Adaptive Consistency for Distributed SDN Controllers

Mohamed Aslan
Department of Systems and Computer Engineering
Carleton University
Ottawa, ON K1S 5B6, Canada.
Email: maslan@sce.carleton.ca

Ashraf Matrawy
School of Information Technology
Carleton University
Ottawa, ON K1S 5B6, Canada.
Email: ashraf.matrawy@carleton.ca

Abstract—**In this paper, we introduce the use of adaptive controllers into software-defined networking (SDN) and propose the use of adaptive consistency models in the context of distributed SDN controllers. These adaptive controllers can tune their own configurations in real-time in order to enhance the performance of the applications running on top of them. We expect that the use of such controllers could alleviate some of the emerging challenges in SDN that could have an impact on the performance, security, or scalability of the network. Further, we propose extending the SDN controller architecture to support adaptive consistency based on tunable consistency models. Finally, we compare the performance of a proof-of-concept distributed load-balancing application when it runs on-top of: (1) an adaptive and (2) a non-adaptive controller. Our results indicate that adaptive controllers were more resilient to sudden changes in the network conditions than the non-adaptive ones.**

1. Introduction

Software-Defined Networking (SDN) is a promising network architecture that proposes the decoupling of data and control planes. It uses a logically centralized controller powered by a global view to orchestrate the network, enabling innovation by shifting the task of network administration to network programming.

According to the SDN architecture [1] that is defind by the Open Networking Foundation (ONF), a controller needs to be *logically* centralized. The use of a physically centralized controller can limit the scalability and reliability of large scale networks. A single controller can represent a bottleneck as well as a single point of failure. Recent research in SDN [2], [3], [4], [5], [6] recommends the use of logically centralized but physically distributed controllers. On contrary to centralized controllers, distributed controllers can scale-out by installing new controllers when needed and can be fault-tolerant in case of controller failure.

The design of SDN applications that run on top of distributed controllers is a non-trivial task due to the complexity of handling controllers state synchronization which in-turn can affect the application's performance. Levin et al. [4] studied the impact of the inconsistent controllers on the SDN application performance, they found that inconsistency can significantly degrade the performance of SDN applications. In previous work [7], we confirmed their findings using an implementation as well as a mathematical model of a distributed SDN load-balancer using a different application performance indicator, and this motivated the work in this paper. We also showed that the network state collection mechanism employed by the controller and the type of traffic had an impact on the load-balancer's performance.

Brewer theorem [8], [9] (also known as CAP theorem) stated that it is impossible for a distributed system to simultaneously provide the following guarantees: *Consistency*, *Availability*, and *Partition tolerance*, and that there is always a trade-off between the system's consistency and availability in the presence of network partitions. For example, in the case of a datastore cluster that is comprised of a set of distributed nodes. At one point of time, those nodes got partitioned due to a network failure, while new data update requests continued to arrive at some of the nodes. If those nodes continued to handle the requests, the stored data might become inconsistent but the cluster will still be available. Otherwise the data would remain consistent but it would be said that the cluster is unavailable.

As the CAP theorem applies to any distributed system, we believe that in SDN that would imply that designing an SDN application that runs on top of physically distributed controllers encounters a trade-off between consistency and availability, in case of network partitions.

Panda and et al. [10], investigated how these trade-offs apply to software-defined networks. They concluded that availability and partition tolerance are identical in networks and datastores, however the notion of consistency may differ. In datastore systems, the consistency of data across replicas is the primary concern, but in SDN it is the consistent application of policies across the network.

In this paper, we make the following contributions: (1) we propose the use of adaptive controllers based on tunable consistency in distributed SDNs as we expect that the use of such model could be a solution for some of the apparent problems in SDN, (2) we show how the typical

Authors' copy for personal use. Version: July 3rd, 2016. The final version of this paper will appear in the proceedings of the 17th International Network Strategy and Planning Symposium (Networks 2016). *©2016 IEEE.*

SDN controller architecture can be extended in order to support adaptive consistency, and (3) we implement a proof-of-concept distributed load-balancing SDN application and compare its performance when run on-top of: (1) non-adaptive controllers and (2) adaptive controllers.

The rest of this paper is organized as follows: we present the related work in distributed SDN controllers in Section 2. In Section 3 we provide a background on the consistency models, the problem of inconsistency, and its impact on SDN applications. In Section 4 we explain adaptive consistency and discuss its need in SDN. An extended architecture for adaptively consistent controllers is presented in Section 5. The proof-of-concept distributed load-balancing application is demonstrated in Section 6. While in Section 7 we discuss tunable consistency. Finally Section 8 will be our conclusion and an outline for possible foreseeable work.

