performance for the latter has not been investigated much, in particular, its scalability with respect to the number of flow entries. We tested the effectiveness of dpctl method with respect to the number of flow entries used for path switching; in above tests, for each path fail-over, two flow entries are used by both dpctl and ODL methods on each border switch. We increased the number of additional flow entries to 10 and 100 for each fail-over in dpctl method, and the resultant TCP throughput measurements are shown in Figure 11(a), along with those of the original ODL method. The impact of additional flow entries in dpctl method is not apparent from a visual inspection of these plots, primarily due to TCP throughput variations. The response means for these cases are plotted in Figure 11(b) which show no significant differences due to the additional flow entries. But, they show that the response of dpctl method is still 2-3 seconds faster than ODL method on average even with the additional flow entries.

From an engineering perspective, the above performance comparisons based on the measurements seem intuitively justified, but the soundness of such conclusions is not that apparent. In the next section, we provide a statistical justification for the use of response mean $\hat{R}(t)$ as an estimate of the response regression $\bar{R}(t)$ by exploiting the underlying monotonic properties of the performance parameter, namely its degradation followed by recovery, as illustrated by TCP throughput measurements. Due to the somewhat technical nature of the derivations, these details are relegated to separate next section.



(a) TCP traces



(b) response means

Figure 11. Comparison of response means of dpctl (configuration A) with 10 and 100 additional flows and original ODL (configuration C) methods using 100 path degradations with $T = 50$ seconds.

B. Finite Sample Statistical Analysis

A generic empirical estimate $\tilde{R}^X(t)$ of $\bar{R}(t)$ based on discrete measurements collected at times $t = j\delta$, $j = 0, 1, ..., T/\delta$, is given by

$$\tilde{R}^X(j\delta) = \frac{1}{n}\sum_{i=1}^{n}\left[g\left(R_i^X(j\delta)\right)\right]$$

for an estimator function g. We consider that the function class $\mathcal{M}$ of $\tilde{R}^X(.)$ consists of unimodal functions, each of which consists of degradation and recovery parts when viewed as a function of time. For ease of notation, we also denote $\tilde{R}^X(.)$ by f in this section such that it is composed of a degradation function f_D and a recovery function f_R as follows:

$$f(R_i(t)) \quad = \quad \begin{cases} f_D(R_i(t)) & \text{if } t \in [0, T_D + T_I] \\ f_R(R_i(t)) & \text{if } t \in (T_D + T_I, T] \end{cases} \tag{1}$$

where $f_D \in \mathcal{M}_D$ and $f_R \in \mathcal{M}_R$ correspond to the leading and trailing edges of the response regression. The *expected error* $I(f)$ of the estimator f is given by

$$\begin{aligned} I(f) &= \int [f(t) - R_i^X(t)]^2 d\mathbf{P}_{R_i^X(t),t} \\ &= \int_{[0,T_D+T_I]} [f_D(t) - R_i^X(t)]^2 d\mathbf{P}_{R_i^X(t),t} \\ &\quad + \int_{(T_D+T_I,T]} [f_D(t) - R_i^X(t)]^2 d\mathbf{P}_{R_i^X(t),t} \\ &= I_D(f_D) + I_R(f_R). \end{aligned}$$

The *best expected estimator* $f^* = (f_D^*, f_R^*) \in \mathcal{M}$ minimizes the expected error $I(.)$, that is

$$I(f^*) = \min_{f \in \mathcal{M}} I(f).$$

The *empirical error* of an estimator f is given by

$$\hat{I}(f) = \frac{\delta}{Tn} \sum_{i=1}^{n} \sum_{j=1}^{T/\delta} \left[f(j\delta) - \left(R_i^X(j\delta) \right) \right]^2.$$

The *best empirical estimator* $\hat{f} = (\hat{f}_D, \hat{f}_R) \in \mathcal{M}$ minimizes the empirical error $\hat{I}(.)$, that is,

$$\hat{I}(\hat{f}) = \min_{f \in \mathcal{M}} \hat{I}(f).$$

Since the response mean $\hat{R}(t)$ is the mean at each observation time $j\delta$, it achieves zero mean error, which in turn leads to zero empirical error, that is, $\hat{I}\left(\hat{R}\right) = 0$; thus, it is a best empirical estimator. By ignoring the minor variations for the smaller values of n, we assume that $\hat{R}$ is composed of a non-decreasing function $\hat{R}_D$ followed by a non-increasing function $\hat{R}_R$ that correspond to decreasing and increasing parts of the performance parameter (such as TCP throughput), respectively. This assumption is valid for the response means of dpctl and ODL methods in Configurations A and C, respectively, shown in Figure 10. In both cases, the response mean is composed of an increasing part followed by a decreasing part once the small variations in the tail of ODL method are ignored.

