Impact of Adaptive Consistency on Distributed SDN Applications: An Empirical Study

Ermin Sakic, *Student Member, IEEE,* and Wolfgang Kellerer, *Senior Member, IEEE*

***Abstract*—Scalability of the control plane in a Software Defined Network (SDN) is enabled by means of decentralization of the decision-making logic, i.e. by replication of controller functions to physically or virtually dislocated controller replicas. Replication of a centralized controller state also enables the protection against controller failures by means of primary and backup replicas responsible for managing the underlying SDN data plane devices. In this work, we investigate the effect of the the deployed consistency model on *scalability* and *correctness* metrics of the SDN control plane. In particular, we compare the *strong* and *eventual* consistency, and make a case for a novel *adaptive* consistency approach. The existing controller platforms rely on either strong or eventual consistency mechanisms in their state distribution. We show how an adaptive consistency model offers the scalability benefits in terms of the total request-handling throughput and response time, in contrast to the strong consistency model. We also outline how the adaptive consistency approach can provide for correctness semantics, that are unachievable with the eventual consistency paradigm in practice. The adaptability of our approach provides a balanced and tunable trade-off of scalability and correctness for the SDN application implemented on top of the adaptive framework. To validate our assumptions, we evaluate and compare the different approaches in an emulated testbed with an example of a load balancer controller application. The experimental setup comprises up to five extended OpenDaylight controller instances and two network topologies from the area of service provider and data center networks.**

Keywords - consistency models, RAFT, SDN, distributed control plane, scalability, OpenDaylight

I. Introduction

The SDN paradigm aims at centralizing the network logic in a decision-making entity known as the *SDN controller*. The concept of knowledge centralization has recently gained traction for its potential advantages in the abstraction and added simplicity of network control and management operations [1]. The centralization of the controller's knowledge state, however, introduces two new challenges: the *single-point-of-failure (SPOF)* and the *scalability* of the control plane [2]. A number of approaches have been proposed in literature to alleviate the SPOF [3]–[6] issue, with the major approaches relying on direct state- and function-replication across the *replicas* of the SDN controller cluster.

With the concept of state replication, the SDN controller instances replicate their data store contents to other members that take part in a logical *controller cluster*, using a state distribution protocol of choice (e.g. RAFT [7], [8]). When a failure of a controller is suspected, another replica from its *cluster* is able to take over and continue to serve future application's requests. The selection of the *consistency model* leveraged by the replication process affects the incurred synchronization overhead in terms of the resulting packet load, the experienced commit response times and the processing order of commits.

In the *Strong Consistency* model (SC), each consecutive operation that modifies the internal state of the controller is serialized and confirmed by a quorum of replicas, before forwarding the state and processing subsequent transactions. In the leader-based SC approaches (e.g. in RAFT), all requests are serialized by a cluster *leader*, in order to provide for a consistent data store view across all cluster *followers*. Thus, with SC, a large distributed system consisting of multiple controller replicas, is effectively constrained into a monolithic system where each data store modification incurs a minimum of two message rounds and a linear message complexity (in the case of a stable leader) in order to synchronize the controller views [8]–[10].

In the *Eventual Consistency* (EC) model [3], [11], [12], state transitions may be delayed or reordered for an arbitrary period of time. In EC, message updates are advertised in a single round and with linear message complexity. From SDN controller perspective, each controller instance in EC is able to autonomously service the client requests. The updates to the internal data store are thus non-blocking and are executed without incurring an additional delay in SDN application's processing time [9]. However, in the EC the missing constraint of state serialization potentially leads to write conflicts and inefficient decision-making [11].

Recent works have introduced the paradigm of *Adaptive Consistency* (AC) [13], [14]. In general, AC realizes the state synchronization as a non-blocking task. However, after exceeding a configurable number of maximum concurrent per-replica state-modifications, an AC system blocks further updates until all replicas have synchronized to a common state [14]. If the system detects that the *staleness* constraints of an SDN application may be violated by a concurrent state-modification, it blocks the future state modifications until the state consistency across all replicas is reestablished. Additionally, AC autonomously adapts the consistency level metric of the system. This adaptation advocates an asynchronous state synchronization at a dynamically decided frequency across the controller cluster. Hence, the maximum number of allowed concurrently executed per-replica transactions varies based on the current SDN application performance observed during runtime. The adaptation mechanism thus optimizes the trade-off between the correctness and scalability in SDN application's decision-making logic.

Until now, the AC paradigm has lacked an experimental

E. Sakic is with the Department of Electrical and Computer Engineering, University of Technology Munich, Germany; and Siemens AG, Munich, Germany, E-Mail: (ermin.sakic@{tum.de, siemens.com}).

W. Kellerer is with the Department of Electrical and Computer Engineering, University of Technology Munich, Germany, E-Mail: (wolfgang.kellerer@tum.de).

implementation and a proof of its practicability. Furthermore, from the simulation results presented in [14], the overhead of the AC's state-update blocking and state-update distribution during controller operations in the congestion periods lacked a proper analysis. In this paper, we provide the insights into the realization of an AC framework that internalizes the concept presented in [14], and directly compare the developed framework with the SC and EC model realizations w.r.t.: i) the response time; ii) the distribution overhead; and iii) the correctness metrics. Furthermore, we present various means and design paths that can be followed to realize its adaptive component, and compare the different design options. For the comparative study, we leverage the built-in SC APIs exposed by the open-source implementation of RAFT consensus [7] in the OpenDaylight (ODL) controller [15]. We implement our AC/EC framework as an additional registry component in ODL. The framework can thus be deployed as an alternative or an addition to the existing SC framework.

We organize the paper as follows: Sec. II elaborates the system model. In Sec. III we briefly introduce the SC and EC state synchronization models. In Sec. IV, we outline the architecture of our AC framework. Sec. VI motivates the coexistence of different consistency models. Sec. V discusses the building blocks and algorithms of the AC framework. Sec. VII outlines the exemplary load balancer implemented for the purpose of consistency evaluation. It also discusses the system setup and the evaluation methodology. Sec. VIII discusses the results of the qualitative comparison of SC, EC and AC. Sec. IX discusses the related work. Sec. X concludes this paper.

II. System Model

We assume a distributed control plane model, where multiple SDN controller replicas interconnect in order to form a logical *cluster*. Each replica in our system includes a default set of decision-making applications, e.g. those that make resource reservations based on routing or load balancing decisions. The replicated SDN applications and hence the controller replicas expose a set of northbound interfaces (NBIs), allowing for the acceptance of client requests. Depending on the requirements of the data synchronization, the applications hosted in different replicas are also able to process the computations given a client request in either concurrent or serialized manner. Therefore, two possibilities exist: i) processing a client request is possible only on a single replica at a time; ii) processing a client request is possible in concurrent manner in any available replica. The first method guarantees for the true serialization of resource reservations, as the decision-making is coupled with the resource state reservations. The latter method instead relies on the isolated state synchronization and convergence of resource reservation updates after the potentially concurrent decisions were made in isolated replicas. Obviously, in the first scenario, a serialized decision-making may lead to processing bottlenecks in a highly-loaded system [9]. In contrast, we consider a more scalable method, allowing for each reservation-state-update to initiate concurrently and asynchronously at an arbitrary controller replica. It is left up to the deployed consistency model and the state-distribution mechanism to decide the actual ordering of updates.

We consider a *healthy* replica an active, non-corrupt (non-buggy) controller replica that, given the latest up-to-date state, will make *correct* decisions in-line with the expected SDN application design. In contrast, a *failed* replica is an inactive (downed) replica that is both: a) unable to acknowledge the acceptance of state update commits distributed by remote replicas; b) unable to service new application requests. While tolerated by design, we do not evaluate partial controller failures (i.e. resulting in incorrect decision-making or faulty data store updates) but decide to focus on correctness disadvantages stemming from desychronization of controller instances. Our design does not consider Byzantine faults.

SDN controller replicas exchange their applied state-updates for the purpose of achieving high availability of the control plane. The duration of the state convergence is governed by the selection of the state synchronization protocol and the corresponding consistency model. The formation of the cluster is independent of the controllers' placement, hence spatial optimization for objectives of e.g. minimized control plane response time is orthogonal to the synchronization issue.

To guarantee a successful synchronization of the committed state-updates across all replicas, we assume a system with enabled partial synchrony and an *eventual synchronous* communication model. Thus, to make progress in SC and in AC (during blocking period) systems, a guaranteed eventual delivery of each state-update to all healthy replicas is assumed. Commiting an update in these systems requires confirmations of the majority of cluster replicas [8]. Therefore, the SC system is unable to move forward in the case of the outstanding replica confirmations. We assume a *fair* and *robust* control channel, where given a non-partitioned network, messages sent infinitely often are delivered infinitely often [12].

For replica failures, the eventual propagation of updates to all nodes after a controller failure (i.e. the sender's failure) can be achieved by persisting the updates to an in-memory data store and deploying a replica recovery mechanism (e.g. a watchdog mechanism) that reinitializes the controller [8]). This assumption holds for the *fail-recovery* [16] process abstractions, which we assume in the remainder of this work. The controller instances are allowed to fail and rejoin the controller cluster arbitrarily. Before re-enabling the recovered controller, a mechanism for synchronization of missing state updates from healthy replicas (e.g. using anti-entropy or pulling of log snapshots) [12]) must have completed successfully.