2. Related Work: Distributed Controllers

FlowVisor [11] is an OpenFlow proxy which acts as a network virtualization layer that lies between the network devices and control applications. It can host multiple *guest* OpenFlow controllers, one controller per slice while ensuring that slices are isolated from one another.

HyperFlow [3] is a distributed event-based controller that uses a publish/subscribe messaging paradigm built on-top of a distributed file system, giving the applications control over consistency and durability. HyperFlow was designed to be resilient in the presence of network partitioning, where partitions must continue their operation independently, being eventually consistent and favoring availability.

Onix [5] is a general API for controller implementation, allowing the control applications to make their own trade-offs among consistency, and scalability. The authors observed that applications often have different requirements for the consistency of the network state they manage. Onix provides the control applications with two storage subsystems employing different consistency models: a transactional Database (DB) with strong consistency and a memory-based single-hop Distributed Hash Table (DHT) with eventual consistency.

Open Network Operating System (ONOS) [6] is distributed NOS platform for SDNs. ONOS runs instances of Floodlight [12] controllers, and makes use of a distributed DB in order to provide a global view for the controllers. Two prototypes were developed: (1) The first prototype was concerned with implementing the global network view on a distributed platform providing scalability and fault-tolerance. Cassandra [13], [14] an eventually consistent DB was selected as the distributed data store. On top of Cassandra, ONOS deployed a graph DB and provided a graph API for the SDN applications developers. The first prototype was suffering from low performance issues due to the complexity of its data model, the excessive data store operations, and the controllers had to keep polling network state information from the switches. (2) The second prototype was concerned with improving the low performance issues of the first. The authors replaced Cassandra DB with a low latency distributed in-memory key-value data store, and added a caching layer for the network topology information.

DIstributed SDN COntrol plane (DISCO) [15] is a distributed controllers platform that was designed for multi-domain SDN networks, it is built on top of Floodlight [12] SDN controllers, and employs an AMQP-based publish/subscribe messaging module. To support other functionalities such as QoS, DISCO uses agents that can be dynamically be added at the different controllers.

3. Consistency in Software-Defined Networks

3.1. Consistency Models

In distributed systems, the consistency of data among different nodes (different nodes holding copies of same data are known as *replicas*) is governed by the *consistency model* being employed. Tanenbaum and Van Steel [16], presented different consistency models for distributed datastores. They defined the consistency model as a contract between the applications and the datastore, which embodies that if applications agree to obey certain rules, the store promises to work correctly. In the light of their work, the main consistency models of many distributed systems can be categorized into: *strong*, *weak* or *eventual*. In the presence of network partitioning, a strong consistency model would favor consistency to availability, a weak consistency model would favor availability to consistency. While an eventual consistency model, would be in the favor of availability, relax its consistency requirements so that replicas will *eventually* converge to the same state (i.e. become consistent) in case of no further updates were served.

To the best of our knowledge, we believe that most of the research conducted in the area of distributed controllers can be categorized as either strong or eventually consistent controllers.

3.2. Impact of Inconsistency in Networks

As aforesaid, Levin et al. [4] studied the impact of inconsistency among distributed controllers on the SDN application performance, the less consistent the controllers become, the lower the application performs. Levin et al. [4] only studied the impact of inconsistency on application performance. We also cofirmed that the inconsistent state information among the controllers or between the controllers and switches had an impact on the application performance [7]. However, beside application peformace degradation, inconsistency can create other severe problems in the network such as *forwarding loops*, *black holes* and *isoluation and reachability violation*, which we discuss later in this subsection. Guo et al. [17] identified some of these problems.

An example of an application that would employ strong consistency is a security-sensitive firewall application. Such application would probably employ a strong consistency model as it must ensure certain policies are met, and any inconsistency in such policies could led to illegitimate traffic

traversing restricted links. On the other hand, an application that would tolerate using a weaker consistency model is a load-balancing application. Such application could tolerate some inconsistency between the controllers, as long as they agree on the least-loaded server in order to avoid creating forwarding loops which we discuss in details next. In such case, non-strongly consistent load-balancers must develop techniques to ensure that all the controllers agree on the least-loaded server, or to be designed in a way to avoid such conflicts. Indeed this can complicate the design of such SDN application.

