Adaptive distributed SDN controllers: Application to Content-Centric Delivery Networks

Fetia Bannour, Sami Souihi, Abdelhamid Mellouk

▶ To cite this version:

Fetia Bannour, Sami Souihi, Abdelhamid Mellouk. Adaptive distributed SDN controllers: Application to Content-Centric Delivery Networks. Future Generation Computer Systems, 2020, 113, pp.78 - 93. 10.1016/j.future.2020.05.032 . hal-03492191

HAL Id: hal-03492191
https://hal.science/hal-03492191v1

Submitted on 18 Jul 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

Version of Record: https://www.sciencedirect.com/science/article/pii/S0167739X19328316
Manuscript_229ab46192729c6717f24016db8f05c9

Adaptive Distributed SDN Controllers : Application to Content-Centric Delivery Networks

Fetia Bannour, Sami Souihi and Abdelhamid Mellouk
LISSI Laboratory, University of Paris-Est Créteil (UPEC), France

***Abstract*—This paper aims to propose a deployment solution for Content-Centric Delivery Networks (C-CDN) based on the SDN technology in an IoT-like environment. To meet today's network requirements in terms of reliability, scalability and performance and to match the specific IoT needs, the SDN architecture needs to be physically-distributed, but logically-centralized. The key idea is to consider the knowledge plane natively provided by the SDN controllers to build Information-Centric Networking (ICN) applications (e.g. Video distribution services). To achieve this, we put forward an adaptive and continuous consistency model for the distributed SDN controllers in large-scale deployments. More specifically, we propose to turn the standard replication technique into a scalable and intelligent replication strategy following Quorum-replicated consistency. That strategy uses the read and write Quorum parameters as adjustable control knobs for a fine-grained consistency level tuning. The main purpose is to find at run-time appropriate partial Quorum configurations that achieve, under changing network and workload conditions, balanced trade-offs between the SDN application's continuous performance and consistency requirements. When compared to ONOS's static consistency model, our approach which was implemented for a Content-Centric Delivery Network (C-CDN) application that we designed on top of ONOS, proved efficient in minimizing the application's inter-controller overhead, while satisfying the SLA-style application requirements.**

***Index Terms*—Distributed SDN control, scalability, eventual consistency, adaptive consistency, Quorum consistency, CDN, CCN, ICN, C-CDN, ONOS controller, IoT applications.**

I. Introduction

Content-Centric Networking (CCN) or Information-Centric Networking (ICN) has recently received a lot of attention as a potential alternative paradigm for the future Internet architecture. In ICN, the focus is placed on the data itself instead of its hosting location, as content objects are accessed from anywhere in the network rather than from specific end hosts [1]. More specifically, ICN advocates a shift in the current network architecture from an IP-based and host-oriented communication model to a content-oriented model in which content information is accessible by names rather than host addresses. In particular, location-independent naming, in-network caching and name-based routing are the most important features characterizing an ICN architecture [2]. The main objective of ICN is indeed to improve content delivery, enhance data dissemination over the network, improve the overall network performance, and most importantly fulfill the users' requirements [3].

By providing easier data access, more efficient resource utilization, enhanced content-based security, better mobility and improved scalability, ICN has appeared as a viable framework to support the Internet of Things (IoT) technology which is information-based and content-oriented in nature. In fact, the ICN concept offers a network infrastructure that is well suited to meet IoT application requirements [4] and to address the new communication patterns and the practical challenges associated with interconnecting several billions of heterogeneous and smart IoT devices.

However, designing and deploying an ICN-based platform for IoT has brought up new challenges. Some of these challenges can be answered with the Software-Defined Networking (SDN) paradigm. The SDN concept could indeed provide the degree of flexibility, automation and security that is needed for the control and management of expanding IoT networks. Recent studies [5, 6] argue that network softwarization with SDN can be beneficial for IoT as it is expected to combat the growing complexity facing IoT services, and also facilitate the dynamic and automatic configuration of IoT platforms.

More precisely, SDN represents a network paradigm shift that decouples the control plane and the data plane of the network making it easier to program the network. That said, SDN natively offers a Knowledge Plane (KP) that can be leveraged to build ICN services in an IoT ecosystem.
In this context, various deployment configuration options have been suggested for ICN using SDN capabilities, most of which seemed attractive for IoT scenarios. In particular, RFC 8763 [7] outlines the different proposals being made regarding the various ICN approaches that might be well suited to SDN-based networks. These approaches study the use of ICN "as-a-Clean-slate", "as-an-Overlay", "as-an-Underlay" or "as-a-Slice" while reviewing the different possible ways to leverage the emerging SDN paradigm for ICN goals.

On the other hand, to meet today's network requirements in terms of scalability, reliability and performance and to match the specific IoT needs, the SDN architecture should be distributed (logically-centralized but physically-distributed [8]). For this purpose, consistency has been regarded as an essential design principle. The common distributed SDN architecture uses conventional consistency models to manage the distributed network state among the SDN controller replicas in the cluster. As explained in [9], the consistency models used in SDN can be categorized into *strong*, *eventual* and *weak* [10, 11]. These static and standard consistency models have both advantages and drawbacks.

In large-scale SDNs and particularly in IoT-based environments, the *Strong Consistency* control model might be extremely expensive and costly to maintain for certain applications. Indeed, that model requires important synchronization efforts among the controller replicas at the cost of causing se-

© 2020 published by Elsevier. This manuscript is made available under the CC BY NC user license
https://creativecommons.org/licenses/by-nc/4.0/

rious network scalability and performance issues. By contrast, the *Eventual Consistency* control model implies less inter-controller communication overhead as it sacrifices the strict consistency guarantees for higher availability and improved performance. In practice, many scalable control applications (running) in modern distributed storage systems like Apache's Cassandra [12] and Amazon's Dynamo [13] opt for eventual consistency to provide such requirements on a large scale. However, these applications might suffer from the associated relaxed (weak) consistency guarantees that may temporarily allow for too much inconsistency.

Recent research works in the area of distributed SDN control have explored the concepts of *Adaptive Consistency* control for various applications [14, 15]. Such categories of consistency models follow different adaptation strategies that mainly focus on dynamically adjusting the levels of consistency at run-time under various network conditions in order to meet the application-defined consistency and performance needs.

Unlike strong and eventual consistency options, adaptive consistency control models leverage the broad space of intermediate consistency degrees between these two extremes. They, indeed, use time-varying consistency levels to support balanced real-time trade-offs between the desired consistency and performance requirements which can be specified in the application-defined Service-Level Agreements (SLAs) [16].

In this work, we propose an adaptive consistency model (based on eventual consistency) for the SDN controller applications that are deployed in large-scale IoT-like networks. We target the class of applications that tolerate relaxed forms and degrees of eventual multi-consistency for the sake of scalability and performance but yet can benefit from improved consistency features. More specifically, our approach mainly consists in turning the eventual consistency model currently used by popular SDN controller platforms like ONOS into an adaptive consistency model based on a scalable and more intelligent replication strategy following Quorum-replicated consistency models.

In the following, we list the main contributions of this work with respect to our previous work [17] :

- In this work, we included a more comprehensive analysis of the application's inter-controller overheads. Unlike our previous work, we tried to study the impact of using different read and write Quorum sizes on the read and write inter-controller overheads which are analyzed separately.
- Besides, we enhanced the Q-learning reinforcement learning algorithm used by our adaptive consistency model. We adopted a *stateful* variant of the Q-learning approach in which we consider the transition between states. We also followed an incremental strategy where only some actions are possible to transition from one state to another. In our Q-function, we gave equal importance to the immediate rewards for the current action and the expected future rewards for all possible actions in the next state ($\gamma = 0.5$). This is in contrast to our previous *stateless* approach where the actions performed by the agents are considered as separate events with no state transition rules, and where immediate rewards for current actions are given the utmost importance ($\gamma = 0$).
- Moreover, we developed new adaptive mechanisms that we integrated into our adaptive consistency strategy for the distributed SDN controllers. Thanks to these new features, our consistency model has the ability to adapt, not only to the dynamic application SLA requirements like in the previous proposal, but also to the dynamic changes in application workloads (a read-dominated workload, a write-intensive workload and a balanced workload). Indeed, our main objective is to find at run-time, and under varying network and application conditions, optimal Quorum replication configurations that achieve balanced trade-offs between the application's continuous performance (*latency*) and consistency (*staleness*) requirements. These real-time trade-offs should provide minimal read and/or write inter-controller overheads while satisfying the application-defined thresholds specified in the given application SLA. Our approach was implemented for a Content-Centric Delivery Networks (C-CDN) application that we developed on top of the distributed open-source ONOS controllers. Our C-CDN application, which natively works on top of a distributed SDN controller platform, raises the possibility of transitioning towards ICN solutions that can be integrated into networks that support virtualized architectures in the form of SDN.