Fig. 1 presents our envisioned controller design. It depicts a number of controller replicas, interconnected for the purpose of achieving high-availability of the controller-switch and controller-client connections. Each controller executes a number of SDN applications (i.e. routing, load-balancing). Fig. 1 depicts the case where each controller instance executes a copy of each application. For the remainder of the paper, we hold to this assumption. Thus, we allow each controller replica to execute an instance of each available SDN application (control functionality) individually. The SDN controller applications base their decisions on the current content of either one or more in-memory data store implementations which leverage different consistency models. The total controller data state in an SC cluster is partitioned into a number of *data-shards*.

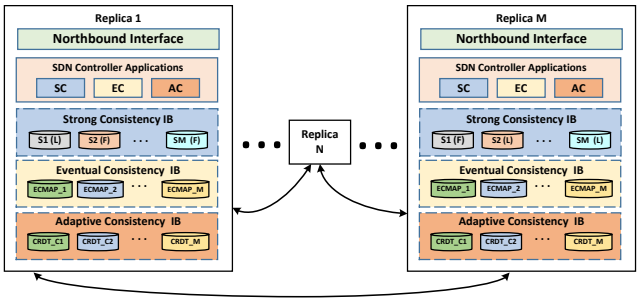


Fig. 1. The internal controller model comprising data stores with varying degrees of state consistency. The controller designer should be allowed to select the appropriate data quality based on their application correctness and throughput requirements. In the case of SC, leaders are elected at a per-*shard* granularity. EC and AC state instances represent exemplary *EC map* realizations for the ONOS [5] EC approach and our AC approach, respectively.

The "sharding" of the full data store state into the data-shards is configurable at an arbitrary granularity. In OpenDaylight's realization of the SC model, an individual instance of RAFT consensus and thus the controller cluster leadership is maintained for each shard. In the remainder of the paper, for the SC model, we assume a single default data-shard replicated across each controller instance, and thus a single instance of RAFT consensus responsible for distribution of state updates. Failures of a shard leader lead to an unavailability of the read and write operations during the re-election period for the particular shard previously under the failed leader's controller. EC maps, on the other hand, are data structures whose synchronization is enforced in the background using a gossiping/broadcast primitive. In ONOS [5], for example, EC maps are replicated to all controller instances that are members of a common cluster. Finally, the replication of the AC data structures presented henceforth is handled on a per data-state instance basis, so to allow for granular guarantees on the minimum synchronization interval of the observed data-state.

III. Strong and Eventual Consistency Models

For completeness, we henceforth give a brief overview of the two common consistency models implemented in the SDN controller platforms ONOS [5] and OpenDaylight [15].

A. Strong Consistency (SC) Model

In leader-based *Strong Consistency* (SC) consensus algorithms (e.g. RAFT [7], Paxos [17]), each replica is assigned either a *follower*, *leader* or a *candidate* role. Whenever a data store update is initiated by a cluster client at any active replica, the receiving controller proxies the received client request to the current cluster *leader*. The leader is the controller instance that orders the incoming state-update requests, so as to allow for a serialized history of updates and thus ensure the operational state consistency during runtime. Following a committed state-update at the leader, the update is propagated to the cluster *followers*. It is committed to their data store only after half of the *followers* have *agreed* on the update. A number of distributed consensus protocols were proposed in the past [7], [17]–[21]. Currently, OpenDaylight [15] and ONOS [5] are the most attractive open-source SDN controller platforms with the largest user-/tester-base. They both provide for white-box testing and insights related to their consensus implementations. As of the time of writing this document, they both implement RAFT as the only consensus protocol. Hence, RAFT was selected as a valid implementation representative for our measurement-based study of the SC model.

In contrast to the correctness benefits of the serialization of state-updates, RAFT possesses the disadvantage of an added overhead in the expected response time and lower availability [8] compared to using the eventual consistency primitives. Namely, a single data store update initiated at a follower replica requires a round trip to the leader for confirmation; as well as reaching consensus among the majority of replicas [9]. This leads to an added blocking period and an overhead in confirmation of transactions. Furthermore, quorum-based consensus algorithms can tolerate a maximum of $F = \lceil C/2 - 1 \rceil$ failures in a cluster of C controllers. This limitation relates to the requirement of ensuring data consistency in the case of network partitions, an invariant feasible only when a majority of nodes are involved in confirming the transactions [22], [23]. In the best case, the cluster operates at the speed of the leader, and in the worst case, at the speed of the slowest follower [8].

B. Eventual Consistency (EC) Model

Eventual consistency (EC) claims that replicas eventually converge to the same final values independent of the applied order of operations, assuming that users (i.e. applications) eventually stop submitting new operations [24]. In EC, all reads and writes are performed locally at the processing speed of the local replica. Hence, applications written on top of the EC primitives proceed their operation without a penalty of confirmation time. The state-updates in EC are propagated in the background. In ONOS, state distribution across the EC replicas occurs using an update-push distribution process and the *anti-entropy*, where replicas continuously compare their local state and eventually converge the deltas. Furthermore, updates to the states may be marked with local timestamps, hence allowing for global ordering of updates. EC favors the performance at the expense of consistency, potentially leading to correctness issues if the applications rely on the *non-staleness* of the local state for their correct operation [11].

IV. Adaptive Consistency (AC) Model

In order to emphasize on the novelties introduced in this work, we now briefly summarize the concept and describe our realization of an *Adaptive Consistency* (AC) framework.

The AC framework allows for the *create*, *remove*, *update*, *delete* operations on the granular state instances (such as counters, registers, maps etc.). The operations are *eventually* synchronized between the controller replicas. However, in contrast in contrast the EC model introduced in Sec. III-B, that allows for enqueueing of an unbounded number of buffered unconfirmed operations, the maximum number of enqueued manipulations in an AC framework is limited by the size of an *update distribution queue* and a *timeout-based automated*

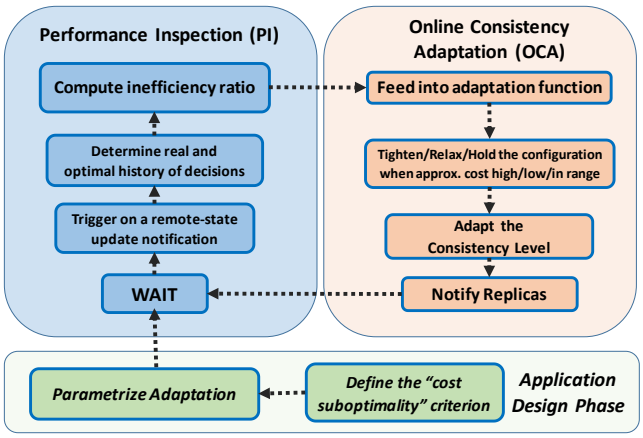


Fig. 2. High-level architecture of the AC framework. The PI block is responsible for the inspection of the negative effect of the state synchronization on the quality of decision-making and the generation of a corresponding inefficiency ratio. The OCA block takes this variable as an input for the adaptation of the consistency level currently applied for the monitored state.

distribution of the enqueued updates. The maximum size of the state-update distribution queue and the maximum timeout duration are governed by the currently applied *consistency level* (CL). The maximum distribution queue size and the timeout are maintained at the granularity of an observed data store state. Given an application "inefficiency" metric and the optimization target, the AC framework decides on the optimal CL that is to be applied for upcoming operations.

The high-level process flow of the AC is depicted in Fig. 2. During the *Application Design* phase, the designer writes an SDN application built atop of the AC framework and parametrizes the adaptation functions. During this phase, the developer must present an "inefficiency" metric related to his application logic (e.g. the optimality of routing decisions [14]), as well as to parametrize the adaptation thresholds/efficiency targets. The specification parameters of the adaptation target vary with the choice of the adaptation function. We discuss the *threshold*- and *PID*-based functions in Sec. V-D.

At time t_L, the *Performance Inspection* (PI) block triggers on an eventually delivered state-update U, initiated by a remote SDN controller replica C_R. In the PI block, each controller replica C_L locally decides its own view of the real history of state-updates, by ordering the updates based on the time-stamp of updates. Let t_U denote the time when update U was initiated at C_R. Then, after deciding the global order of updates, each replica C_L evaluates the effect of being *late*-notified of the update U. To do so, it compares two histories: i) The set of results associated with the actual actions taken during the period $[t_U, t_L]$; ii) The set of results associated with the actions that would have been taken during the period $[t_U, t_L]$ *if* the update U had been serialized and known to C_L at t_U. Thus, two sets of results are stored, the set of real (i.e. suboptimal) decisions, and the set of ideal (i.e. optimal) decisions.

An *inefficiency* metric (i.e. an approximation factor), estimates the ratio between the suboptimal and optimal decision, and thus the cost of eventual state-update delivery. The latest measured inefficiency is fed into the *Online Consistency Adaptation* (OCA) block. In order to decide on the most-fitting CL for the observed state, the OCA block considers the latest inefficiency report as well as, optionally, the *history* of previous inefficiency reports. The OCA block then decides upon the new best-fitting CL and disseminates this decision to all cluster replicas. In our AC realization, the overhead of the OCA block is centralized at a single controller that collects the remote replica's inefficiency metrics and decides on the most-fitting CL. PI and OCA blocks are pipelined for a particular state, but are parallelized for updates on different state instances, thus enabling scalable consistency adaptation.