There are some reasons where network application developers would choose not to design their applications to be strongly consistent. First, several distributed services require low latency guarantees to properly function. Second, in dynamic (i.e. nodes join and leave) and partitionable networks where node failure is common, oftentimes these services ought to relax their consistency expectancy choosing availability at the expense of strong consistency [18]. Third, many enterprise-grade or carrier-grade applications require high availability guarantees (at least five-nines [19]) while strong consistency comes at the cost of less availability. In addition, applications can have different functional (or non-functional) [1] requirements. Hence, they often opt to use different consistency models (depending on the application requirements).

Further, it is of great importance to highlight the following critical problems that can occur as result of inconsistency between the SDN controllers. It is worth noting that those problems can also be caused by other means e.g. implementation or misconfiguration bugs (even in case of non-distributed controllers). However, in this paper we are only considering the case where those problems were caused as result of controllers' inconsistency.

1) Forwarding loops [17], [21],
2) Black holes [17], [21],
3) and Isolation and Reachability Violation [21], [22].

First, loops can occur when a packet returns to a port it has visited before [22]. Moreover, loops can be finite or infinite (when packets loop indefinitely). Forwarding loops can occur in SDN when two (or more) switches, based on improper flow rules installed by the controllers, repeatedly keep forwarding some flows among themselves in a loop which may never terminates, causing the flows not to reach their intended destinations. The improper rules can be a consequence of an unintended misconfiguration made by the SDN application developer, or due to inconsistent network views at the controllers. For example, in the case of a distributed load-balancer, a controller can decide that a client request (flow) should be forwarded to a least-loaded server (according to its own view), and when the traffic passes through a port that belongs to a switch administered by a different controller (also running the same load-balancing application), then the latter decided to forward the flow back to a different least-loaded server (according to its inconsistent view) that is connected to a switch port that is administered by the first controller. The repetition of this scenario can lead to the flow never reaching any server, causing the client request not to be handled.

Next, a black hole occurs when packets belonging to certain flows (or all packets) get dropped when they traverse a specific switch port. Hence, that switch acts like a black hole in the network. Black holes can occur in SDN for the same reasons as the forwarding loops.

Finally, isolation and reachability policies can be violated as a result of improper flow rules inserted by the controllers due to out-dated information or inconsistent views. Veriflow [21] can check the SDN for violations of policies in real-time by verifying the data-planes (instead of the SDN control applications) for correctness. Further, network alaysis frameworks as [22], [23], [24] can be used to statically check networks for any violations of policies.

4. Proposed Adaptive Consistency

In general, we define an adaptive controller as:

> *"One that can autonomously and dynamically tune its configuration in order to achieve a certain level of performance measured in predefined metrics and based on its requirements."*

In the case where the tunable configuration is the consistency level, we call it an *adaptively consistent* controller. In other words, an adaptively consistent controller is one that can tune its level of consistency in order to reach the desired level of performance based on specific metrics. Adaptive consistency models have previously been proposed for the use in cloud computing environments [25], [26].

There are a number of reasons, which we believe are enough, for justifying the need to adopt the concept of adaptive consistency in SDN controllers. Adaptively consistent controllers can:

1) Reduce the complexity at the applications. Without adaptive consistency, application developers would need to implement application-specific consistency models directly into their applications as every application has different requirements. In turn this could contribute to a lower application implementation cost. Nevertheless, the need for adaptively consistent controllers becomes more apparent in case of deploying multiple applications with different requirements where application developers ought to implement multiple consistency models.
2) React rapidly to the changing network conditions. By tracking the applications' performance in real-time, adatively consistent controllers can tune the consistency level in order to maintain a certain performance level based on pre-defined metrics. In other words, adatively consistent controllers could provide the applications with robustness and reliability against sudden changes in network conditions.

1. *Functional* requirements are the requirements that a system should provide in order to properly function. They show how would the system behave or act for different inputs or situations. *Non-functional* requirements are those constraints on the functions provided by a system. [20]

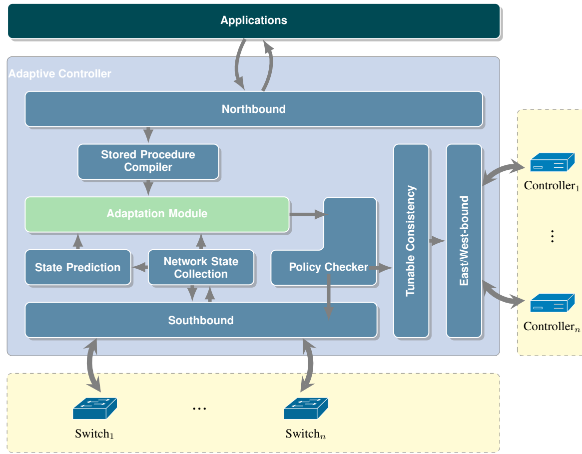

Figure 1: The Extended Architecture for Adaptive Controllers

3) Reduce the overhead of controllers state distribution by eliminating unnecessary state distribution messages without compromising the application performance, especially in case strong consistency is not a requirement, or network states do not have to be replicated to all the controllers.