The rest of this paper is organized as follows: In Section II, we conduct a background review of eventual consistency models in modern distributed data-store systems. Inspired by the modern consistency techniques used in these scalable data-stores, we present, in Section III, our adaptive and continuous Quorum-based consistency model for the distributed ONOS controllers in large-scale deployments. In Section IV, we describe our methodology for implementing the proposed consistency strategy on a C-CDN-like application that we designed on top of the ONOS controllers. Finally, Section V elaborates on the test scenarios we developed to evaluate our proposal and discusses the experimental results.

II. Background on Eventual Consistency in Distributed Data-Stores

A. Consistency and performance Metrics

Guaranteeing the consistency of replicated data in distributed database systems has always been a challenging task. Today's fundamental consistency models (e.g. strong consistency, sequential consistency, causal consistency, eventual consistency) ensure different discrete levels and degrees of consistency guarantees. For instance, the strong consistency model offers up-to-date data, but at the cost of high latency and low throughput. As a result, weaker forms of consistency (in the consistency spectrum)-most notably the popular notion of eventual consistency- have been widely adopted in modern distributed data-stores which need to be highly-available, fast and scalable [13, 12]. However, despite being regularly acceptable and desirable in practice for the latency and throughput benefits they offer, eventual consistency models provide no bounds on the inconsistency of data they return. Another

major limitation of these models is that the trade-offs they make among consistency and performance (latency) are difficult to evaluate. In fact, measuring the concrete consistency guarantees of eventually-consistent distributed stores remains challenging.

Yu and Vahdat proposed the TACT framework [18] which fills in the consistency spectrum by providing a continuous conit-based and multi-dimensional consistency model. The latter can be leveraged by replicated Internet services to dynamically choose their own tunable and fine-grained trade-offs between consistency, performance and availability, based on client, service and network characteristics. In TACT, the authors quantify consistency by bounding the amount of inconsistency or divergence of the replicated data items in an application-specific-manner using three application-independent metrics: *Numerical error*, *Order error* and *Staleness*. Besides, Bailis *et al.* [19, 20] presented an approach based on a set of probabilistic models to predict the expected consistency guarantees as measured by the *staleness* of reads observed by client applications in eventually-consistent Dynamo-style partial quorum systems. The authors introduced the WARS Probabilistically Bounded Staleness (PBS) model which provides bounds on the expected staleness in terms of both versions (using the *k-staleness* metric) and wall clock time (using the *t-visibility* metric). Another interesting work found in [21] proposes an automated self-adaptive consistency approach called Harmony which embraces an intelligent estimation of the *stale read rate* metric in Cloud storage systems, allowing to automatically adjust the consistency level at run-time according to application needs. That was achieved by elastically scaling up or down the number of replicas involved in read operations to preserve a low tolerable fraction of stale reads. When compared to the static eventual consistency approach in Cassandra, Harmony significantly enhances the consistency guarantees by reducing the rate of stale reads while adding only minimal latency. Besides, when compared to the strong consistency model in Cassandra, Harmony improves the performance of the system by increasing the overall throughput while maintaining the desired consistency requirements of the applications.

B. Adaptive consistency control

Modern distributed database systems supporting standard eventual consistency models suffer from the inevitable trade-offs between consistency, availability and request latency. To overcome this major limitation, these storage systems have introduced the concept of adaptive consistency in order to find appropriate consistency options depending on application requirements and system conditions. In literature, adaptive consistency techniques have been broadly classified into two categories: *user-defined* and *system-defined* [22].

In contrast to user-defined adaptive consistency methods where data and operations need to be mapped in advance to the desired consistency levels (using specific parameters), system-defined adaptive consistency methods take into account the fact that user and system behaviors might change dynamically over time making the consistency decision-making process challenging and tricky for application developers. That is why, system-defined techniques usually rely on system intelligence and adaptability to automatically provide fine-grained control over the consistency guarantees at run-time. Accordingly, many factors can be considered to dynamically estimate and predict the appropriate system consistency, including data access patterns, system load, but also the application's consistency SLAs as discussed in Section II-A. One famous form of system-defined adaptive consistency is the continuous consistency model used in TACT [18].

Additionally, it is worth mentioning that designing system-defined adaptive consistency (falling within the scope of this paper) requires careful considerations of the appropriate consistency adaptation strategy. In particular, existing adaptive mechanisms use different control knobs to be configured for consistency tuning such as the *consistency level*, the *artificial read delay*, the *replication factor* and the *read repair chance* [23].

C. Existing modern tunable consistency systems

To ensure eventual consistency in existing distributed datastore systems, different replication mechanisms and reconciliation techniques can be implemented. The most commonly used replication mechanism is optimistic (lazy) replication which is believed to offer high-availability, performance and scalability. There are several variants of optimistic replication systems [24], but the common basic concept is to passively replicate the updates to other replica nodes in the system, and let them be read by clients without the need to wait for a prior synchronisation of all the copies. In some implementations, a minimum number of nodes (called a quorum) are involved in the updates. Other design choices like the number of masters in the system, might also result in different optimistic replication system variants.
Along with optimistic replication, eventual consistency systems may resort to extra conflict resolution and reconciliation techniques in order to reconcile differences (after they occur) between multiple copies of distributed data. The most appropriate approach to reconciliation depends on the considered application. For example, Amazon's DynamoDB [13] uses Vector Clocks for conflict resolution. In general, reconciliation strategies include read repair, Anti-Entropy recovery, write repair and asynchronous repair operation mechanisms.

Popular distributed Cloud storage systems, most notably Apache's Cassandra [12], Amazon's Dynamo [13], Riak [25], and Voldemort [26] opt "by default" for eventual consistency guarantees in exchange for extremely high availability. However, these systems attempt to provide the applications with more control over the consistency and performance trade-offs via built-in settings and features. They indeed extend the concept of eventual consistency by offering tunable consistency levels for application developers and users based on Dynamo-style quorum replication policies.

In Cassandra, the consistency level specifies the size of a quorum for reads and writes, which is the appropriate number of replicas in the cluster that must acknowledge a read or write operation before considering the operation

successful. The native and well-known consistency options (levels) in Cassandra are three: ONE replica, a QUORUM of replicas, and ALL of the replicas. Accordingly, different choices of read and write consistency levels (quorums) ensure different consistency guarantees. For instance, to achieve the highest strong consistency, different quorum configurations may be selected, but they must satisfy the overlapping quorum property between read and write replica sets (*strict quorums*). On the other hand, to provide acceptable consistency with improved availability (minimum latency), it is desirable to use weaker forms of consistency such as the default eventual consistency option. Such weak consistency levels can be achieved through different quorum configurations that do not satisfy the overlapping quorum intersection property (*partial (non-strict) quorums*).

As a result, modern storage systems like Cassandra can be classified in the category of *user-defined* adaptive consistency as discussed in the previous section, given that they offer multiple consistency options. However, although these systems offer adaptive consistency on top of tunable consistency models that are aimed at creating balanced trade-offs between consistency and performance, it is usually difficult for application developers to decide in advance about the required consistency options for a particular request [22].

III. The proposed adaptive Quorum-inspired consistency for ONOS

In this paper, we propose a novel quorum-based and *system-defined* adaptive consistency model for the distributed ONOS controllers. Our approach was partly inspired by the quorum-replicated consistency techniques used by the modern data-store systems discussed in Section II-C.

The ONOS approach to state consistency in the latest releases was described in detail in [9]. It mainly relies on two consistency schemes that provide two levels of consistency: strong consistency and eventual consistency. While the strong consistency model is leveraged by ONOS controller applications that require strong consistency and correctness guarantees, the eventual consistency model is intended for ONOS controller applications that favor scalability and performance over strict consistency.

In this paper, we target the second class of scalable control applications that have optimistic relaxed consistency needs, but that can benefit from improved performance and automated SLA-aware consistency tuning at scale, as offered by our adaptive continuous consistency strategy.

A. A continuous consistency model for ONOS

As explained in [9], the applications on top of ONOS can benefit from the continuous consistency model introduced with TACT [18], by continuously and dynamically specifying their consistency requirements using three application-independent metrics to capture the consistency spectrum and bound consistency: *Numerical Error*, *Order Error*, and *Staleness*.

In this work, we focus on the type of applications whose application-specific consistency semantics can be expressed using the staleness of data as a metric to quantify the level of consistency. With such SLA-style consistency metrics, these applications can prevent the challenges related to potentially *unbounded staleness* as in eventual consistency.