V. Realization of an AC Framework

In this section, we present the mechanisms behind our prototype realization of the AC framework, comprising: i) a CRDT-based in-memory data store; ii) a generalized load balancer SDN controller application (SDN-LB); iii) the corresponding PI block; iv) the OCA block comprising the *threshold*- and *PID*-based adaptation functions; v) the mechanism for cluster-wide data store state-updates synchronization.

A. CRDT model for state-updates

Convergent Replicated Data Types (CRDTs) [25] are a novel approach to handling conflict-free distributed updates on a set of eventually consistent data structures. The useful property of the CRDTs is that the isolated views of a single CRDT at different replicas eventually converge to the same value, independent of the order of updates. Thus, CRDTs preserve the correctness invariant, even in the case of an increased network latency and packet loss. With CRDTs, updates monotonically advance according to a partial order, subsequently converging towards the least upper bound of the most recent value. An example of a replicated counter datatype is a *PN-Counter* (Increment/Decrement Counter), whose increment and decrement manipulations commute. Our take on a PN-Counter realization is presented in Alg. 1. We have also leveraged CRDT register and set structures in our framework. However, for brevity we present here only the PN-Counter, and refer to [25] for an overview of other data-types.

In our model, individual state-updates are synchronized across the SDN controller replicas, and are stored in a log-tree, together with their initiation time-stamp, for the purpose of later reference during the steps taken in the PI block. The accepted updates to a CRDT-modeled state are synchronized across the controller replicas, while rejections result in data store update failures and a subsequent notification to the requesting application. The admission control for new updates is based on the properties of the queue distribution (i.e. the maximum queue size), governed by the currently applied CL associated with the target CRDT (ref. Section V-E).

Alg. 1 presents our PN-Counter realization. Upon a new client request to modify a particular data store object (realized as a counter CRDT instance), the local controller executes the admission control (Lines 3-9). If the update is accepted, the controller MERGEs the update with its local CRDT (Lines 13-17). It then enqueues the update for a cluster-wide distribution (ref. Section V-E). On receiving the update initiated by a

remote controller, the local controller executes the MERGE function (Lines 22-28). Each CRDT additionally implements the QUERY function, allowing to read its current state.

Algorithm 1 Distributed CRDT PN-Counter

Notation:
C_R Remote controller replica
C_L Local controller replica
B_j Client requesting a CRDT state-update
Ctr_k PN-Counter targeted for update
$S^{C_L}_{Ctr}$ Set of PN-Counters stored in C_L's AC data store
U_{Ctr_k} Update request for state Ctr_k
$B[B_j, Ctr_k]$ Update-log for client B_j and state Ctr_k

1: **upon event** *client-update*$< B_j, U_{Ctr_k} >$ **do**
2: **if** $Ctr_k \in S^{C_L}_{Ctr}$ **then**
3: $success := evalAddToDistributionQueue(U_{Ctr_k})$
4: **if** *success* == *True* **then**
5: $B[B_j, Ctr_k] \leftarrow B[B_j, Ctr_k] \cup U_{Ctr_k}$
6: $merge(U_{Ctr_k})$
7: $notify(B_j$, *update-success* $< U_{Ctr_k} >)$
8: **else**
9: $notify(B_j$, *update-failed* $< U_{Ctr_k} >)$
10: **else**
11: $notify(B_j$, *update-failed* $< U_{Ctr_k} >)$
12:
13: **function** MERGE(U_{Ctr_k})
14: **if** $U_{Ctr_k}.operation == DECREMENT$ **then**
15: $Decr[Ctr_k] \leftarrow Decr[Ctr_k] \cup U_{Ctr_k}.amount$
16: **else if** $U_{Ctr_k}.operation == INCREMENT$ **then**
17: $Incr[Ctr_k] \leftarrow Incr[Ctr_k] \cup U_{Ctr_k}.amount$
18:
19: **function** QUERY(Ctr_k)
20: *return*$(\sum_j Incr[Ctr_k]_j - \sum_j Decr[Ctr_k]_j)$
21:
22: **upon event** *remote-update*$< C_R, < B_r, U_{Ctr_k} >>$ **do**
23: **if** $Ctr_k \notin S^{C_L}_{Ctr}$ **then**
24: $notify(C_R$, *update-failed* $< U_{Ctr_k} >)$
25: **else if** $Ctr_k \in S^{C_L}_{Ctr}$ **then**
26: $B[B_r, Ctr_k] \leftarrow B[B_r, Ctr_k] \cup U_{Ctr_k}$
27: $merge(U_{Ctr_k})$
28: $notify(C_R$, *update-success* $< U_{Ctr_k} >)$

B. Performance Inspection (PI)

The adaptation of the CLs of a particular state is based on a provided application *inefficiency* metric. We define the inefficiency metric as the approximation ratio between the series of *observed* and *optimal* results of an SDN controller application's decisions. The *optimal* result comprises the decisions the application would have made if each update in the system had been serialized (i.e. consensus-based). The *observed* result is the one the local replica has achieved in an online manner, based on its own local state, and without consideration of the status of other replicas. For a replica to compute the optimal result, the knowledge about the content and timing of the eventually delivered updates must be available. The timing characteristics are necessary for the total ordering of the observed and eventually delivered state-updates.

The generalized calculation of the inefficiency metric is depicted in Alg. 2. In Lines 6-8 the PI block identifies the previously executed operations on the observed state, in the period before the *remote* update $U_{Ctr_k^{remote}}$ initiation at the remote replica C_R. Thus, Line 8 yields an array of consistent entries which correspond to a part of the serialized true history of updates $S_{U_{cnst}}$. Lines 10-12 identify the set of client requests that have resulted in *potentially* suboptimal decisions, made in the past by the local replica C_L, without the consideration of the eventually delivered remote state-update. Lines 14-16 then derive the application-specific optimal decisions, given the identified optimal history $S_{U_{cnst}}$, the serialized remote update $U^{remote}_{Ctr_k}$ and a set of client requests $R_{U_{incnst}}$, previously served in a suboptimal manner. The method `CompInefficiency()` in Line 20 takes as an argument the consistent (optimal) history of decisions $S_{U_{cnst}}$, and the actual, potentially suboptimal, history of decisions $S_{U_{incnst}}$. It then returns the inefficiency (approx. ratio) ϕ given the two series of decisions.

Let σ^R_u and σ^R_o denote the cost of suboptimal and optimal decisions for a request R in general case, respectively. Then, the binary value of X^R_{subopt} denotes an *inefficient* result, induced by the staleness (caused by delayed synchronization):

$$X^R_{subopt} = \begin{cases} 1 & \text{when } \sigma^R_u > \sigma^R_o \\ 0 & \text{when } \sigma^R_u \leq \sigma^R_o \end{cases}$$

`CompInefficiency()` and `AppLogic()` functions are application-specific implementations. In the next section we present an exemplary `CompInefficiency()` realization for a generalized online load balancer [26]. Its `AppLogic()` realization is assumed to optimally assign each incoming client request to the replicated server instances based on its *current local* view of the server resource utilizations. We evaluate the algorithm in implementation in Sec. VII.

C. Computation of the inefficiency metric for a generalized online load balancer SDN application

For a set of defined data store states S and a state-update $U(t-n)$, timestamped at time $t-n$, let $T(t-n)$ be the matrix of observations of the states encompassing the period $[t-n \,..\, t]$. Then, $T(t-n)$ is a matrix of $|S|$ x n elements.

Let $S(i)$ be the ith vector of the observed state values at time $t-n+i$, so that:

$$\mathrm{S}(i) = (s_1(i), s_2(i)..s_m(i)) : S(i) \in T(t-n) \text{ s.t. } N_{res}(i) = |S(i)|$$

First, let $S_{U_{incnst}}$ contain the suboptimal (real) history of state-updates. Let $S_{U_{cnst}}$ accordingly hold the computed ideal (optimal) history of state-updates (computed as per Lines 18-19 of Alg. 2). Then, for each vector (time-point) i of observed state values $S_{U_{incnst}}$ and $S_{U_{cnst}}$ we can compute the costs of optimal and suboptimal decisions at time i, $\sigma^{R_i}_o$ and $\sigma^{R_i}_u$ respectively, using standard deviation metric:

$$\sigma^{R_i}_o = \sqrt{\frac{1}{N_{res}(i)} \sum_{j=1}^{N_{res}(i)} (s_j(i) - \mu_{S^{R_i}_{cnst}})^2} \text{ where } s_j(i) \in S_{U_{cnst}}(i)$$

$$\sigma^{R_i}_u = \sqrt{\frac{1}{N_{res}(i)} \sum_{j=1}^{N_{res}(i)} (s_j(i) - \mu_{S^{R_i}_{incnst}})^2} \text{ where } s_j(i) \in S_{U_{incnst}}(i)$$

where

Algorithm 2 Inefficiency calculation for a distributed CRDT

Input:
C_R Remote controller replica
C_L Local controller replica
$U^{remote}_{Ctr_k}$ Reported remote update request for state Ctr_k
$U^{local}_{Ctr_k}$ Local update request for state Ctr_k
$S_{Ctr^U_k}$ Set of previously logged updates for state Ctr_k
$U(T)$ Timestamp of the state-update U at C_R
$U(R)$ Client request that resulted in the update U

procedure HANDLE NEW COUNTER UPDATE
upon event *update* $< C_R, U_{Ctr_k^{remote}} >$ **do**
  $S_{U_{cnst}} := \emptyset$
  $S_{U_{incnst}} := \emptyset$

  for all $U^{local}_{Ctr_k} \in S_{Ctr^U_k}$ **do**
    if $U^{local}_{Ctr_k}(T) < U^{remote}_{Ctr_k}(T)$ **then**
      $S_{U_{cnst}} \leftarrow S_{U_{cnst}} \cup U^{local}_{Ctr_k}$

  for all $U^{local}_{Ctr_k} \in S_{Ctr^U_k}$ **do**
    if $U^{local}_{Ctr_k}(T) >= U^{remote}_{Ctr_k}(T)$ **then**
      $R_{U_{incnst}} \leftarrow R_{U_{incnst}} \cup U^{local}_{Ctr_k}(R)$

  for all $R_U \in R_{U_{incnst}}$ **do**
    $U^{localOpt}_{Ctr_k} :=$ *AppLogic*$(R_U, S_{U_{cnst}})$
    $S_{U_{cnst}} \leftarrow S_{U_{cnst}} \cup U_{Ctr_k^{localOpt}}$