We will now show that Vapnik-Chervonenkis theory [19] guarantees that the response mean $\hat{R}(t)$ is a good approximation of the response regression $\bar{R}(t)$, and furthermore its performance improves with more measurements from connection degradation events. Such performance guarantee is a direct consequence of the monotone nature of the underlying f_D and f_R functions. Furthermore, this performance guarantee is distribution-free, that is, independent of the underlying joint distributions of controllers and switches, and is valid under very general conditions [10] on the variations of performance parameter (such as TCP throughput) measurements. We note that the underlying distributions could be quite complicated and generally unknown, since they depend on complex interactions between controller software and switches, which are individually complex.

We now provide a complete proof of the above performance result which was briefly outlined in [1]. Let $\hat{R} = \left(\hat{R}_D, \hat{R}_R\right)$ such that the estimator is decomposed into two monotone parts, namely non-decreasing $\hat{R}_D$ and non-increasing $\hat{R}_R$ such that

$$\hat{I}(\hat{R}) = \hat{I}_D(\hat{R}_D) + \hat{I}_R(\hat{R}_R).$$

We now apply Vapnik-Chervonenkis theory [10] in the following to show that the error of estimator $\hat{R}$, given by $I\left(\hat{R}\right)$, is within ϵ of the optimal error $I(f^*)$ with a probability that improves with the number of observations n. More precisely, we show that the probability

$$\mathbf{P}\left\{ I\left(\hat{R}\right) - I(f^*) > \epsilon \right\}$$

decreases with n independent of other factors related to controller and switch distributions. We will establish this result in the following three basic steps. In the first step, we have the basic inequality

$$\begin{aligned} &\mathbf{P}\left\{ I\left(\hat{R}\right) - I(f^*) > \epsilon \right\} \\ &\quad \leq \quad \mathbf{P}\left\{ I\left(\hat{R}_D\right) - I(f_D^*) > \epsilon/2 \right\} \\ &\qquad + \mathbf{P}\left\{ I\left(\hat{R}_R\right) - I(f_R^*) > \epsilon/2 \right\}, \end{aligned}$$

which follows from the observation that the negation of the condition in either right term implies the negation of the condition in left term. Then, by applying the uniform convergence property of the expected and empirical errors over function classes $\mathcal{M}_D$ and $\mathcal{M}_R$ corresponding to the first and second terms on the right hand side [10], respectively, we obtain

$$\begin{aligned} &\mathbf{P}\left\{ I\left(\hat{R}\right) - I(f^*) > \epsilon \right\} \\ &\quad \leq \quad \mathbf{P}\left\{ \max_{h \in \mathcal{M}_D} \left| I_D\left(h\right) - \hat{I}_D(h) \right| > \epsilon/4 \right\} \\ &\qquad + \mathbf{P}\left\{ \max_{h \in \mathcal{M}_R} \left| I_R\left(h\right) - \hat{I}_R(h) \right| > \epsilon/4 \right\}. \end{aligned}$$

Then, by applying the uniform bound ([20], p. 143) provided by Vapnik-Chervonenkis theory to both right hand side terms, we obtain

$$\begin{aligned} &\mathbf{P}\left\{ I\left(\hat{R}\right) - I(f^*) > \epsilon \right\} \\ &\quad \leq \quad 16\mathcal{N}_\infty\left(\frac{\epsilon}{2\mathbf{B}}, \mathcal{M}_D\right) n e^{-\epsilon^2 n/(8\mathbf{B})^2} \\ &\qquad + 16\mathcal{N}_\infty\left(\frac{\epsilon}{2\mathbf{B}}, \mathcal{M}_R\right) n e^{-\epsilon^2 n/(8\mathbf{B})^2} \end{aligned}$$

where $\mathcal{N}_\infty(\epsilon, \mathcal{A})$ is the ϵ-cover size of function class $\mathcal{A}$ under L_∞ norm. Detailed properties of the ϵ-cover can be found in [20], and for our purpose, we note that the ϵ-cover size is a deterministic quantity that depends entirely on the function class. Consequently, the above probability bounds are distribution-free in that they are valid for any joint distribution of controllers and switches at the sites, since the ϵ-covers here depend entirely on the function classes $\mathcal{M}_D$ and $\mathcal{M}_R$.