5. The Design of Adaptive Controllers

In this section, we present an abstract design for adaptive controllers. We extended the typical SDN controller architecture to support controller adaptation. Fig. 1 shows the main constituents of such controllers (which we call modules). These modules are supposed to be physically a part of every controller (i.e. reside at each controller) in a given set of distributed controllers (with the exception of the application module). Based on their functionality, many of these modules are already employed by several controller implementations either explicitly as separate modules or implicitly as parts of other modules. In this section, we show all modules in order to highlight their interoperability to form an adaptively consistent controller. Given certain application specific performance indicators, the adaptively consistent controller will try to autonomously tune, based on *measurements* or *predictions*, the consistency level in order to achieve the lowest possible overhead while maintaining the required level of application performance according to specified performance indicators. The adaptively consistent controller should also abide to certain constraints such as: no forwarding loops or black holes. Generally speaking, we believe that an adaptively consistent model for distributed SDN controllers would turn the problem of inconsistency into an automatic control problem, where the system is comprised of a measurement (and/or prediction) module as well as a control module forming a feedback loop. The control module based on the performance level measured by the measurement module would tune the consistency level of such system.

Southbound Module. The first module is the the southbound module which is responsible for implementing the controller-side of the SDN south-bound protocol. An SDN south-bound protocol enables controllers to communicate with the switches and instruct them how to handle flows. Administered by the ONF, OpenFlow [27] is the defacto standard south-bound protocol.

Northbound Module. The northbound module is responsible for enabling the communication between the SDN applications and the controllers. The SDN applications can run on the same machines as the controllers or on separate machines. Hence, a northbound interface is needed to enable the SDN applications to seamlessly communicate with the controllers via an Application Programming Interface (API) (locally or remotely).

East/West-bound Module. The controllers' east/west-bound module is the module implementing the east-west communication protocol. An east-west protocol is responsible for coordinating controller-to-controller communication.

Network State Collection Module. This module is responsible for collecting network state information about the network so that controllers can build their own views. There are different mechanims that can be used to collect network state information which are mainly dependent on the APIs provided by the Soutbound module. In previous work [7], we studied network state collection and its impact on SDN application performance. The network state information provided by this module can subsequently be used by other modules or it can queried by the applications. The Adaptation module needs the information provided by this module in order to calculate the application performance indicators.

Tunable Consistency Module. A tunable consistency module is one that implements the tunable consistency model and provides a configurable consistency level. In other words, this module encapsulates the complexity of maintaining distributed information across multiple controllers and provides other modules with a uniform interface that can be used to change the consistency of such information in between strong and weak consistency levels.

State Prediction Module. When consulted by the adaptation module, this module is responsible for predicting the behavior of the tunably consistent systems before using some particular configuration. In some cases, predicting the behavior of the application given certain configurations can help the controller to select a suitable configuration for achieving the required application performance.

Stored Procedure Compiler Module. As different SDN applications employ different performance metrics, the application needs to instruct the controller on how to calculate its performance indicator. The application does that by creating a stored procedure (similar to the ones used in Database

systems [28]) that will be executed at the controller. The procedure is executed at the controller to prevent any unnecessary delays as the application might not physically be running on the same machine as the controller. The stored procedure compiler Module is responsible for validating such procedures and compiling them into a representation understood by the Adaptation Module.

Adaptation Module. At the core of an adaptive controller, lies down the adaptation module. The adaptive module is one that is given a current state for the network, calculates application-specific performance indicators and apply an adaptive *strategy* in order to find suitable values for the tunable parameters that would maintain the required level of application performance.

Policy Checker. This module is responsible for checking the controllers' configuration before applying them. It can verify the configurations before being set by the controller against user defined policies. It can also be used to verify any new rules created by the SDN application against the user policies before sending them to the switches via the southbound interface.

Application Module. Finally, the application module holds the application logic of the network application that is implemented by the network developer. This module can physically reside at the controller, or can communicate with the controller through some interface (Northbound). Each application is responsible for choosing its performance indicators and instructing the controller on how to calculate it.