Generally speaking, the staleness metric measures data freshness in distributed data-stores; it describes how far a given replica lags behind in data operations in comparison to up-to-date replicas, either expressed in terms of time or versions. In the literature, the notion of data staleness falls indeed into two common categories: staleness in time (*time-based staleness*) [18, 19], and staleness in data version (*version-based staleness*) [19].

In TACT [18], the staleness metric places a real-time bound on the amount of time before a replica is guaranteed to see a write accepted by a remote replica. In [20], the authors propose a probabilistic consistency framework that provides expected bounds on data staleness with respect to both versions and wall clock time in eventually-consistent data-stores. In their model, time-based staleness (*t_visibility*) describes the probability that a read operation, starting t seconds after a write commits, will observe the latest value of a data item [19]. On the other hand, version-based staleness (*k_staleness*) describes how many versions the value returned by a read lags behind the most recent write. It is measured as the probability of returning a value within a bounded number k of versions.

In this work, we adopt the data staleness metric from a strictly time-based perspective. In our SDN controller application, we characterize staleness by an "Age of Information (AoI)" timeliness metric [27] that describes the difference between the query time of a data item and the last update time on that item. If the last successfully received update was generated at time $u(t)$ then its age at time t is $\Delta(t) = t - u(t)$.

Applications on top of the distributed ONOS controllers could also benefit from SLA-style performance requirements, to continuously specify their own fine-grained trade-offs between performance and consistency. In our work, we consider the read request latency/delay as our performance metric. In addition, we evaluate the inter-controller communication overhead for our ONOS application.

More detailed information about the way we measure our continuous consistency and performance metrics when implementing our state consistency approach for the new controller application that we designed on top of ONOS is provided in Section V-A.

B. Our Quorum-inspired consistency adaptation strategy for ONOS

1) Quorum consistency:

As explained in Section II-C, quorum-replicated systems ensure different consistency guarantees:

- Strong consistency can be guaranteed with *strict quorums* that satisfy the condition that sets of replicas written to and read from need to overlap:
  $R+W > N$, given N replicas and read and write quorum sizes R and W.
- Eventual consistency occurs with *partial quorums* that fulfill the condition that sets of replicas written to and read from need not overlap:

$R+W \leq N$, given N replicas and read and write quorum sizes R and W.

Traditionally, partial quorum-replicated systems ensure eventually-consistent guarantees, with no limit to the inconsistency of the data returned, which may not be acceptable for certain applications. However, with the PBS model [19], it has been possible for applications to analyze the staleness of the data returned, quantify the consistency level, and therefore measure and control the trade-offs between latency and consistency for partial quorum systems.

Building on these concepts, we propose an adaptive consistency model for the ONOS applications using *partial quorums*, given the latency and scalability benefits they offer. Indeed, in the context of large-scale networks and IoT-based environments, most applications opt for relaxed consistency control models like eventual consistency which can be guaranteed with *partial quorums* in order to fulfill their requirements in terms of availability and performance at the large scale. This is in contrast with strong consistency models which, being ensured with *strict quorums*, are extremely costly to maintain for such scalable applications due to their important synchronization overheads among the controller replicas in the cluster.

On the other hand, to measure the consistency semantics (e.g the staleness metric) of the ONOS applications and thus meet their consistency requirements (e.g bounded staleness), we leverage the continuous consistency model discussed in Section III-A.

Furthermore, using eventually-consistent partial quorums, it is possible to configure the size of read and write quorums, denoted respectively as R and W such that $R + W \leq N$, to ensure various consistency levels (e.g. degrees of staleness). These multiple quorum configurations allow the applications to achieve different trade-offs between consistency and latency.

2) Adaptive architecture:
In this work, we propose to turn the eventual consistency model into an adaptive and continuous tunable consistency model using partial quorums [17]. The proposed model uses the quorum replication parameters as a control knob, allowing for an adaptive fine-grained tuning and control over the trade-offs between consistency and performance. In the following, we describe the main architecture components of our adaptive consistency model.

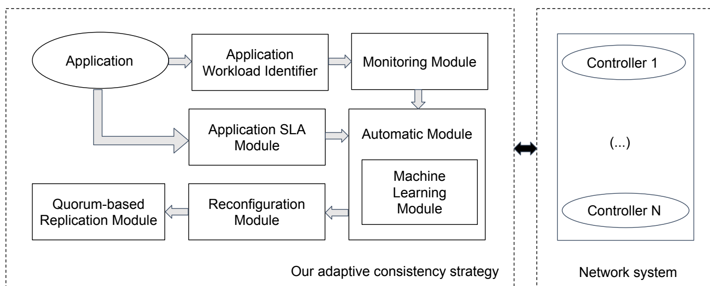


Fig. 1: Architectural overview of our adaptive Quorum-based consistency strategy

- `Automatic Module`
  The choice of the size of read and write Quorums used when executing read and write operations is a fundamental factor that affects the application's consistency guarantees but also the performance provided by the network system. However, selecting the appropriate Quorum configuration is a non-trivial task. Our Automatic Module attempts to find the optimal configuration of the read and write Quorum sizes while taking into account the current application workload conditions. The main objective is to minimize the overhead generated by the application (the scalability challenge), and potentially other network and application metrics, while satisfying the consistency and performance SLAs specified by the application.
  This module is fed with a set of application workload characteristics which are gathered by the Workload Identifier Module. In our case, it relies on a Machine Learning Module to predict the expected optimal configuration of the Quorum parameters for the determined workload, and then feed them to the Reconfiguration Module.
- `Machine Learning Module`
  This module uses Reinforcement Learning (RL), an area of Machine Learning (ML) inspired by behaviorist psychology, and concerned with how software agents take actions in an environment so as to maximize some notion of cumulative reward.
  More specifically, we use a Q-Learning (QL) model-free RL technique [28]. The main idea is to train an *agent* which interacts with its *environment* by performing *actions* that change the environment, going from one *state* to another. These actions result in a *reward* received by the agent as an evaluation of its actions (reinforcement) (see Figure 2). In this way, the agent learns some rules and develops a strategy, referred to as a *policy*, for choosing actions that maximize its reward.

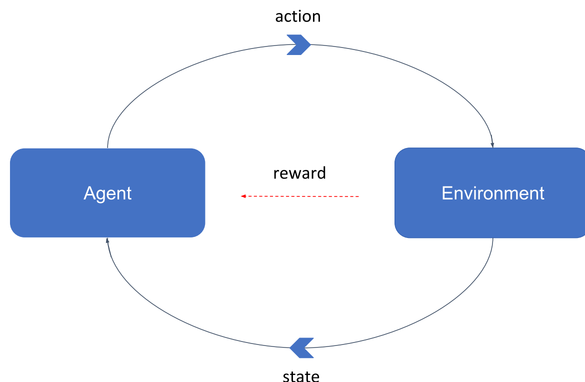


Fig. 2: Reinforcement Learning (RL) architecture

The Q-Learning update rule makes use of the so-called Action-Value function, commonly known as the *Q-function*, representing the "quality" of a certain action in a given state. The expression of the Q-function is given by the following Bellman equation [28]:

$$Q(s_t, a_t) \leftarrow (1-\alpha) \times \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \times$$

$$\left[\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \times \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} \right] \quad (1)$$

The above Q-function is used to update the *Q-table* with Q-values at each *episode*. A Q-value is assigned to a possible pair of a state s_t and a current action a_t. The Q-function takes as input the pair (s_t,a_t), observes a new state s_{t+1} and returns the expected rewards of that action at that state. More specifically, the Q-function maps state-action pairs to the highest combination of the immediate reward r_t for that action with all discounted (using γ) future rewards that might be collected by later actions. The future rewards are computed using the maximum value of Q, given by $\max_a Q(s_{t+1}, a)$, for all possible actions in the next state, assuming that the agent continues to follow the optimal policy. We also note that the discount factor $\gamma \in [0; 1[$ determines the importance of future rewards with respect to immediate/current rewards, whereas the learning rate $\alpha \in]0; 1]$ determines to what extent newly acquired information (during the learning process) overrides the previous old information.

We also note that the learning agent should achieve a good strategy for balancing the trade-off between exploration and exploitation, which is inherent to reinforcement learning. That dilemma consists in choosing the appropriate action at a given episode: either to *exploit* the environment by selecting the best action at that specific time step given the current knowledge provided by the Q-table, or to *explore* the environment by choosing random actions. After each action, the agent is expected to update the Q-table.