Then, the monotonicity of functions in $\mathcal{M}_D$ and $\mathcal{M}_R$ establishes that their total variation is upper bounded by $\mathbf{B}$. This property in turn provides the following upper bound for the ϵ-cover sizes of both function classes [20]: for $\mathcal{A} = \mathcal{M}_D, \mathcal{M}_R$, we have

$$\mathcal{N}_\infty\left(\frac{\epsilon}{2\mathbf{B}},\mathcal{A}\right) < 2\left(\frac{4n}{\epsilon^2}\right)^{(1+2\mathbf{B}/\epsilon)\log_2(2\epsilon/\mathbf{B})}.$$

By using this bound, we obtain

$$\mathbf{P}\left\{I\left(\hat{R}_i\right) - I(f^*) > \epsilon\right\} < 64\left(\frac{4n}{\epsilon^2}\right)^{(1+2\mathbf{B}/\epsilon)\log_2(2\epsilon/\mathbf{B})} ne^{-\epsilon^2 n/(8\mathbf{B})^2}.$$

This bound provides qualitative insights into this approach when "sufficient" number of measurements are available. The exponential term on the right hand side decays faster in n than the growth in other terms, and hence for sufficiently large n it can be made smaller than a given probability α. Thus, the expected error $I(\hat{R})$ of the response mean used in the previous section is within ϵ of the optimal error $I(f^*)$ with a probability that increases with the number of observations, and is independent of the underlying distributions. An indirect evidence of this artifact is noticed in the increased stability of the response mean as we increase the number of connection degradation events from 10 to 100 in Figures 9(a) and 10, respectively.

The above probability estimates are not necessarily very tight in part due to the distribution-free nature of the performance guarantee. Nevertheless, this analysis provides a sound statistical basis for using the response mean $\hat{R}$ as an approximation to the underlying response regression $\bar{R}$ for comparing different controllers and configurations.

VI. Conclusions

We considered scenarios with two sites connected over a dedicated, long-haul connection, which must fail-over to a standby connection upon degradations that affect the host-to-host application performance. Current solutions require significant customizations due to the vendor-specific software of network devices and applications, which often have to be repeated with upgrades and changes. Our objective is to exploit the recent network virtualization technologies to develop faster and more flexible software fail-over solutions. The presence of a single long-haul connection and application-level triggers in these scenarios necessitate a solution that is different from usual single controller methods commonly used in many SDN solutions.

We first presented a light-weight method that utilizes host scripts to monitor the connection rtt and dpctl API to implement the fail-over. We then presented a second method using two OpenDaylight (ODL) controllers and REST interfaces. We performed experiments using a testbed consisting of HP and Cisco switches connected over long-haul connections emulated in hardware by ANUE devices. They show that both methods restore TCP throughput, but their comparison was complicated by the restoration dynamics of TCP throughput which contained significant statistical variations. To account for them, we developed the impulse-response method based on statistical finite-sample theory to estimate the response regressions. It enabled us to compare these methods under different configurations, and conclude that on the average the dpctl method restores TCP throughput several seconds faster than the ODL method.

It would be of future interest to generalize the proposed methods to trigger fail-overs based on parameters of more complex client-server applications that utilize TCP for reliable delivery. The performance analysis of such methods will likely be much more complicated since the application dynamics may be modulated by the already complicated TCP recovery dynamics. Our test results are based on using CUBIC congestion control mechanism [18] which is the default on Linux systems, and it would of future interest to test the performance of other congestion control mechanisms, including high-performance versions for long-haul [21] and satellite [22] connections.

Currently there seems to be an explosive growth in the variety of SDN controllers including open source products, such as Helium, Lithium and Beryllium versions of ODL, Floodlight [23], ONOS [24] Ryu [25] and others, and also vendor specific products and adaptations. Furthermore, there is a wide variety of implementations of OpenFlow standards by switch vendors, ranging from building additional software layers on existing products to developing completely native implementations. It would be of future interest to develop general performance analysis methods that enable us to compare various SDN solutions (that comprise of controllers, switches, docker containers and application modules) for more complex scenarios such as data centers and cloud services distributed across wide-area networks. In particular, it would be of interest to develop methods that directly estimate the performance differences between different configurations from measurements using methods such as the differential regression method [26].