6. Proof-of-Concept

In order to evaluate the validity and effectiveness of the proposed adaptive controller, we implemented a distributed load-balancing application running on-top of adaptive controllers. With regards to the proposed design (Fig. 1), our implementation is missing the "prediction", "stored procedure compiler", and "policy checker" modules. The reason is we wanted to show that even without these modules, the whole system still will perform better that the non-adaptive one in the presence of changing network conditions. For this particular implementation, the mathematical model developed for the load-balancer presented in [7] could be used in the implementation of a prediction module.

6.1. Experimental Setup

Our experimental setup shown in Fig. 2 consists of two domains, each domain includes a controller, a switch, a server, and a set of 32 clients that generate the traffic. The controllers run POX [29] to handle southbound communication. Mininet was used to emulate the network. When a flow arrives at the switch with no rules to match, the switch will notify its controller which in-turn decides where to assign the flow (i.e. which server should handle it). The decision is based on the controller's network view of the least-loaded server. The traffic generated by the clients follows a Poisson distribution with the following parameters: flow arrival rates at the switches are 6 and 4 flows/sec, and packets-per-flow rates for the switches are 34 and 30 pkts/sec. In order to simulate a sudden network change, we change the parameters of traffic shortly after 100 sec to 8 and 2 flows/sec for the flow arrival rates, and to 48 and 16 pkts/sec for the packets-per-flow rates.

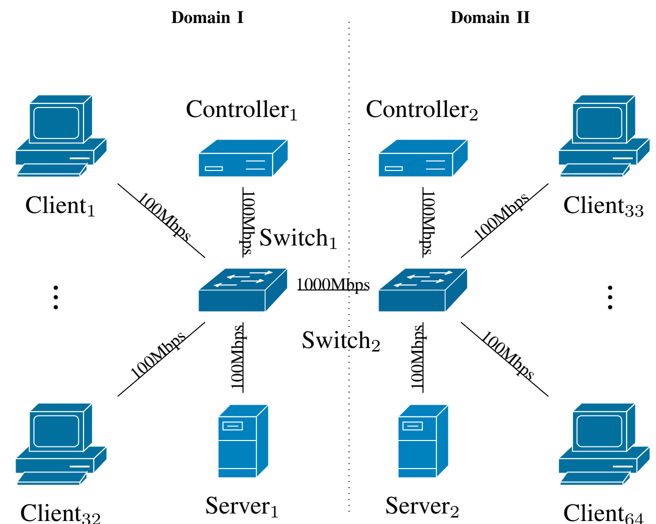

Figure 2: The network topology with two distributed controllers used.

6.2. Implementation

Network State Collection Module. We used a *passive* network state collection mechanism. Where, each controller only maintains the number of flows assigned at its local-domain server (f_i). In previous work [7], we explored different SDN network state collection mechanisms and how they impact the performance of SDN applications. As for the application performance indicator, we opted for the relative difference as defined in [7], the lower the relative difference the better the performance. The module will calculate the relative difference in the number of flows as measured at the controllers (ξ_f) shown in (1) for the use by the adaptation module. For the evaluation, we used the relative difference in the number of bytes as measured at the servers (ξ_b) shown in (2) as a performance indicator. We make the following notations:

- f_i — the flow count assigned to server i ($i \in \{1, 2\}$) at a given time (measured at the controller in our experiment).
- b_i — the amount of traffic received (in bytes, measured at the server) by server i at a given time.
- ξ_f — the relative difference in the number of flows (control parameter).
- ξ_b — the relative difference in the number of bytes (performance indicator).

$$\xi_f = \frac{|f_1 - f_2|}{f_1 + f_2} \quad \cdots \quad s.t. \quad 0 \leq \xi_f \leq 1 \tag{1}$$

$$\xi_b = \frac{|b_1 - b_2|}{b_1 + b_2} \quad \cdots \quad s.t. \quad 0 \leq \xi_b \leq 1 \tag{2}$$

Tunable Consistency Module. The tunable consistency module used in our implementation is based on the *delta* consistency model [30]. The delta consistency model relaxes its data staleness constraints, where a read returns the last value that was updated at most delta time units (also known as the synchronization period) prior that read operation. In other words, the delta consistency model guarantees that all the controllers converge to a point after delta time units, where they all see the same shared data values if no new updates occurred. The tunable consistency module will expose the synchronization period as the tunable parameter (consistency level indicator) to the adaptation module. The adaptation module will automatically tune the value of the synchronization period based on the measured performance of the load-balancer. Using small values for delta might not be feasible, as small values would mean more synchronization messages between the controllers which subsequently affects the application performance. Also using large values for delta could have a negative impact of the application performance.