In our case, the Q-learning agent attempts to learn online the best combination of the read and write Quorum size parameters, respectively R and W, in an environment built using our Monitoring Module. An action is defined as an update of R and W to certain possible values, thereby transforming the environment to a state defined by a new estimation of the network (inter-controller overhead) and application (latency and staleness) metrics. In our case, one of four possible actions is allowed at each episode (i.e. incrementing R by one, or decrementing R by one, or incrementing W by one, or decrementing W by one).

The reward received by the agent for updating the Quorum parameter values is a function of the read and write overheads to be minimized. The agent should also learn how to respect some constraints in order to satisfy the application requirements specified in the given SLA.

- `Reconfiguration Module`
  This module is able to dynamically adjust the values of the read and write Quorum sizes, denoted respectively as R and W. It basically relies on the Automatic Module to optimize the configuration of the quorum system. The reconfiguration process launched by this module is a non-blocking process that is able to re-configure at run-time the Quorum settings selected by the Automatic Module. A more detailed description of the way the reconfiguration module sets the values of R and W at run-time is provided in Section V-B1.
- `Quorum-based Replication Module`
  Given the quorum replication settings, we adopt the following consistency strategy when reviewing the two main techniques employed by ONOS's eventual consistency model:
  - Replication Strategy: As explained in [9], ONOS's eventually-consistent stores employ an optimistic replication technique that consists in replicating local updates across all the controllers in the cluster, hence causing control plane overhead. Instead, we put forward a partial quorum replication strategy, where an eventually-consistent data store writes a data item on the local replica first and then sends it potentially to another set of replicas, obeying the given write quorum parameter (W). On the other hand, to serve read requests, we propose that the eventually-consistent data store fetches the data from the local replica first and then potentially from another set of replicas, depending on the given read quorum (R). This is in contrast to ONOS's strategy where the read requests are always processed by the local replica.
  - Anti-Entropy reconciliation mechanism: As explained in [9], ONOS's optimistic replication strategy is complemented by a background Anti-Entropy mechanism. That periodic reconciliation approach ensures that the system state across all replicas eventually converges to the consistent state. This is particularly useful in repairing out-of-date replicas and fixing state inconsistencies potentially resulting from controller failures. In this work, we assume that the system is reliable as we experiment with well-functioning emulated network topologies in the absence of controller failure scenarios. Therefore, we propose to deactivate the Anti-Entropy protocol, and focus on ONOS's replication strategy. However, it is worth noting that using additional Anti-Entropy (*expanding partial quorums* [19]) might be useful in particular cases where state inconsistencies become high and can no longer tolerated by the concerned applications.
- `Application SLA Module`
  This module offers the possibility for applications on top of ONOS to express their high-level SLA-style consistency and performance requirements such as the staleness and latency guarantees. Accordingly, for a given ONOS application that we develop on top of ONOS, our consistency model continuously measures the real-time metrics involved in quantifying the trade-off between consistency and latency. The Automatic Module translates

these requirements into appropriate time-varying partial quorum replication configurations (R, W, N) that achieve balanced trade-offs between the specified guarantees.

- `Workload Identifier Module`
  This module identifies the application's workload characteristics. It considers three different workloads that are representative of three different application scenarios [29]. The first workload has a balanced ratio between read and write operations. The second workload represents a write-dominated scenario in which 70% of the generated operations are write accesses. Finally, the third workload describes a read-intensive scenario where 70% of operations are read accesses.
- `Monitoring Module`
  This module is responsible for periodically gathering the application traffic information in a non-intrusive manner. More specifically, the module measures the system Key Performance Indicators (KPIs), for different read and write Quorum configurations and according to different application workload scenarios. These KPIs include the performance (e.g. response time) and consistency (e.g staleness) metrics related to client requests for specific application contents, as well as the generated read and write application overheads. These measurements are used by our Automatic Module (more particularly the Machine Learning Module) to learn online the appropriate Quorum configurations.

IV. Implementation approach on ONOS

In this section, we describe the implementation details for realizing the proposed consistency strategy on the Java-based open-source ONOS controller platform. That strategy was explained in detail in the previous section and summarized in Figure 3.

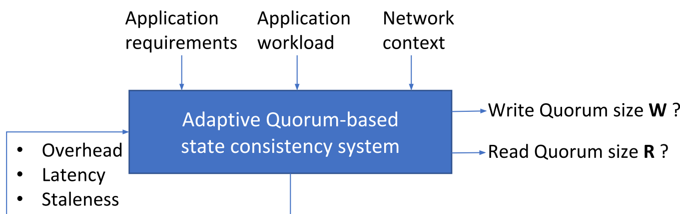


Fig. 3: The proposed adaptive consistency system

A. Design of a C-CDN-like application PoC

To validate our adaptive consistency approach, we developed a new distributed Content-Centric Delivery Networks (C-CDN) application running on top of a cluster of multiple ONOS controllers in an emulated SDN network. Our application replicates contents from content providers to hosting cache servers that are located in multiple geographical locations (ONOS domains) close to users. These cache servers are Mininet hosts that run simple HTTP web servers. We propose to consider a single origin server located in each ONOS domain. The main idea is to serve client hosts with the most up-to-date copies of the requested content and within a reasonable time (low latency).

More specifically, our application consists of two main components: An `ApplicationManager` and a `DistributedApplicationStore`. The Application Manager component which is an implementation of the Application Service is responsible for creating a virtual network of cache servers and providing mesh connectivity between these server hosts. On the other hand, the Distributed Application Store which is an implementation of the Application Store performs the task of persisting and synchronizing the information received by the application manager. It is backed by an eventually consistent map with eventual consistency guarantees for storing the service's application state, namely the list of origin servers in the network and their respective set of generated contents: `EventuallyConsistentMap <OriginServerID, Set<Content>>`.

Each controller replica that is responsible for a given ONOS domain operates on a local view of the eventually consistent map. That view consists of the local origin server from the same ONOS domain with its generated set of contents, and other potential origin server hosts located in different ONOS domains in the network with their respective set of contents, as seen by the local replica after application state synchronization.

Besides, we design a cached map that is local to each controller application instance and that represents the contents cached in the local C-CDN server within the same ONOS domain. The local cached map is closely linked to the local view of the eventually consistent map, and it reflects the contents stored in the local C-CDN server. The latter performs the functions of an origin server and at the same time a cache server. It contains indeed the contents created locally (the origin server), and potentially other contents that are replicated from other origin servers (the cache server).

More specifically, on a local controller replica, updates to the eventually consistent state map (e.g PUT) might trigger specific actions to feed the local C-CDN server and consequently update the local cached map. If the update to the content is associated in the map with the local origin server, that means that the updated content has already been generated on that origin server. On the other hand, if the update to the content is associated in the map with another origin server from another ONOS domain, our application checks the relevance of that content. In case the content is important to our application, then the update to the content gets automatically pulled from the origin server to the local C-CDN server (cache server) and gets cached in the local cached map `CachedMap <ContentName, Set<Content>>`.

B. State synchronization and content distribution

The custom eventually consistent map we use for the synchronization of our C-CDN application state is based on our own implementation of the `EventuallyConsistentMap <K,V>` distributed primitive. Indeed, the new implementation we propose for the eventual consistency map abstraction models the quorum-inspired consistency discussed in III-B1.