  $S_{U_{cnst}} \leftarrow S_{U_{cnst}} \cup U^{remote}_{Ctr_k}$
  $S_{U_{incnst}} \leftarrow S_{Ctr^U_k} \cup U^{remote}_{Ctr_k}$
  $\phi =$ *CompInefficiency* $(S_{U_{incnst}}, S_{U_{cnst}})$

  $reportIneff(\phi)$

$$\mu_{S^{R_i}} = \frac{\sum_{j=1}^{N_{res}(i)} s(j)}{N_{res}(i)} \tag{1}$$

represents the mean utilization of resources at time-offset i, and R_i represents the client request at time-offset i.

Finally, after having computed the costs of optimal- and suboptimal decisions, the average inefficiency Φ_T for the observation interval $T(t-n)$ can be computed as:

$$\Phi_T = \frac{\sum_{i=0}^{\|T\|} \sigma_u^{R_i}}{\sum_{i=0}^{\|T\|} \sigma_o^{R_i}}$$

D. Online Consistency Adaptation (OCA)

The OCA block is responsible for the collection of computed inefficiency values and their online evaluation. The output of the OCA block is the adapted CL for the observed state instance. The computed inefficiency value ϕ is input into the OCA block and the adaptation function `reportIneff()` is called, as depicted in Fig. 2 and Alg. 2, respectively.

We present two methodologies for adapting the applied CL, given a historical set of inefficiency reports:

1) Threshold-based CL adaptation: If the observed mean inefficiency over a window of inefficiency observations of size W is below, above or in between the lower and upper thresholds, the adaptation function decides to raise, lower or keep the currently applied CL, respectively. Threshold-based CL adaptation is specified in Alg. 3.

2) PID-based CL adaptation: In addition to the *integral* part, the PID-based feedback compensator also considers *proportional* and *differential* parts of the recent inefficiency reports. Each part can be assigned a corresponding weight, thus allowing to favor either a fast adaptation response or long-term adaptation accuracy. For the PID-based adaptation we additionally configure the single target value the function aims to achieve at runtime. Alg. 4 describes the procedure.

Algorithm 3 Threshold-based Consistency Level Adaptation

Input:
CL_{Ctr_k} Currently applied CL for counter Ctr_k
$S^{Ctr_k}_\phi$ Set of previously stored inefficiency reports
T_U Upper adaptation trigger for the threshold metric
T_L Lower adaptation trigger for the threshold metric
W Window size of considered inefficiency observations

procedure HANDLE NEW INEFFICIENCY REPORT
  function REPORTINEFF(ϕ)
    $S^{Ctr_k}_\phi \leftarrow S^{Ctr_k}_\phi \cup \phi$
    $S_{Rlv} := S^{Ctr_k}_\phi[|S^{Ctr_k}_\phi| - W : |S^{Ctr_k}_\phi|]$
    $\mu_{S_{Rlv}} := \frac{\sum_{i=0}^{|S_{Rlv}|} S_{Rlv}(i)}{|S_{Rlv}|}$

    if $\mu_{S_{Rlv}} \geq T_U$ **then**
      raiseCL (CL_{Ctr_k})
    else if $\mu_{S_{Rlv}} \leq T_L$ **then**
      lowerCL (CL_{Ctr_k})

Algorithm 4 PID-based Consistency Level Adaptation

Input:
CL_{Ctr_k} Currently applied CL for counter Ctr_k
$S^{Ctr_k}_\phi$ Set of previously stored inefficiency reports
T_O Target oscillation value
I_g, P_g, D_g Integral, proportional and differential gains
W Window size of considered inefficiency observations

procedure HANDLE NEW INEFFICIENCY REPORT
  function REPORTINEFF(ϕ)
    $S^{Ctr_k}_\phi \leftarrow S^{Ctr_k}_\phi \cup \phi$
    $P_{term} := P_g * (T_O - S^{Ctr_k}_\phi[|S^{Ctr_k}_\phi|])$
    $I_{term} := I_g * \sum_{i=|S^{Ctr_k}_\phi|-W}^{|S^{Ctr_k}_\phi|} (S^{Ctr_k}_{\phi\ i} - T_O)$
    $D_{term} := D_g * [(S^{Ctr_k}_\phi[|S^{Ctr_k}_\Phi|] - T_O) - (S^{Ctr_k}_\phi[|S^{Ctr_k}_\phi| - 1] - T_O)]$

    $T := P_{term} + I_{term} + D_{term}$
    if $T > T_O$ **then**
      raiseCL (CL_{Ctr_k})
    else if $T < T_O$ **then**
      lowerCL (CL_{Ctr_k})

E. State synchronization strategies

To restrict the staleness, i.e. to limit the amount of unseen updates for a particular state on diverged controller replicas, AC ensures that a reliable distribution of a bounded set of updates has occurred before a new data store transaction

for the target state is allowed in the system. Each time a client requests a new state-update, we evaluate the number of previously submitted unconfirmed state-updates on the local replica. If this number is above the maximum queue size governed by the currently applied CL, the transaction is rejected. Otherwise, the state-update is enqueued in a per-state FIFO queue. Depending on the distribution strategy, we distinguish two abstractions of update-state distribution. These abstractions have their trade-offs in terms of response time and the generated update distribution load in the control plane, but they both ensure the property of limited staleness by bounding the maximum number of enqueued isolated updates per replica:

1) Fast-Mode State Distribution: The first procedure of Alg. 5 realizes this distribution abstraction. If the actual occupancy of the state distribution queue is below the CL-governed threshold, the state-update is *admitted* for processing (Lines 3-6), otherwise it is dropped (Lines 7-8). If admitted, the update is prepared for the distribution to the other members of the cluster. The unconfirmed updates in the system are first enqueued in the distribution queue. Any new update is merged at the tail of the queue (Line 4). The distribution procedure then serializes all outstanding unconfirmed updates and distributes these to the remote replicas (Line 5). The sender replica then waits on the asynchronous confirmations for the individual updates. After all *active* cluster members have acknowledged the state-update(s), the sender removes the *acknowledged* updates from the distribution queue (refer to procedure *"On Acknowledgment of distribution"*).

2) Batched-Mode State Distribution: The second procedure of Alg. 5 realizes this distribution abstraction. The transmission of a series of unconfirmed updates on each new update has the advantage of the lowered response time and reliability in the case where some of the previously sent out packets are lost. Nevertheless, generation of a new frame for each new state-update may cause unnecessary load if the response time is not the optimization criterion. For such scenarios, we have realized a batching queue that collects a number of state-updates (Line 4), up to the maximum amount defined by the applied CL for the particular state, and distributes these in a batch to the peer replicas (Line 7). For infrequently updated state-instances, we introduce an asynchronous timer that triggers the state-update distribution whenever a non-empty queue is not distributed for the duration of time specified by the applied CL (Lines 14-17).

Algorithm 5 Fast and batched distribution of state-updates

Input:
$U^{local}_{Ctr_k}$ Local update request for state Ctr_k
$CL^{Ctr_k}_{QS}$ Max. distribution queue size for the applied CL
$CL^{Ctr_k}_{TO}$ Distribution timeout for the applied CL
$Q^{E}_{Ctr_k}$ Distribution queue for the unacknowledged state-updates committed at the local replica
QC_{Ctr_k} List of local state-updates acknowledged by all remote replicas
C The set of remote controller replicas

1: **procedure** FAST-MODE DISTRIBUTION
2: **function** EVALADDTODISTRIBUTIONQUEUE($U^{local}_{Ctr_k}$)
3: **if** $occupied(Q^{E}_{Ctr_k}) < CL^{Ctr_k}_{QS}$ **then**
4: $enqueue(Q^{E}_{Ctr_k}, U^{local}_{Ctr_k})$
5: $distribute(C, Q^{E}_{Ctr_k})$
6: *return* True
7: **else if** $occupied(Q^{E}_{Ctr_k}) \geq CL^{Ctr_k}_{QS}$ **then**
8: *return* False
9:

1: **procedure** BATCHED-MODE DISTRIBUTION
2: **function** EVALADDTODISTRIBUTIONQUEUE($U^{local}_{Ctr_k}$)
3: **if** $occupied(Q^{E}_{Ctr_k}) < CL^{Ctr_k}_{QS}$ **then**
4: $enqueue(Q^{E}_{Ctr_k}, U^{local}_{Ctr_k})$
5: **if** $occupied(Q^{E}_{Ctr_k}) == CL^{Ctr_k}_{QS}$ **then**
6: $T_{Ctr_k} == null$
7: $distribute(C, Q^{E}_{Ctr_k})$
8: **if** $T_{Ctr_k} == null$ **then**
9: $T_{Ctr_k} :=$ *init-timer*$(CL^{Ctr_k}_{TO})$
10: *return* True
11: **else if** $occupied(Q^{E}_{Ctr_k}) \geq CL^{Ctr_k}_{QS}$ **then**
12: *return* False
13:
14: **upon event** $expired < T_{Ctr_k} >$ **do**
15: $T_{Ctr_k} == null$
16: **if** $occupied(Q^{E}_{Ctr_k}) > 0$ **then**
17: $distribute(C, Q^{E}_{Ctr_k})$

1: **procedure** ON ACKNOWLEDGMENT OF DISTRIBUTION (FAST- AND BATCHED-MODE)
2: **upon event** $acknowledged < QC_{Ctr_k} >$ **do**
3: **for all** $U^{local}_{Ctr_k} \in QC_{Ctr_k}$ **do**
4: $Q^{E}_{Ctr_k}$.remove($U^{local}_{Ctr_k}$)

VI. COEXISTENCE OF THE CONSISTENCY MODELS

A number of use cases speaks for coupling the SC and AC in a single system, specifically in the case where SC invariants may not be invalidated for only a *subset* of the deployed SDN controller operations. On the other hand, AC may benefit from SC when consensus is useful for a particular non-frequent AC procedure. We henceforth name some use cases:

1) Policy handling with consistency invariants: Sometimes, the properties of the AC model alone are insufficient because of the strict invariant requirements. For instance, handling a routing or security policy in a consistent manner may be required when a possibility of temporary incorrect configuration exists (e.g. black holes and forwarding loops [27], [28]). Similarly, when optimal decision-making based on the current data store state is a requirement, a globally up-to-date view in each replica must be ensured at all times.

2) Exactly-once semantics: Ensuring the exactly-once semantics, when multiple replicas are notified of an external event, requires consensus in order to elect the executing controller instance [6]. Multiple triggers may lead to such events, e.g. switch state-change notifications. AC could process such events in the RSM-manner (as a Replicated State Machine [29]) and subsequently re-configure the switches, so that the result of the computations in the RSM instances are both compared and applied *in* the switch. This, however, induces complexity that requires extended switch functionalities [6], [12], [30]. Optionally, with a hybrid SC/AC deployment, SC mechanisms [7] could provide for the leadership semantics for the exactly-once processing on the leader, while AC would handle the subsequent state-update distribution.

3) AC/SC-based CL notification distribution: Following a CL adaptation in the AC, an *agreement* between the replicas is necessary to ensure the consistent global re-configuration of the CL in each controller. The agreement in *all* nodes can be ensured by reaching a consensus about the newly applied CL. Noted, the OCA block may execute at a single node at any point in time (e.g. the actual cluster leader) or each replica in distributed manner. The latter variant requires the nodes reaching a consensus on the new applied CL after collecting the remote replicas' respones (e.g. using a PBFT-like signalling protocol [19]). In any case, the OCA block does not represent an SPOF in the system.

VII. Evaluation Methodology

A. Application model

To present the trade-offs in deploying either Strong Consistency (SC), Adaptive Consistency (AC) or Eventual Consistency (EC) in a multi-controller testbed, we have implemented and evaluated a distributed load balancer application (SDN-LB) as a component of a modified OpenDaylight [15] distribution. The SDN-LB allows for the embedding of isolated independent services via a YANG-modeled REST interface, characterized by the type and cost (i.e. comprising a capacity requirement). Each SDN controller replica runs data store implementations for all three consistency models, and is enabled to accept new embedding requests.

A data-plane SDN-LB has already been investigated in the past in the context of the link-load distribution scenarios [11], [30]. However, we generalize the goals of the SDN-LB to support allocation of any type of resources (i.e. bandwidth/CPU/memory) on the selected optimal service node, given a subset of the feasible candidate nodes and their current utilizations in terms of the mappable resource as the inputs. The algorithm then decides to assign the service request, under consideration of hard resource constraints, on the node deemed as optimal w.r.t. total balance of resource utilization. We adapt the algorithm defined in [26] to facilitate immediate scheduling, i.e. an online resource mapping process.

We model the state of the current reservations and the available resources as in-memory state instances in our data store realizations. SDN-LB decisions are made based on the current value of these states. Upon each successful embedding, the current node utilization is updated to include the cost of the latest request. The controller is then in charge of disseminating the local reservation update using the update-distribution and commit mechanism implemented by the underlying data store.

In the case of the SC model, every single update in the data store, to each service node, is serialized by the RAFT leader, using the consensus abstraction. In the EC and AC framework, each new resource reservation necessitates an increment or decrement update to the respective CRDT PN-Counter object (ref. Alg. 1). Combined with the commutative increment/decrement operations, the PN-Counter ensures eventual convergence for both EC and AC models and thus represents a good data structure fit for resource tracking realization. In AC, the state-updates to the PN-Counter are queued and distributed across the cluster based on the CL-timer and maximum queue-distribution thresholds, governed by the currently applied CL. Indeed, the adaptation function (ref. Sec. V-D) adapts the CL, and thus manipulates the worst-case time required to synchronize the value of the counters. Thus, the adaptation affects the quality of the embedding decisions made by the replicas running the AC framework. The inefficiency metric provided as an input for the consistency adaptation ϕ maps to the approximation ratio introduced in Sec. V-C. Finally, in the case of EC, state-updates are queued in the state-specific FIFO distribution queue and are distributed as fast as possible (i.e. excluding any waiting period).

B. Data Store Realizations

To empirically compare the effects of deploying the different consistency models, we have implemented and integrated in OpenDaylight the three data store variations:

1) Strong Consistency: The evaluated SC data store is realized using the unmodified RAFT [7] implementation atop of Java and Akka.io[1] concurrency framework included in the OpenDaylight Boron-SR4 distribution. We have modeled the data-models required by the generic SDN-LB application using YANG[2] modeling language. We then synthesized these models into REST APIs using ODL's YANG Tools[3] compiler.

2) Adaptive Consistency: The implementation abstractions of the AC framework are based on the algorithms presented in Sec. IV. The framework is implemented as a set of Java bundles, and has been integrated in the OpenDaylight's OSGi environment as an in-memory data store in parallel to the SC data store. In the AC environment, similar to above, we expose the data-model for the SDN-LB application using the YANG and REST APIs. Additionally, the CL adaptation as well as the distribution of the CRDT state-updates (Sec. V-E) require a new protocol definition. We have used Google Protobuf[4] to describe the corresponding data structures, as well as to serialize the on-the-wire transmissions. Asynchronous replica acknowledgments are sent out to the senders in order to acknowledge the successful state-/CL-updates at the receivers.

3) Eventual Consistency: The EC implementation is based on the AC framework implementation. In the AC realization, the update-distribution queue thresholds are derived from the specification of the currently applied CLs. In EC, however, the CLs hold no relevance for the state distribution, hence the maximum queue sizes of the distributed state-updates are unbounded and can thus theoretically grow infinitely for very high service request arrival rates.

C. On topology and parameter selection

We base our evaluation of the consistency models using an in-band OpenFlow control plane and an emulated forwarding plane, consisting of a number of interconnected Open vSwitch (OVS) instances instantiated and isolated in individual Docker containers. We have emulated the Internet2 Network Infrastructure Topology as a representative of an ISP network, as

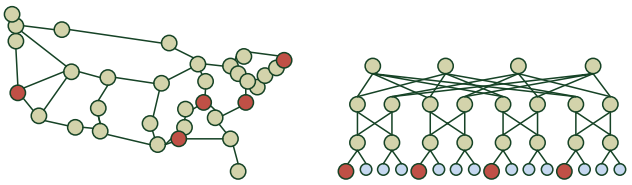
[1] Akka Clustering and Remoting - https://akka.io/

[2] YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF) - https://tools.ietf.org/html/rfc6020

[3] Yang Tools - https://wiki.opendaylight.org/view/YANG_Tools:Main

[4] Google Protocol Buffers - https://developers.google.com/protocol-buffers/

well as a standard fat-tree data-center topology, controlled by a 5- and 4-controller cluster, respectively. To reflect the delays incurred by the length of the optical links in the geographically scattered Internet2 topology, we assume a travel speed of light of $2 * 10^6 km/s$ in the optical fiber links. We derive the link distances and hence the propagation delays from the publicly available geographical Internet2 data[5]. The links of the fat-tree topology were modeled to incur a variable propagation and processing delays averaging $1ms$. In the ISP topology we leveraged a controller placement that targets the maximized robustness against the controller failures and a minimization of the probability of occurrence of switch partitions, as per the optimal placement presented in [31], [32]. The resulting controller placement is depicted in Fig. 3a. SDN controller replicas in the data-center topology are assumed to run on the leaf-nodes, deployed as virtual machines (VMs) (Fig. 3b), similar to the controller placement presented in [33].



(a) Internet2 topology [31] (b) Fat-tree topology [33]

Fig. 3. Exemplary network topologies and controller placements used in the evaluation of the SC, AC and EC frameworks. Elements highlighted in green and blue represent the forwarding and compute devices, respectively. Red elements are the OpenDaylight controller instances placed as per [31], [33].