Adaptation Module. We used a simple controller adaptation module. The core of the module is the adaptation function shown in (3). The adaptation function accepts the ξ_f as an input and returns a value for the synchronization period s from the set $\mathcal{S}$ of allowed periods. In our implementation, we used $\mathcal{S} = \{1, 2, 4, 8, 16, 32\}$. The values of $\mathcal{S}$ were chosen to be powers of 2 (2^{α}) in order to quickly impact the application performance. We make the following notations:

- ξ_f^T — the relative difference threshold (set by the application).
- ϵ^+ and ϵ^- — the threshold margins.
- $s \in \mathcal{S}$ — the synchronization period ($2^{\alpha_{min}} \leq s \leq 2^{\alpha_{max}}$).
- $f(\xi_f)$ — the controller adaptation function which selects a new value for s.

$$f(\xi_f) = \begin{cases} 2^{min(log_2(S)+1, \alpha_{max})}, & 0 \leq \xi_f < \xi_f^T - \epsilon^- \\ s, & \xi_f^T - \epsilon^- \leq \xi_f \leq \xi_f^T + \epsilon^+ \\ 2^{max(log_2(S)-1, \alpha_{min})}, & \xi_f^T + \epsilon^+ < \xi_f \leq 1 \end{cases} \tag{3}$$

Fig. 3 shows how the adaptation function selects the next value of the synchronization period. The function divides the value of the relative difference ξ_f (recall that $0 \leq \xi_f \leq 1$) into three regions. The first region $0 \leq \xi_f < \xi_f^T - \epsilon^-$ where the value of ξ_f is considered low relative to the threshold (better performance) and the synchronization period must be increased in order to decrease the number of unnecessary synchronization messages between the controllers. The second region $\xi_f^T - \epsilon^- \leq \xi_f \leq \xi_f^T + \epsilon^+$ where the value of ξ_f is considered moderate and synchronization period is left intact

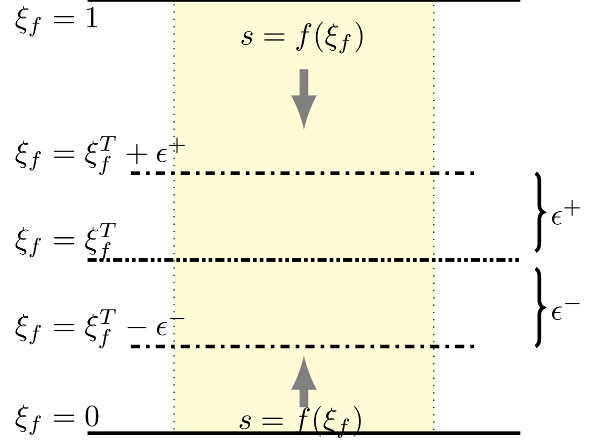

Figure 3: The Controller Adaptation Function

(i.e. unchanged). The third region $\xi_f^T + \epsilon^+ < \xi_f \leq 1$ where the value of ξ_f is considered high (worse performance) and the synchronization period must be decreased in order to enhance the application performance. In our implementation, the adaptation function was invoked every 2 secs.

6.3. Results

We compare the results of a load-balancing application running on-top of non-adaptive distributed controllers against the same application running on-top of adaptive ones. The results were obtained by running the load-balancing application using the setup shown in subsection 6.1 and might differ in other scenarios. We set the synchronization period of the non-adaptive controllers to a constant of 4 sec. As for the adaptive controllers we used the following values for the parameters: $\xi_f^T = 0.25$, $\epsilon^+ = 0.2$, and $\epsilon^- = 0.1$.

Fig. 4 shows how the application performance represented by the relative difference in the number of bytes as measured at the servers ξ_b changes over time in both cases. Fig. 4a shows that in the case of non-adaptive controllers, the application performance was affected by the changing network condition, shortly after time = 100 sec when the traffic parameters changed, the application performance degraded (higher ξ_b). While in the case of adaptive controllers shown in Fig. 4b, the application performance was more resilient (smaller variation in ξ_b) to the sudden change in the network. This behavior can clearly be observed in Fig. 5 which shows the simple moving average (over 10 points) for the relative difference (ξ_b) in both cases. Fig. 6 demonstrates how the adaptive controllers altered the values of their synchronization period over time.