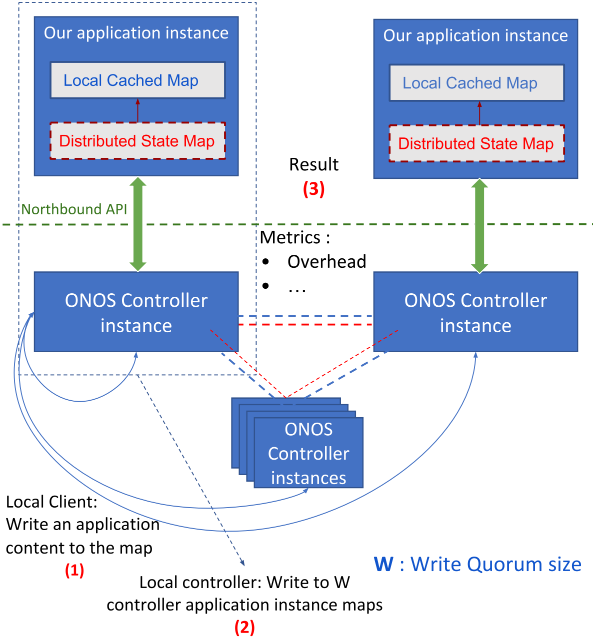


Fig. 4: Quorum-inspired Write operations in our C-CDN-like application

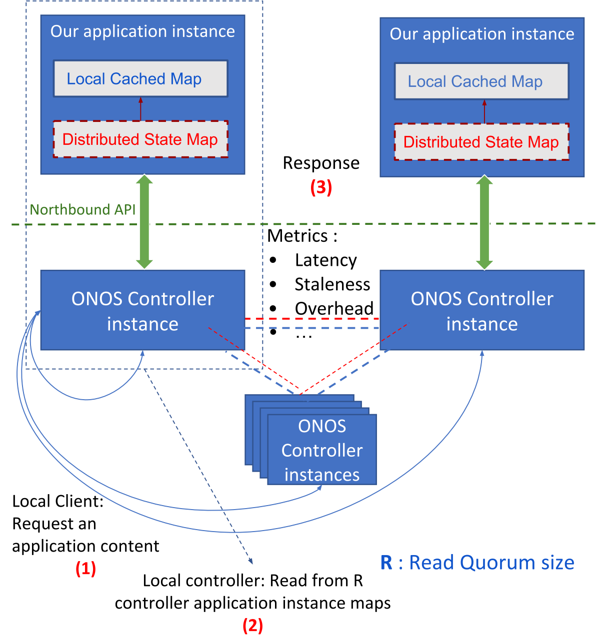


Fig. 5: Quorum-inspired Read operations in our C-CDN-like application

In particular, it takes into account the size of the write quorum parameter (W) when replicating the updates related to our application's eventually consistent map among the controllers (see Figure 4). On each local replica, updates to the local map are queued in time to different `EventAccumulators` allocated for different controller peers. The latter are selected randomly, and their number depends on the write quorum size W. Whenever an event accumulator is triggered to process the previously accumulated events and propagate them to the associated peer, that peer is removed from the list of quorum peers. New updates will immediately trigger the creation of a new accumulator associated with a new randomly selected peer that is is added to the list of quorum peers. That accumulator will collect the updates together with the other event accumulators associated with the rest of the quorum peers. That way, we guarantee that updates to the eventually consistent map on a local replica are replicated at run-time to exactly W replicas, including the local replica.

As explained in Section IV-A, such updates to the eventually consistent map on a local controller replica trigger specific actions that might feed the local C-CDN server with new contents (content distribution) and thus update the local cached map for our application.

C. Content delivery to customers

During a read operation performed by a client, our controller application instance running on the local controller replica within the same ONOS domain as that client, receives the read request to be fulfilled following Quorum-inspired read consistency protocols (see Figure 5).

More specifically, if the read consistency level is higher than ONE (read quorum size R greater than 1), then the local controller node which serves in our case as the coordinator node, sends the read request to the remaining randomly-selected controller replicas forming the read Quorum. The size R of the read Quorum including the local controller replica is set in advance by the read consistency level.

We use ONOS's `ClusterCommunicationService` to assist communications between the local controller node and the rest of the controller cluster nodes in the read Quorum. More specifically, the local controller node sends the read request message with a particular subject to each of the concerned controller nodes using the `sendAndReceive` method of the cluster communication service. It expects a future reply message from each of the involved controllers that have already subscribed to the same message subject.

That said, to serve the client's read request for a specific content (`ContentName`), each controller node that has subscribed to the specified message subject receives the read request and uses the application's handler function for processing the incoming message. Accordingly, the application instance on each controller replica of the read Quorum (including the local replica) consults the local cached map. As explained in Section IV-A, the cached map represents the list of contents (created by different origin servers) being observed in the local view of the eventually consistent map, and then pulled to be cached in the local C-CDN server. Using that map, each application instance compares the cached versions of the requested content (`ContentName`) based on their `LogicalTimestamp` properties in order to determine the freshest version of the content. Then, it produces a reply containing the selected `Content` with its four properties discussed in IV-A, and more importantly the IP address of the local cache server that has just delivered the requested

content.

Each content that is created on the origin server, and then eventually propagated to cache servers has four properties; a `ContentName`, an identifier `ID`, a real time-based `CreationTime`, a `LogicalTimestamp`, and a `Version`.

The local controller replica playing the role of the coordinator, waits for the read Quorum of replicas to respond. Then, it merges the R responses (including the response produced on the local replica) to figure out the location of the freshest version of the requested content among the concerned C-CDN servers (equal to R in our scenario). Finally, it sends back the final response to the client and makes sure a host-to-host connectivity intent is added between the client host and the determined cache server host, using the `ONOS Intent Framework`. Based on that response, the client which has issued a HTTP request specifying the URL of the requested content, is redirected, using our C-CDN-like strategy (described above) and a DNS resolution service, to the selected cache server in order to retrieve the specified version of the content.

After each client request, our application collects the continuous consistency and performance metrics related to that request. These metrics are described in detail in Section V-A.

V. Performance evaluation

A. Application-specific performance and consistency metrics

Here, we show the considered continuous and SLA-style performance and consistency metrics. More specifically, we show how we measure these metrics when implementing our adaptive consistency strategy for the C-CDN application that we designed on top of the distributed ONOS controllers.

- Performance metrics:
  - Network-related metrics:
    We consider the application inter-controller overhead as a network performance metric. We first capture all inter-controller traffic using TCP port 9876. Then, we filter the captured traffic based on different conditions in order to evaluate the application's inter-controller overhead.
    Our goal is to minimize the application overhead due to write and read operations, depending on the given application SLA, the application workload and the network context.

$$AppOverhead = WriteOverhead + ReadOverhead \quad (2)$$

  - Client-centric metrics:
    We also consider the response time to a client request as a performance metric. As defined by our application, the response time consists of the delay to fetch the appropriate version of the requested content from the local cached maps of the application instances running on the R controller replicas of the read Quorum (`Latency1`), and the delay to retrieve the specified version of the content from the selected cache server host (`Latency2`). We also note that these latency times do not overlap.

$$ResponseTime = Latency1 + Latency2 \quad (3)$$

- Consistency metrics:
  As explained in Section III-A, we consider the application-specific staleness metric from a strictly time-based perspective: It describes the age of the information in terms of wall-clock time. Accordingly, the staleness of the application content `C` being returned by a read operation at time t is measured as follows:

$$Staleness(C) = QueryTime - CreationTime(C) \quad (4)$$

  Besides, we set the staleness ranges used in the consistency SLA based on the application content refresh rate.

B. Experimental setup

Our experiments are performed on an Ubuntu 18.04 LTS server using ONOS 1.13. We also use Mininet 2.2.1 and an ONOS-provided script (*onos.py*) to start an emulated ONOS network on a single development machine; including a logically-centralized ONOS cluster, a modeled control network and a data network. Wireshark is used as a sniffer to capture the inter-controller traffic which uses TCP port 9876.

1) TCL-Expect scripts:
In this section, we test our proposed adaptive consistency approach explained in Sections III and IV which we will subsequently refer to as ONOS-WAQIC (ONOS-With Adaptive Quorum-Inspired Consistency) for brevity. The proposed approach was implemented for our C-CDN application on ONOS-WAQIC.

To that end, we write two `Expect` Tcl-based scripts (`main.exp` and `onos.exp`). In each script, we specify a set of required steps to follow to automate the tasks for our test scenarios on ONOS-WAQIC as summarized below:

- First, we run our startup `Expect` script (`main.exp`). With Mininet and *onos.py*, we start up an ONOS cluster and a modeled data network for the specified topology. The selected number N of the ONOS controller replicas that will be forming the ONOS cluster is passed as an argument to the executed script.
- Then, we run the Mininet CLI built-in `pingall` command to discover the network topology. We also launch a spawned process to install and activate the C-CDN-like application we developed on ONOS-WAQIC. To force device/switch mastership re-balancing, we connect to one of the running ONOS controller instances, and launch the ONOS CLI `balance-masters` command.
- First, we parse the output of the `dump` Mininet command using regular expressions in Tcl in order to build a key-value array mapping the IP addresses of hosts to their Mininet names (*array1)*. Then, in the main Expect script, we launch N spawned processes that connect to the N running ONOS controller instances using the