We model the arrival rates of the incoming service embedding requests using a negative exponential distribution [34]. To emphasize the effects of the EC on the quality of decision-making in the SDN-LB application, we distribute the total request load non-uniformly across the controller replicas. The arrival rates and the per-replica load distribution weights are included in Table I. The Docker- and OVS-based topology emulator, as well as the 5 controller replicas, were deployed on a single commodity PC equipped with a recent multi-core AMD Ryzen CPU and 32 GB of RAM.

VIII. Results

A. Correctness of the SDN Application's Decision-Making

Fig. 4 visualizes an exemplary adaptation process in the AC framework. In particular, blue, green and cyan lines depict the CL applied for the SDN-LB-related CRDT PN-Counter instances on three different controller replicas. Red and black lines correspond to the actual capacity assignments managed by the SDN-LB instances on the different replicas. The resources are assigned on two different servers providing for the utilizable capacity. Indeed, in case of a strongly consistent SDN-LB (SC), the black and the red lines would continuously overlap as the state of reservations would be serialized and an optimal placement executed for each incoming service embedding request. Fig. 4b highlights the adaptation of the CL

[5]Pareto Optimal Controller Placement (POCO) -https://github.com/lsinfo3/poco/tree/master/topologies

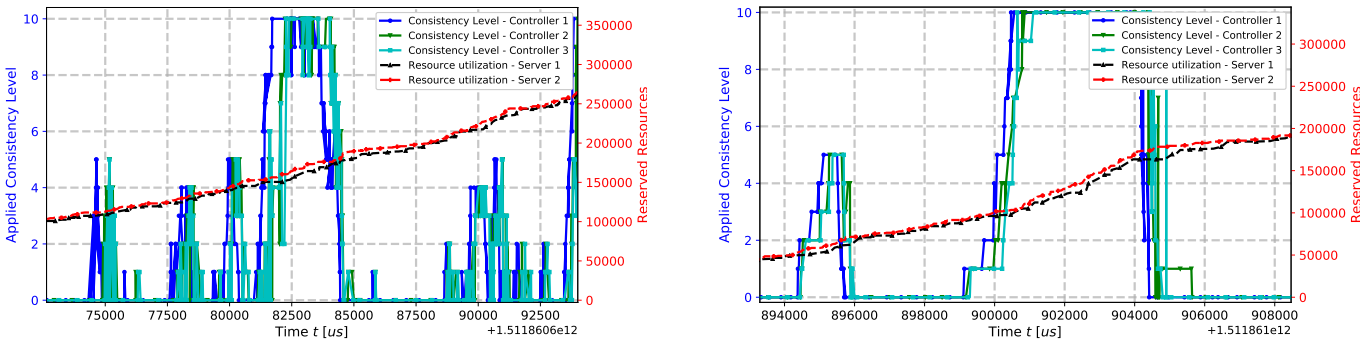
Parameter	Model	Value	Comment
Number of Replicas	SC, AC, EC	$[4^*, 5^+]$	Internet2$^+$ and fat-tree*
Consistency Levels (Granularity)	AC	[1..10]	Ref. Alg. 3 and 4
$CL_{QS}^{Ctr_k}$	AC	[3..15]	Ref. Alg. 5
$CL_{TO}^{Ctr_k}$	AC	[100..1000]	Ref. Alg **??**
Initially applied CL	AC	3	N/A
P_g	AC	0.2	Ref. Alg. 4
I_g	AC	0.2	Ref. Alg. 4
D_g	AC	0.1	Ref. Alg. 4
T_O	AC	2	Ref. Alg. 4
W	AC	5	Ref. Alg. 3 & 4
T_L	AC	1.5	Ref. Alg. 3
T_U	AC	3.5	Ref. Alg. 3
$SDN-LB_{\#C}$	SC, AC, EC	2	SDN-LB - No. service types
$SDN-LB_{\#S}$	SC, AC, EC	2	SDN-LB - No. servers
$SDN-LB_{C_{Cost}}$	SC, AC, EC	[500..600]	SDN-LB - Service cost
$C_{Weights}$	SC, AC, EC	$[1,1,2,1,5]^+$ $[1,2,2,5]^*$	Req. load for Internet2$^+$ and fat-tree* topologies

TABLE I
Parametrizations used during our study.

at the time point $905000\ us$, where the imbalance and thus the inefficiency of the SDN-LB lead to an adaptation trigger and a steep decrease of the utilized CL from 10 (the most relaxed CL) to 0 (the most strict CL). The consistency adaptation function modifies the maximum available queue capacity $CL_{QS}^{Ctr_k}$ for any new state-updates as well as the worst-case timeout $CL_{TO}^{Ctr_k}$, as per Table I. With the strictness of the applied CL, the SDN-LB resource assignment discrepancy decreases, but the overhead of blocking time for new state-updates increases.

Fig. 5 depicts the measured inefficiencies in the SDN-LB's assignment for the case of the AC- and the EC-based models in the fat-tree topology. The AC model uses the threshold-based adaptation of the CLs, and depicts a faster convergence in the case of the fast-mode based AC update distribution, compared to the EC case. Indeed, the fast-mode converges first to the worst-case inefficiency for all depicted request arrival rates. The batched-mode shows a lower average inefficiency than the EC model in the case of the request arrival rates of $2\ ms$. For the case of less frequent $5\ ms$ arrivals, batched-mode state-update distribution shows higher mean inefficiency than the EC case. This is explained by the fact that the batched-mode distribution of the state collects and distributes the outstanding state-updates only after a queue-threshold has exceeded or a scheduled CL-governed timer has expired. However, the time duration taken to fill up the distribution queue for the batched-mode is inversely proportional to the rate of update arrivals. Hence, compared to EC, for slow request arrivals the staleness caused by delaying the updates' distribution offsets the benefits of bounding the total number of state-updates.

Fig. 6 portrays the comparison of threshold- and PID-based AC consistency adaptation, as well as the EC consistency model in the case of the Internet2 topology. Similar to the result in Fig. 5, the usage of bounded state-update distribution queues leads to a bounded worst-case inefficiency metric with both evaluated AC adaptation functions. Interestingly, threshold-based adaptation depicts a lower average- and worst-case performance for the estimated inefficiency, compared to the PID-based model. This behavior could be caused by the tendency of the PID-model to relax the frequency of state synchronization more often, thus leading to a slightly higher inefficiency, at the benefit of a higher transaction throughput.



(a) PID-based CL adaptation

(b) Threshold-based CL-adaptation

Fig. 4. *PID-based* (a) and *threshold-based* (b) adaptation of consistency levels. The PID-based approach is more volatile compared to a rather resistant threshold-based approach. As per Alg. 3 the threshold-based approach keeps the current CL unmodified, as long as the measured inefficiency stays in a specific range (i.e. between the specified upper and lower thresholds). The PID-based approach, on the other hand, oscillates around the specified target inefficiency value. For brevity, we only depict the historical data for $C = 3$ controllers here (of a total of $C = 5$).

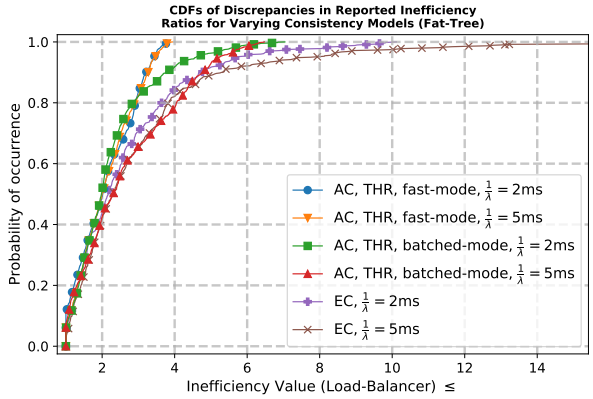


Fig. 5. CDFs of reported inefficiencies (approx. ratios) for various deployed consistency models and request arrival rates in the fat-tree topology (ref. Sec. 3b). High inefficiency values indicate a more unbalanced performance of the distributed SDN-LB instances. Compared to the EC model, the AC model with the threshold-based adaptation converges faster to the worst-case with both depicted distribution modes. The fast-mode configurations result in the lowest worst-case inefficiency values. For the batched-mode distribution, the system has a similar average inefficiency as the EC. This is related to the delayed distribution of the state-updates, which is initiated only once the distribution queue is fully utilized. In the case of less frequent request arrivals (i.e. for $1/\lambda = 5ms$), the distribution queue takes the longest to fully fill up.

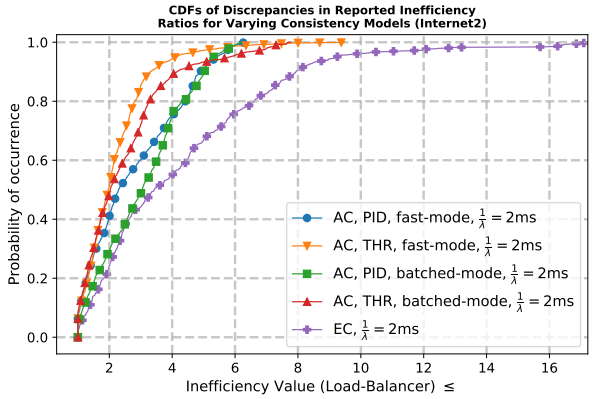


Fig. 6. CDFs of the reported inefficiencies (approx. ratios) for the Internet2 topology. The average- and the worst-case inefficiencies of the EC model are larger than in the case of the fat-tree topology, as a result of the larger controller-to-controller network delays in the geographically spaced Internet2 network. The AC model deployment does not suffer from this issue because of the limited amount of incurable staleness, guaranteed by the maximum amount of unsynchronized updates. Threshold-based adaptation model shows a better performance than PID-based adaptation, possibly having to do with the higher probability of the relaxation of CLs in PID-mode (refer to Fig. 4).