7. Discussion on Tunable Consistency

As different SDN applications may have different consistency requirements [5], they tend to employ different consistency models. One consistency model that we believe

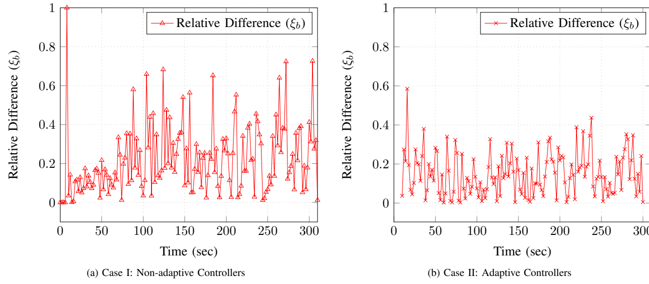

Figure 4: Relative Difference ξ_b vs Time

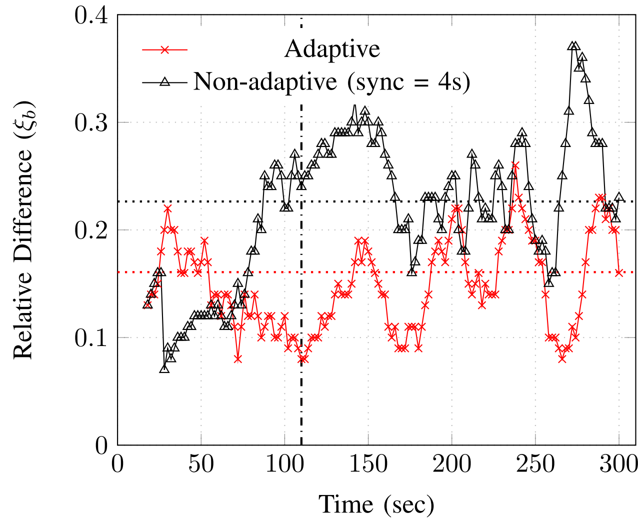

Figure 5: Adaptive vs Non-adaptive Controllers

is generic and could fulfill the requirements of different network applications is the tunable consistency model used in some widely deployed distributed DB such as Amazon DynamoDB [31] and Apache Cassandra [13], [14]. Dixon [32] explained how this model could be used for distributed SDN controllers. For the rest of this section, we refer to such model simply as the "tunable" consistency model. Cassandra is a single-hop (peers have information on all other peers) peer-to-peer (P2P) distributed NoSQL DB, built on top of a distributed hash table (DHT) with a look-up complexity of $\mathcal{O}(1)$. The consistency model underlying Cassandra sometimes is said to be eventually consistent [33]. However, the paramters of the consistency level used in Cassandra can be tuned [33]. An application can *tune* some parameters within each query specifying if it needs that query to be executed strongly or eventually consistent (also known as the *consistency level*) [33], [34]. We believe that adopting a similar model for distributed SDN controllers can be beneficial, where each SDN application can use a consistency level that fulfills its requirements. Also, each query can be accompanied with a different consistency level. Thus by using tunable consistency, even within the same application, developers can have fine-grained control over the consistency of each query.

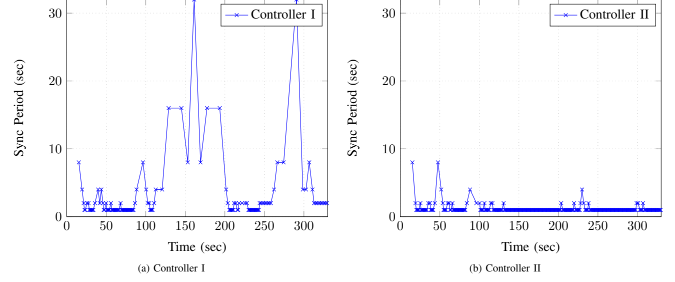

Figure 6: Synchronization Period vs Time

The parameters that can be tuned by the application are: First, N is the total number of replicas (or nodes) in a cluster. Next, R is the number of replicas that must acknowledge a read operation in order to be successful. In other words, how many nodes that the system must wait for to acknowledge a read operation in order to consider it a success and to return the value to the application. Finally, W is the number of replicas that must acknowledge a write operation in order to be successful.

In case a strong consistency level is desired, the following equality [35] needs to be true:

$$R + W > N \tag{4}$$

When retrieving a key from the DB, the equality shown above will guarantee the *overlap* property that there exist at least one node among the R nodes that has the most recent written value. Vogels [34] explained how the aforementioned parameters could be tuned in order to control the consistency level of a distributed DB system.