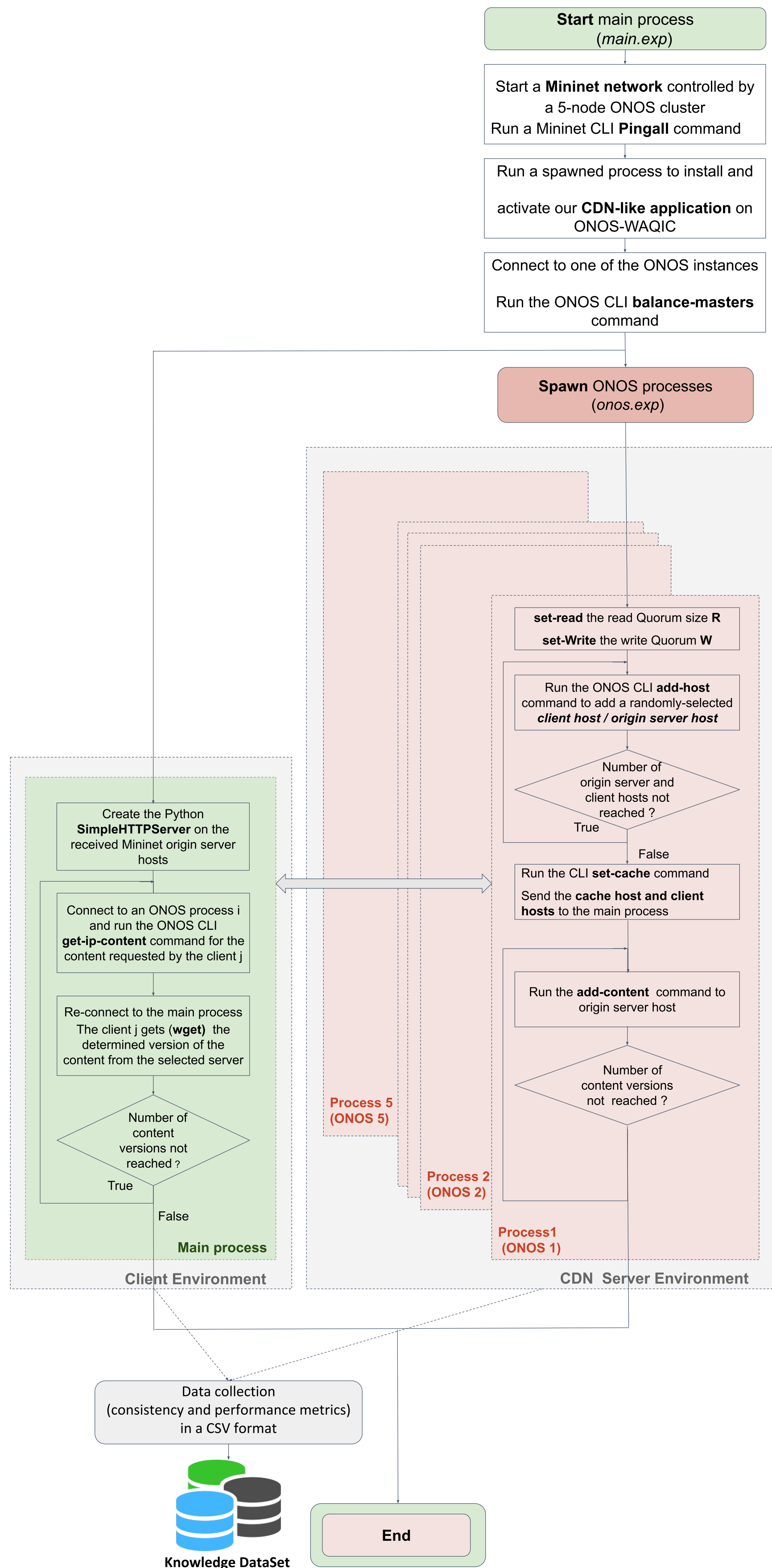


Fig. 6: Overview of the main tasks executed by our TCL-Expect scripts

same `Expect` script (`onos.exp`) we developed, but run with different arguments (controller IP address, content name, maximum number of content versions). In the `onos.exp` script, we analyze the output of the `masters` ONOS CLI command to construct an array mapping each controller IP with the set of associated switches (MAC IDs) (*array2*). In addition, using the output of the Mininet CLI `hosts` command, we construct two additional arrays: the first array associates each host MAC ID with its IP address (*array3*), and the second array associates each host MAC ID with the switch ID to which it is connected (*array4*).

- It is worth noting that our `onos.exp` script starts by running two ONOS CLI commands (`set-read` and `set-write`). We created these commands to set the read and write Quorum sizes R and W to the values specified by the consistency level for a given ONOS controller instance. These values are passed as command arguments.
- Using *array2*, *array3* and *array4*, each of the N currently spawned processes running the `onos.exp` script for a specific ONOS instance builds another Tcl array (*array5*) that identifies the list of hosts (MAC addresses) associated with each ONOS controller instance (controller IP address) in the network. Based on that array, our script randomly selects, for the specified ONOS controller instance, a list of hosts that will serve as origin cache servers and a list of hosts that will serve as clients in the concerned ONOS controller domain. The number of selected cache and client hosts depends on the application scenario/workload (see Section III-B2). Each ONOS process communicates the MAC and IP addresses of the origin server to the local application instance using our ONOS CLI `set-cache` command. Our script also runs the ONOS CLI `add-host` command which we created to add the cache server hosts to our application's `EventuallyConsistentMap` (discussed in Section IV-A).
  Besides, information about these cache server hosts is sent (using "puts") to the running `main.exp` script process. The latter identifies the Mininet names of these hosts using *array1* and connects to the Mininet CLI command in order to install a `SimpleHTTPServer` on each of the cache server hosts.
- At this stage, we make sure that our main process (running *main.exp*) and the N spawned processes (running `onos.exp` with different arguments) are synchronized. Afterwards, each of the N spawned processes connecting to an ONOS controller instance starts adding (then updating with a certain *refresh rate*) the contents to the origin server host in the involved ONOS domain. We use the `add-content` ONOS CLI command that we created to add a given content version (second command argument) to the specified origin server host (first command argument) in the application's `EventuallyConsistentMap`. Further details about content distribution and state synchronization using Quorum-inspired write consistency are provided in Section IV-B .
  On the other hand, in parallel with the updating of contents, our main process that is handled by the `main.exp` script starts issuing and serving client requests for specific contents. That was achieved using our `get-IP-content` CLI command which takes one argument, namely the requested `ContentName`, and returns the IP address of the cache server containing the freshest/selected version of the requested content, Then, our script retrieves the content from the determined server using "wget". In addition, after each client request, continuous application-specific consistency and performance metrics related to that request are collected with our script using regular expressions in Tcl. More details about the content delivery strategy we follow using Quorum-inspired read consistency are given in Section IV-C.

2) OpenAI Gym simulator:
To implement the Machine Learning Module (see Section III-B2) for our C-CDN-like application on ONOS-WAQIC, we build a simulator based on OpenAI Gym [30], an open-source Python toolkit for developing and comparing reinforcement learning algorithms.

More specifically, we build a new environment to simulate knowledge exchange in an ONOS SDN cluster: We start by building an off-line data-set using our TCL-Expect scripts explained in Section V-B1. Our data-set stores the information collected by the Monitoring Module about client requests for specific C-CDN contents. As detailed in Section IV-C, for a given client request, the returned information contains the current values of the Quorum parameters R and W, the expected returned version of the content (content update step), the actual returned version of the content, *the staleness* of the returned content, *the delay* incurred in searching for the freshest version of the content from R controller replicas (latency1), *the read overhead*, *the write overhead*, and the application scenario determined by the Workload Identifier Module.

The data-set is fed to the Automatic Module which hands it over to the Machine Learning Module to learn online the read and write Quorum size parameters. Implemented with Gym, the latter module uses the data-set to learn the Kernel Density Estimation (KDE with scipy) for each metric using the data of some clients. That client data is selected with respect to the current configuration of R and W parameters. That configuration was set following an action performed by the agent (see the explanation of the Q-learning algorithm in Section III-B2 for more details). That way, using KDE, our ML Module estimates the expected metrics for each selected Quorum configuration, and then updates the Q-table with the Q-value of that action at that state, at each step (or episode) of the Q-learning algorithm.

3) Various learning agent policies:
We implemented three learning agents that adopt different policies. The latter are compared and validated through five scenarios. Each scenario reflects a specific use case (e.g. a latency-sensitive application, a consistency-favoring application). To minimize the application's overall inter-controller overheads, our agents use the estimated overhead as a negative

”reward” when performing actions (setting R and W) that change the environment state. The controlled and constrained agents are proposed with the aim to improve the simple greedy agent. Below is a brief description of these agents:

- *A simple ϵ-greedy agent* [31]: This agent follows a simple ϵ-greedy policy with a fixed ϵ value, where ϵ is the exploration rate and (1-ϵ) is the exploitation rate. We test three simple ϵ-agents: ϵ-greedy5 (ϵ=0.5), ϵ-greedy10 (ϵ=0.10) and ϵ-greedy15 (ϵ=0.15).
- *A controlled dynamic ϵ-greedy agent*: This agent follows a dynamic ϵ-greedy strategy where the exploration rate ϵ decays as the algorithm's episode count increases. The purpose is to account for the fact that the agent learns more about the environment in time, and becomes progressively more confident and ”greedy” for exploitation. We use the following decay function for reducing ϵ as a function of episode count. x is the episode number.