Fig. 7 further emphasizes the effect of the design of CL configurations on the average-case measured inefficiency. The experienced worst-case inefficiency scales with the number of allowed isolated state-updates at a single replica. Hence, careful parametrization of the CL mappings to the maximum state-update distribution queue sizes and the timer durations is necessary to ensure the right trade-off between inefficiency and synchronization overhead.

B. The overhead of distribution of state-updates

We define the update commit-time as the time duration required to accept, distribute and confirm a single new state-update in the underlying distributed controller data store. Fig. 8 depicts the box-plots for the measured commit times in the case of SC and AC models.

AC (local) showcases the time required to apply an update to the local replica and return a corresponding acknowledgement at the requesting application. The *AC (W=3)* case corresponds to the time duration needed to converge the state-update request at 3 of 5 replicas. Thus, in the case of failure of 2 controller replicas, the remaining replicas can still eventually converge on the latest state value.

The analogue case for the SC replicas is depicted in the two right-most box-plots. An SC cluster of 5 replicas tolerates a maximum of 2 controller failures (because of the majority constraint imposed by the CAP theorem [22]). Compared to the *AC (W=3)* case, the SC model offers the advantage of the serialized data stores updates. This benefit, however, comes at a high cost of minimum, average and worst case commit times, especially when state-update requests are received at one of the follower replicas. Indeed, an incoming state-update request at a leader replica leads to the faster commit confirmations,

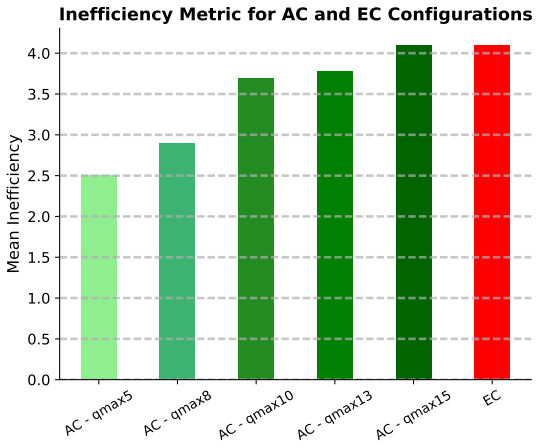


Fig. 7. The measured mean inefficiency for the different $CL_{QS}^{Ctr_k}$ parametrizations for the most-relaxed CL configuration. The larger the maximum allowed state-update queue size, the larger the sojourn time of the locally admitted state-updates without enforced data store synchronization. Thus, the inefficiency of the system scales with the number of unseen state modifications. In the case of *AC - qmax15*, up to 15 state-updates may be enqueued for a particular controller state without enforced synchronization. This case depicts a similar inefficiency as the EC case (without any staleness bounds). This is a result of a limited processing power of our testbed, where, at this point, clients are either unable to request a higher number of parallel concurrent REST-based SDN-LB-requests, or the controller instances are unable to concurrently process a higher number of individual updates than the 15 depicted in this case. We expect the maximum inefficiency for the EC case to deviate further in the scenarios of more scalable testbeds.

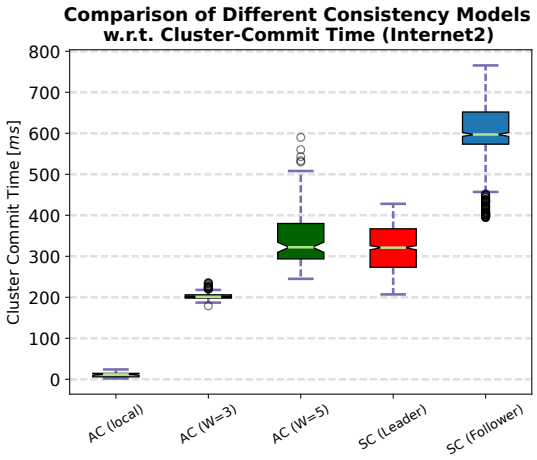


Fig. 8. The commit times using the AC and SC data stores. The average response time corresponds to the *AC (local)* case. The AC framework configuration that replicates to W=3 replicas *AC (W=3)*, depicts a better worst-case commit-time performance, while tolerating the same availability as a comparable SC configuration (with a total of 5 controllers, and *W=3* during blocking writes). The worst-cases *AC (W=3)* and *AC (W=5)* blocking periods, however, only occur when the state-update queue is filled and a distribution is necessary to ensure the *staleness* bound. On the other hand, the throughput of the SC cluster suffers specifically in the case where updates are committed on the RAFT nodes assigned a follower role. This confirms the results of [9].

Consistency Model	Distr. Time [s]	Avg. PPS	Avg. Packet Size [B]	Avg. Load [B/s]
EC	25.12	3802	124.5	469k
SC	115.12	326.82	2612.6B	844.2k
AC (Batched-Mode)	39.501	3460	105.4	365k
AC (Fast-Mode)	40	3552	102.5	372k

TABLE II
PER-REPLICA LOAD WHEN SERVING 1000 SDN-LB REQUESTS IN A 5-CONTROLLER CLUSTER SYNCHRONIZED USING SC, AC OR EC.

as one less uni-directional packet-transmission delay from the followers to the leader is required. The worst-case commit time in the *AC (W=5)* case is similar to the optimistic SC case. It results in a relatively high commit time, because of the geographical separation of the controllers, native to the Internet2 topology. The *AC (W=5)* case, however, tolerates a total of 4 controller replica failures and thus offers a higher availability compared to the SC deployment that would require a minimum of $2*4+1=9$ controller instances to tolerate the same amount of controller failures. The *SC (Follower)* case considers the update requests received at one of the follower controller replicas that require transmission and subsequent serialization at leader, as well as the majority consensus to commit the update. In the geo-replicated scenarios such as in the case of Internet2 topology, this case may not be neglected.

Table II portrays the incurred message load in controller-to-controller communication for the transmission of state-updates resulting from 1000 subsequent SDN-LB mapping requests, distributed uniformly across all instances. The portrayed result considers the average *per-instance* overhead in a cluster of five replicas. The batched-mode in the AC framework incurs the lowest message overhead because of its useful property of aggregation of the state-updates. The SC mode depicts a lower number of frames transmitted during the per-second measurement intervals. However, the average frame size of the SC-transmitted messages is larger compared to the AC/EC deployment. The total time taken to serve 1000 SDN-LB embedding requests in the SC deployment takes a longer time as each write *and* read request is serialized, and no concurrent state modifications are allowed to take place. Previous measurements of the RAFT implementation in OpenDaylight [9], [10] have proven that the overhead of read operations in a consensus-based cluster is similar to that of the write operations, since cluster-wide reads/writes are necessary to reach consensus on the latest state values. Lastly, the distribution time of AC suffers compared to the EC model, since EC processes transactions as fast as possible and does not implement the overhead of consistency adaptation.

IX. RELATED WORK

1) On Strong Consistency: Ongaro et al. [18] and Howard et al. [7] provide the initial experimental performance evaluations of the RAFT consensus algorithm. They focus on the evaluation of the performance of leader election procedure during the controller failure scenario. Suh et al. [9], [10] experimentally measure the throughput and the recovery time of a RAFT-enabled cluster with up to 5 SDN controllers for the use case of flow table reconfigurations. These works do not discuss the effect of failures on the quality of decision-making in the context of SDN applications nor do they cover the aspects of RAFT scalability for high-throughput applications.

Ravana [6] is a proposal for a distributed SDN controller that provides for a total-order of processed control messages, and ensures exactly-once delivery invariant for switch (re-)configurations. The focus of Ravana is on ensuring the

performance guarantees in the face of failures, and not on leveraging the different consistency models for supporting the scalability of the distributed SDN control plane.

In our measurements, we continuously assume the availability of strict serialization and thus the exactly-once and total-order semantics in the SC model. However, a recent research [35] has showcased two scenarios where the interplay of a RAFT-enabled controller cluster and SDN data plane may introduce inconsistency in the control plane. The authors formulate and reproduce the problems of the oscillating and non-converging RAFT leader election, and propose a partial solution. However, they leave the validation of the solution for future work. Thus, even if we compare the AC approach to an idealized SC scenario, we note that RAFT still may lead to poor correctness/availability performance in some cases.

2) On Eventual Consistency: HyperFlow is an EC publish-and-subscribe data-broker approach [3]. In HyperFlow, each controller sends the state-update requests to an external data store that disseminates the state-updates to other controllers. HyperFlow centralizes the state collection and distribution entity in the external data store, thus effectively shifting the issue of SPOF from the context of an SDN controller to another centralized instance and not resolving the SPOF itself. We focus on the internalization of the data store to stay compatible with the current clustering solutions. SCL [12] provides a methodology for preserving safety and liveness invariants without deploying consensus. The authors rely on the *quiescent* period where, during a period with no data plane changes, all controllers eventually converge to same view to ensure correct operation. This can, however, only be guaranteed in networks with very limited reconfiguration dynamics, which is why SCL may occasionally lead to a disagreement of controller views. Our AC concept does not rely on quiescent period. Similarly, we do not rely on the availability of switch agents to guarantee the exactly-once execution semantics, which in contrast SCL does.