8. Conclusion and Future Work

In this paper, we investigated the potential of employing adaptive SDN controllers. We believe that the use of such controllers should reduce the application complexity, provide the applications with robustness against sudden changes in network conditions, and reduce the controllers state distribution message when not needed. Additionally, we proposed an extension to the typical SDN controllers in order to support adaptive consistency, and explained the functionality of its constituents. Finally, we implemented a proof-of-concept distributed load-balancing application that runs on-top of both an adaptive and a non-adaptive controller. We compared the application performance in both cases (adaptive vs non-adaptive), and our results showed that adaptive controllers were more resilient to sudden changes in the network conditions than the non-adaptive ones.

For the foreseeable future, we plan to evaluate the feasibility of adaptive controllers employing a more realistic tunable consistency model (see Section 7) similar to that of distributed datastores such as Amazon DynamoDB and Apache Cassandra.

Acknowledgments

The second author acknowledges support from the Natural Sciences and Engineering Research Council of Canada (NSERC) through the NSERC Discovery Grant program.

References

[1] The Open Networking Foundation. (2015) Software-defined networking (sdn) definition. [Online]. Available: http://www.opennetworking.org/sdn-resources/sdn-definition/

[2] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 7–12.

[3] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. USENIX Association, 2010, pp. 3–3.

[4] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in software defined networks," in *Proc. of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 1–6.

[5] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *OSDI*, vol. 10, 2010, pp. 1–6.

[6] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "Onos: towards an open, distributed sdn os," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 1–6.

[7] M. Aslan and A. Matrawy, "On the impact of network state collection on the performance of sdn applications," *IEEE Communications Letters*, vol. 20, no. 1, pp. 5–8, 2015.

[8] E. Brewer, "Towards robust distributed systems," in *PODC*, 2000, p. 7.

[9] ——, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[10] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "Cap for networks," in *Proc. of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 91–96.

[11] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," 2009.

[12] (2015) Project Floodlight. [Online]. Available: http://www.projectfloodlight.org/

[13] A. Lakshman and P. Malik, "Cassandra: structured storage system on a p2p network," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009, pp. 5–5.

[14] ——, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[15] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain sdn controllers," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–4.

[16] A. Tanenbaum and M. Van Steen, *Distributed systems*. Pearson Prentice Hall, 2007.

[17] Z. Guo, M. Su, Y. Xu, Z. Duan, L. Wang, S. Hui, and H. J. Chao, "Improving the performance of load balancing in software-defined networks through load variance-based synchronization," *Computer Networks*, vol. 68, no. 0, pp. 95 – 109, 2014, communications and Networking in the Cloud.

[18] P. Bailis and A. Ghodsi, "Eventual consistency today: limitations, extensions, and beyond," *Communications of the ACM*, vol. 56, no. 5, pp. 55–63, 2013.

[19] F. Wang, H. Wang, B. Lei, and W. Ma, "A research on high-performance sdn controller," in *Cloud Computing and Big Data (CCBD), 2014 International Conference on*. IEEE, 2014, pp. 168–174.

[20] I. Sommerville, *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[21] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[22] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks." in *NSDI*, 2012, pp. 113–126.

[23] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis." in *NSDI*, 2013, pp. 99–111.

[24] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 113–126, 2014.

[25] X. Wang, S. Yang, S. Wang, X. Niu, and J. Xu, "An application-based adaptive replica consistency for cloud storage," in *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*. IEEE, 2010, pp. 13–17.

[26] H.-E. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Perez, "Harmony: Towards automated self-adaptive consistency in cloud storage," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE, 2012, pp. 293–301.

[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[28] "Information technology – Database languages – SQL – Part 4: Persistent Stored Modules (SQL/PSM)," International Organization for Standardization, Geneva, CH, Standard, 2011.

[29] J. Mccauley. (2014) Pox: A python-based openflow controller. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[30] A. Singla, U. Ramachandran, and J. Hodgins, "Temporal notions of synchronization and consistency in beehive," in *Proc. of the ninth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1997, pp. 211–220.

[31] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 729–730.

[32] C. Dixon, "Consistency trade-offs for sdn controllers," *OpenDaylight Summit*, 2014. [Online]. Available: http://colindixon.com/wp-content/uploads/2014/05/sdn-consistency-ods2014.pdf

[33] (2013) Apache Cassandra: configuring data consistency. [Online]. Available: http://www.datastax.com/documentation/cassandra/1.2/cassandra/dml/dml_config_consistency_c.html

[34] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[35] (2015) Apache Cassandra Architecture Overview. [Online]. Available: https://wiki.apache.org/cassandra/ArchitectureOverview