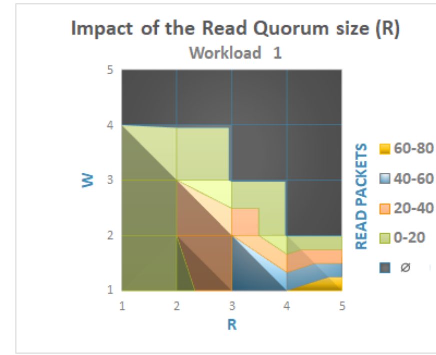
$$f(x) = \epsilon * (0.5 + \log_{10}(2 - arctan(\frac{x}{10} - 2))) \tag{5}$$

  To attempt to satisfy the application's latency and staleness thresholds, the simple and controlled agents reject, at each exploitation episode, any action violating these constraints and remove its Q-value from the Q-table.
- *A constrained ϵ-greedy agent*: To make the agent learn how to satisfy the application's SLA, we create a Q-constraint list that we update over the episodes. Its size corresponds to the number of potential actions: the number of R and W combinations such that $R+W \leq N$. The list represents the number of constraint violations by each Quorum configuration. The considered constraints are both the latency and staleness thresholds specified in the SLA. During each exploitation phase, we update the Q-constraint list, and use it to generate a new Q-list containing the Quorum configurations that give less constraint violations. These configurations are then exploited: They are compared using their Q-values in the Q-table (based on the estimated overhead reward) to select the best Quorum configuration (action) at that episode.

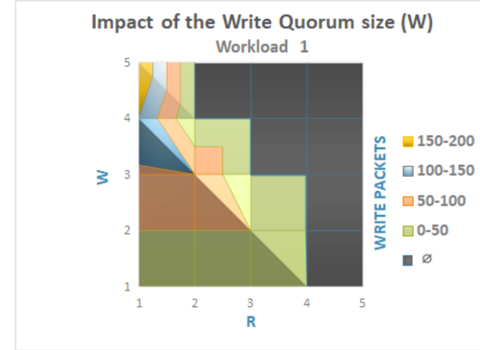
C. Results

1) Impact of the Read and Write Quorum sizes: In this section, we present an experimental study that is aimed at assessing the impact of using different read and write Quorum sizes (R and W respectively) on the read and write inter-controller overheads of our C-CDN-like application running on a 5-node ONOS cluster in the network topology.

In the conducted experiments, we consider three application workloads that are representative of three application scenarios (see the `Workload Identifier Module` in Section III-B2 for more details). For the studied workloads, we show the captured read and write packets within a specified time interval (i.e. 400 ms in our tests) of read and write client operation accesses, for all possible eventually-consistent partial quorum configurations (R,W) (e.g. (R,W) combinations such that $R+W \leq N$ where $N=5$).

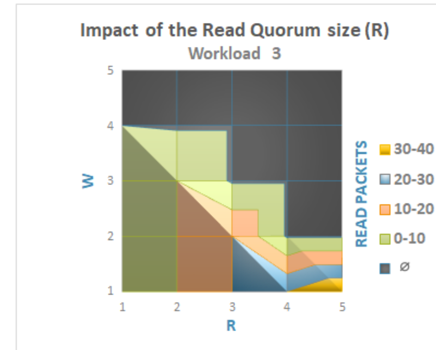


(a) Read packets when varying (R,W)

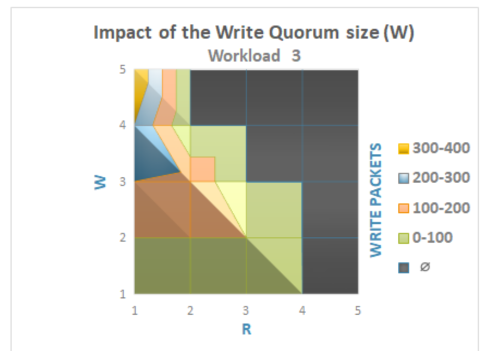


(b) Write packets when varying (R,W)

Fig. 7: Workload 1: A Read-intensive application scenario



(a) Read packets when varying (R,W)



(b) Write packets when varying (R,W)

Fig. 8: Workload 3: A Write-intensive application scenario

Our results clearly show that, when increasing the read Quorum size R, the number of read packets increases, mainly in a read-dominated workload (see Figure 7(a)). In addition, increasing the write Quorum size W results in a drastic increase of the number of write packets, especially in a write-intensive workload. Their number reaches indeed 400 during the specified time interval for a partial Quorum configuration where W is equal to 5 (see Figure 8(b)).

Given the high inter-controller overheads observed in our experimental data for certain Quorum configurations, we propose to tune the R and W parameters and therefore optimize the configuration of the Quorum system to better match the varying application SLA requirements and the dynamic application workloads, as we further discuss in the following sections.

2) Quorum configuration optimization:

a) Dynamic application SLA requirements: To evaluate our ONOS-WAQIC proposal for the C-CDN-like application we developed, we run our TCL-Expect scripts (see Section V-B1) with a 5-node ONOS cluster according to different scenarios. In these scenarios, we use different partial Quorum configurations (R, W), and we follow various application workloads with respect to different ratios between read and write operations. Then, we use the data collected as an input to our Q-Learning simulator (see Section V-B2). In the simulator environment, we set $\alpha = \gamma = 0.5$ and the number of episodes to 1000. We also consider different test scenarios that reflect different application requirements in terms of performance and consistency as summarized in table I.

In particular, using our data-set and knowing the refresh rate of our C-CDN-like application, we learn the t_staleness ranges. In other words, we learn the relationship between the t_staleness value of a certain content being returned and by how many versions that returned content is old. As a result, estimating the t_staleness ranges allowed us to set the time-based staleness thresholds in the SLA while having an idea about the associated version-based staleness thresholds.

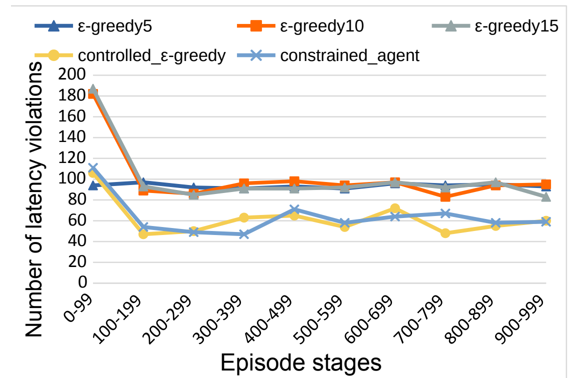
Test scenarios	Latency threshold (ms)	t_Staleness threshold (ms)	k_Staleness Version old
n°1	5	300000	3
n°2	25	220000	2
n°3	50	120000	1

TABLE I: Application SLA scenarios

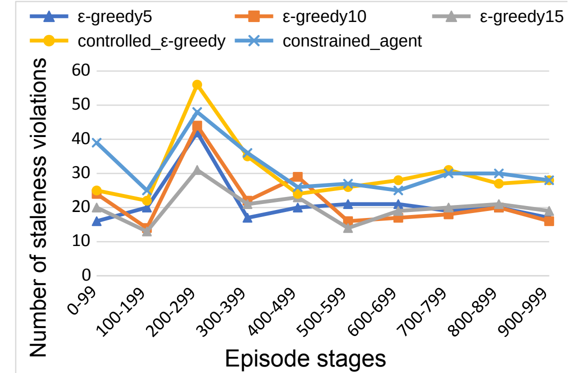
In each test scenario that we run on the simulator, our application expresses the performance and consistency SLAs using the latency threshold (in ms) and the staleness threshold (in ms). For example, in scenario n°3, our application which is consistency-favoring enforces the following SLA: It expects that a read operation gets a reply in under 100ms, and returns a content value no older than 120seconds (i.e. no older than 1 version stale). Accordingly, our consistency approach attempts to find the best Quorum combination of R and W that minimizes the application's read and/or the write inter-controller overheads while ensuring the desired performance-consistency trade-offs.