Levin et al. [11] evaluate the impact of an inconsistent global network view on the load balancer's performance assuming flexible frequencies of synchronization periods. Their results suggest that an inconsistency in the SDN control state view across multiple controller instances significantly degrades the performance of the SDN applications agnostic to the underlying state distribution. Contrary to our work, the authors generalize the synchronization procedure to a periodic task with flexible periods. In the case of SC, we consider a continuous synchronization model where on-the-wire transactions must be serialized in order to ensure a total ordering of decisions. In the EC case, we assume non-periodic state synchronization, as this provides for a more realistic and better performance and lower penalty of state staleness, especially for the case of higher request arrival rates.

3) On Adaptive Consistency: In [14], we have introduced an AC model that employs the concept of 'strong eventual consistency', along with a 'cost-based' approach for quantifying penalties induced by the successfully detected state-merge conflicts. We used simulation to evaluate our model on an example of an SDN routing application and have motivated the potential performance gains.

In [13], the authors compare an adaptive approach to the state synchronization between the controllers with an approach of using the non-adaptive controllers that synchronize state with a constant synchronization period. The authors deploy an adaptation module to apply one of the pre-configured fixed synchronization intervals, which makes the approach inflexible for frequent network changes (i.e. varying controller request loads and network congestions). In contrast, we define an adaptation function which manages the new update admission in order to preserve the worst-case staleness bounds. Our work extends the conclusions on the practical applicability of AC by considering constant re-adaptation of the consistency levels.

TACT middleware [36] enforces consistency bounds among the replicas of a distributed system. To bound the level of inconsistency, TACT defines consistency measures, including: i) the *order error*, which limits the number of tentative writes that can be outstanding at any replica; ii) the *numerical error*, which bounds the difference between the value delivered to the client and the most consistent value; iii) and *staleness*, which places a probabilistic real-time bound on the delay of propagating the writes. A well-known arrival rate for the incoming requests is used to estimate the probabilistic staleness bound. We distinguish ourselves from TACT by introducing an admission control mechanism for serving new state modifications at any randomly selected serving instance. Thus, we proactively place deterministic bounds on the maximum number of allowed isolated local state updates.

A corner case of the exceeded limit values for isolated PN-Counter state-updates is related to the issue of conflicting over-reservations, previously discussed in [14]. Balegas et al. [37], [38] solve this issue by implementing an escrow-based bounded counter CRDT, so to guarantee that a value of a counter never exceeds some limit value.

X. Conclusion and Outlook

This works presents a realization of an Adaptive Consistency (AC) framework. On an example of 5-controller SDN cluster and two realistic network topologies, we highlight its advantages w.r.t.: i) the control plane response time compared to the Strong Consistency approach; ii) the decision-making efficiency compared to the Eventual Consistency approach; iii) the generated controller-to-controller load compared to the both approaches. We introduce two distribution abstractions that enable the controller-to-controller state exchange, while minimizing the response time and the generated average load. Furthermore, we present two adaptation functions, that adapt the system in a closed-loop manner, so that the target inefficiency incurred by the eventuality of state delivery persists according to the SDN application's expectations.

Future works should evaluate the AC framework in scenarios comprising a larger number of controllers. Large-scale demonstrations could additionally emphasize the benefits over the alternative consistency approaches. The adaptation functions take as input the SDN application-triggered inefficiency reports. The benefit of the consistency adaptation comes at the expense of an expanded model parametrization space and the necessity of an SDN application re-design. Further attention

should thus be given to simplifying the related development efforts, e.g. by providing sane configuration defaults or by automating the generation of required parametrizations [39].

ACKNOWLEDGMENT

This work has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement number 780315 SEMIOTICS and in parts under grant agreement number 647158 FlexNets (by the European Research Council). We are grateful to Majda Glotic, Dr. Johannes Riedl, Valeh Sabziyev, and the reviewers for their useful feedback and comments.

REFERENCES

[1] F. Sardis *et al.*, "Can QoS be dynamically manipulated using end-device initialization?" in *Communications Workshops (ICC), 2016 IEEE International Conference on*. IEEE, 2016.
[2] D. Kreutz, Ramos *et al.*, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, 2015.
[3] A. Tootoonchian *et al.*, "Hyperflow: a distributed control plane for openflow," *Proceedings of the 2010 internet network ...*, 2010.
[4] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," in *OSDI*, vol. 10, 2010.
[5] P. Berde *et al.*, "ONOS: towards an open, distributed SDN OS," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014.
[6] N. Katta *et al.*, "Ravana: Controller fault-tolerance in software-defined networking," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM, 2015.
[7] H. Howard *et al.*, "Raft Refloated: Do We Have Consensus?" *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, 2015.
[8] E. Sakic *et al.*, "Response Time and Availability Study of RAFT Consensus in Distributed SDN Control Plane," *IEEE Transactions on Network and Service Management*, 2017.
[9] D. Suh *et al.*, "Toward Highly Available and Scalable Software Defined Networks for Service Providers," *IEEE Communications Magazine*, vol. 55, no. 4, 2017.
[10] D. Suh *et al.*, "On performance of OpenDaylight clustering," in *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE, 2016.
[11] D. Levin *et al.*, "Logically centralized?: State distribution trade-offs in software defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012.
[12] A. Panda *et al.*, "SCL: Simplifying Distributed SDN Control Planes," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017.
[13] M. Aslan *et al.*, "Adaptive consistency for distributed SDN controllers," in *Telecommunications Network Strategy and Planning Symposium (Networks), 2016 17th International*. IEEE, 2016.
[14] E. Sakic *et al.*, "Towards adaptive state consistency in distributed SDN control plane," in *IEEE International Conference on Communications*, 2017.
[15] J. Medved *et al.*, "Opendaylight: Towards a model-driven SDN controller architecture," in *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*, 2014.
[16] C. Cachin *et al.*, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.
[17] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, 2006.
[18] D. Ongaro *et al.*, "In Search of an Understandable Consensus Algorithm," in *USENIX Annual Technical Conference*, 2014.
[19] M. Castro *et al.*, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, 2002.
[20] B. M. Oki *et al.*, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. ACM, 1988.
[21] I. Moraru *et al.*, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013.
[22] A. Panda *et al.*, "CAP for networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013.
[23] S. Gilbert *et al.*, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, 2002.
[24] Y. Saito *et al.*, "Optimistic replication," *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, 2005.
[25] M. Shapiro *et al.*, "A comprehensive study of Convergent and Commutative Replicated Data Types," Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, 2011.
[26] K. J. Naik *et al.*, "A Cost Greedy Price Adjustment based Job scheduling and Load Balancing in Grids," *International Journal of Computing & ICT Research*, vol. 10, no. 1, 2016.
[27] A. Khurshid *et al.*, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, 2012.
[28] W. Zhou *et al.*, "Enforcing Customizable Consistency Properties in Software-Defined Networks," in *NSDI*, 2015.
[29] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, 1990.
[30] L. Schiff *et al.*, "In-band synchronization for distributed SDN control planes," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 1, 2016.
[31] D. Hock *et al.*, "POCO-framework for Pareto-optimal resilient controller placement in SDN-based core networks," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014.
[32] S. Lange *et al.*, "Heuristic approaches to the controller placement problem in large scale SDN networks," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, 2015.
[33] X. Huang *et al.*, "Dynamic Switch-Controller Association and Control Devolution for SDN Systems," *arXiv preprint arXiv:1702.03065*, 2017.
[34] D.-C. Juan *et al.*, "Beyond poisson: Modeling inter-arrival time of requests in a datacenter," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2014.
[35] Y. Zhang *et al.*, "When Raft Meets SDN: How to Elect a Leader and Reach Consensus in an Unruly Network," in *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 2017.
[36] H. Yu *et al.*, "Design and evaluation of a continuous consistency model for replicated services," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 2000.
[37] V. Balegas *et al.*, "Putting consistency back into eventual consistency," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015.
[38] V. Balegas *et al.*, "Extending eventually consistent cloud databases for enforcing numeric invariants," in *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on*. IEEE, 2015.
[39] M. Aslan *et al.*, "A Clustering-based Consistency Adaptation Strategy for Distributed SDN Controllers," *CoRR*, vol. abs/1705.09050, 2017.

Ermin Sakic (S′17) received his B.Sc. and M.Sc. degrees in electrical engineering and information technology from Technical University of Munich in 2012 and 2014, respectively. He is currently with Siemens AG as a Research Scientist in the Corporate Technology research unit. Since 2016, he is pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering at TUM. His research interests include reliable and scalable Software Defined Networks, distributed systems and efficient network and service management.

Wolfgang Kellerer (M′96 – SM′11) is a Full Professor with the Technical University of Munich (TUM), heading the Chair of Communication Networks at the Department of Electrical and Computer Engineering. Before, he was for over ten years with NTT DOCOMO's European Research Laboratories. He received his Dr.-Ing. degree (Ph.D.) and his Dipl.-Ing. degree (Master) from TUM, in 1995 and 2002, respectively. His research resulted in over 200 publications and 35 granted patents. He currently serves as an associate editor for IEEE Transactions on Network and Service Management and on the Editorial Board of the IEEE Communications Surveys and Tutorials. He is a member of ACM and the VDE ITG.