It would be of future interest to generalize the approach of this paper to develop a baseline test harness wherein a controller or a switch can be plugged into a known, fixed configuration. The general approach is to develop canonical configurations each with fixed components of the harness, such as application trigger modules, physical network connections and others. Then, impulses responses of different controllers, switches or other SDN components can be generated in these configurations, and they can be objectively compared to assess their relative performance.

Acknowledgments

This work is funded by the High-Performance Networking Program and the Applied Mathematics Program, Office of Advanced Computing Research, U.S. Department of Energy, and by Extreme Scale Systems Center, sponsored by U. S. Department of Defense, and performed at Oak Ridge National Laboratory managed by UT-Battelle, LLC for U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] N. S. V. Rao, "Performance comparison of SDN solutions for switching dedicated long-haul connections," in Proceedings of International Symposium on Advances in Software Defined Networking and Network Functions Virtualization, 2016.

[2] Y. D. Lin, D. Pitt, D. Hausheer, E. Johnson, and Y. B. Lin, "Software-defined networking: Standardization for cloud computing's second wave," IEEE Computer: Guest Editorial, November 2014, pp. 19–21.

[3] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," Proceedings of the IEEE, vol. 103, no. 1, 2015, pp. 14–76.

[4] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," IEEE Communication Surveys & Tutorials, vol. 17, no. 1, First Quarter 2015, pp. 27–51.

[5] T. D. Nadeau and K. Gray, Software Defined Networks. OReilly publishers, 2013.

[6] “Opendaylight.” [Online]. Available: www.opendaylight.org

[7] Y. Srikant, The Mathematics of Internet Congestion Control. Birkhauser, Boston, 2004.

[8] W. R. Stevens, TCP/IP Illustrated. Addison Wesley, 1994, volumes 1-3.

[9] N. S. V. Rao, J. Gao, and L. O. Chua, “On dynamics of transport protocols in wide-area internet connections,” in Complex Dynamics in Communication Networks, L. Kocarev and G. Vattay, Eds. Springer-Verlag Publishers, 2005.

[10] V. N. Vapnik, Statistical Learning Theory. New York: John-Wiley and Sons, 1998.

[11] N. S. V. Rao, W. R. Wing, , S. M. Carter, and Q. Wu, “Ultrascience net: Network testbed for large-scale science applications,” IEEE Communications Magazine, vol. 43, no. 11, 2005, pp. s12–s17.

[12] J. Tourrilhes, P. Sharma, S. Banerjee, and J. Pettit, “SDN and OpenFlow evoltion: A standards perspecitve,” IEEE Computer, November 2014, pp. 22–29.

[13] C. E. Rothenberg, R. Chua, J. Bailey, M. Winter, C. N. A. Correa, S. C. de Lucena, M. R. Salvador, and T. D. Nadeau, “When open source meets network control planes,” IEEE Computer, November 2014, pp. 46–53.

[14] N. S. V. Rao, S. E. Hicks, S. W. Poole, and P. Newman, “Testbed and experiments for high-performance networking,” in Tridentcom: International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, 2010.

[15] “Open networks foundation.” [Online]. Available: www.opennetworking.org

[16] “Software defined networks,” cisco Systems. [Online]. Available: www.cisco.com

[17] “Software defined networking,” hP. [Online]. Available: www.hp.com

[18] I. Rhee and L. Xu, “Cubic: A new tcp-friendly high-speed tcp variant,” in Proceedings of the Third International Workshop on Protocols for Fast Long-Distance Networks, 2005.

[19] V. N. Vapnik, The Nature of Statistical Learning Theory. New York: Springer-Verlag, 1995.

[20] M. Anthony and P. L. Bartlett, Neural Network Learning: Theoretical Foundations. Cambridge University Press, 1999.

[21] T. Yee, D. Leith, and R. Shorten, “Experimental evaluation of high-speed congestion control protocols,” Transactions on Networking, vol. 15, no. 5, 2007, pp. 1109–1122.

[22] J. Cloud, D. Leith, and M. Medard, “Network coded tcp (ctcp) performance over satellite networks,” in SPACOMM 2014, 2014.

[23] “Project floodlight.” [Online]. Available: www.projectfloodlight.org

[24] “Onos overview.” [Online]. Available: www.onosproject.org

[25] “Build SDN agilely.” [Online]. Available: osrg.github.io/ryu/index.html

[26] B. W. Settlemyer, N. S. V. Rao, S. W. Poole, S. W. Hodson, S. E. Hicks, and P. M. Newman, “Experimental analysis of 10gbps transfers over physical and emulated dedicated connections,” in International Conference on Computing, Networking and Communications, 2012.