For a given Quorum configuration, we compute the read overhead ratio by normalizing the generated read overhead (bytes/s) with respect to the Quorum configuration generating the maximum read overhead and zero write overhead (the configuration ($R = 5$,$W = 1$)) in our case) for each application scenario. We follow the same steps for computing the write overhead ratio based on the generated write overhead with respect to the Quorum configuration ($R = 1$,$W = 5$) which corresponds to the standard implementation of ONOS's eventual consistency model. On the other hand, whenever we aim to minimize both the read and write overheads (e.g. in a balanced workload scenario), we consider the mean of the read and write overhead ratios which we will subsequently refer to as the global overhead ratio.

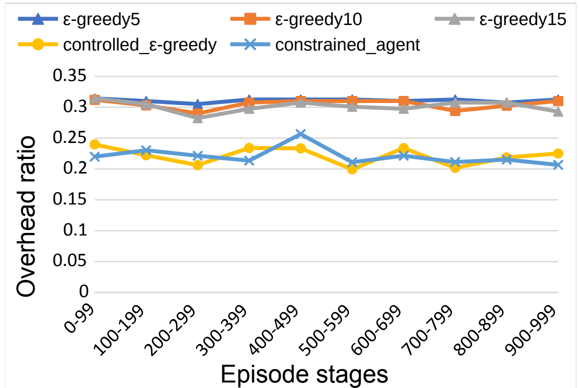
In Figures 9, 10 and 11, we show the results of our experimental tests for the three considered application sce-



(a) Latency violations



(b) Staleness violations



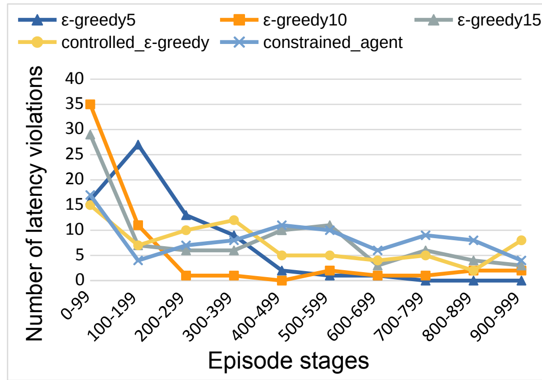
(c) Overhead ratio

Fig. 9: Scenario 1: A Latency-sensitive application

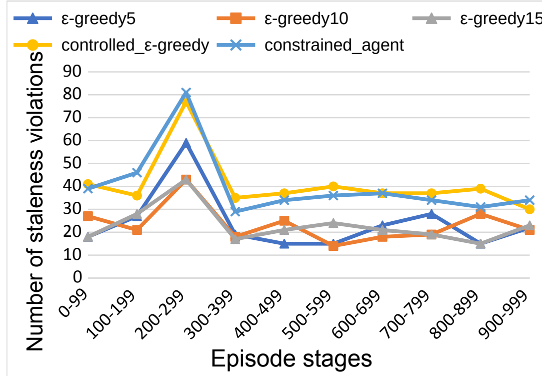
narios. To study the impact of changing the application SLA requirements, we set the application workload to Workload 2 (a balanced workload scenario that has a balanced ratio between the read and write operation accesses) in which our consistency approach attempts to minimize the global overhead ratio, and satisfy the staleness and latency SLA thresholds set by the application. Moving from one application workload scenario to another (e.g. a read-intensive scenario) will be dealt with in the following section.

Figure 9 shows that, in a latency-sensitive application scenario, the constrained and the controlled agent policies are the most appropriate. The number of constraint violations decreases with episode stages (see Figures 9(a) and 9(b)), and the generated global (read and write) inter-controller overhead (see Figure 9(c)) is minimal as compared to the simple greedy agent policy, and to the standard ONOS implementation. We also notice that the three agents converge towards Quorum configurations where $R = 1$ (i.e. $(R = 1, W = 2)$, $(R = 1, W = 3)$ and $(R = 1, W = 4)$). This is due to the given strong constraint on latency.

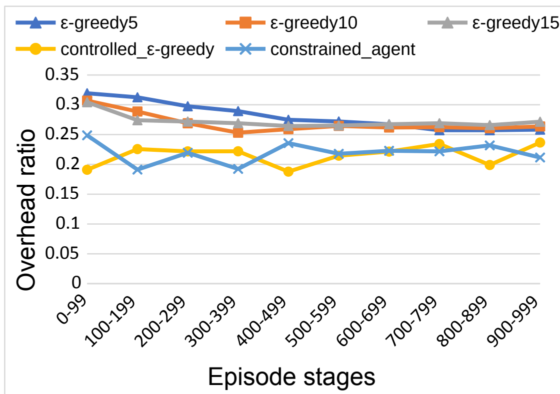
Figure 10 shows that, in a balanced application scenario, the constrained and the controlled agent policies offer the best real-time trade-offs between the application's latency and staleness needs (see Figures 10(a) and 10(b)) while ensuring minimal global overhead ratio (approximately 25%) (see Figure 10(c)). In particular, the constrained agent converges



(a) Latency violations

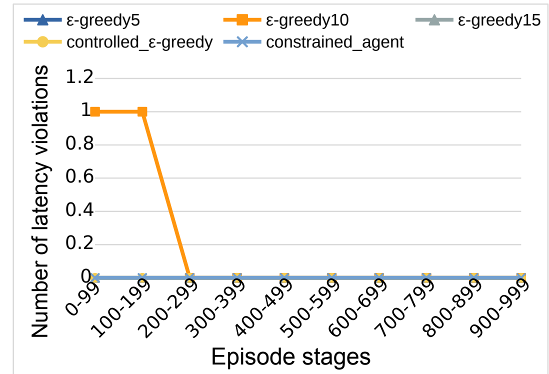


(b) Staleness violations

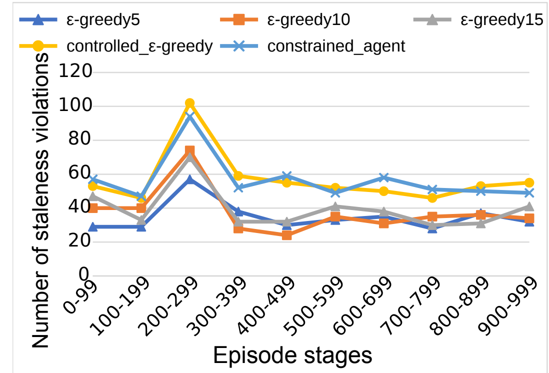


(c) Overhead ratio

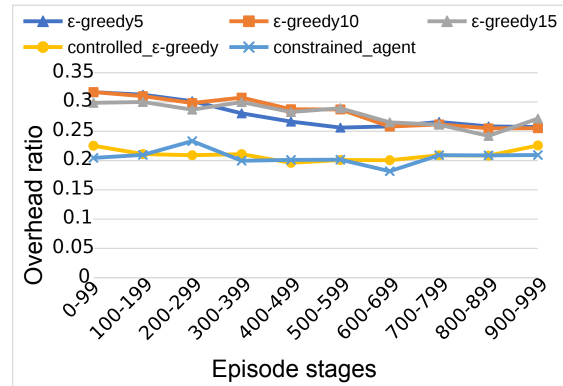
Fig. 10: Scenario 2: A Consistency/Latency-balancing application



(a) Latency violations



(b) Staleness violations



(c) Overhead ratio

Fig. 11: Scenario 3: A Consistency-favoring application

towards balanced Quorum configurations (i.e. $(R = 2, W = 2)$ and $(R = 2, W = 3)$). On the other hand, the simple ϵ-greedy agents provide a small number of latency violations, but at the cost of generating more overhead.

As can be seen from Figure 11, in a consistency-favoring application scenario, all agents perform well at reducing the staleness violations (see Figure 11(b)), especially the simple greedy agents. Besides, all agents respect the relaxed latency constraint (see Figure 11(a). They all converge towards a common Quorum configuration $(R = 3, W = 2)$. We also note that the constrained and controlled agents ensure a significant gain in overhead, almost 80%.

Other application scenarios were tested like an application scenario (scenario n°4) where latency is favored and consistency is completely relaxed ("any"). Our results showed that, in such scenarios, the learning agents converge towards a common Quorum configuration $(R = 1, W = 1)$.

Table II summarizes the final results of the constrained and controlled agents for the considered application scenarios. More specifically, it shows the optimal Quorum configurations (R, W) reached after algorithm convergence for different application SLA requirements.

Application scenario	Latency sensitive	Staleness favoring	Read Quorum size R	Write Quorum size W
n°1	+++	+	1	3
n°2	++	++	2	2
n°3	+	+++	3	2
n°4	++++	–	1	1

TABLE II: Final Q-learning results of the constrained and controlled agents for the considered application scenarios

b) Dynamic application workloads:

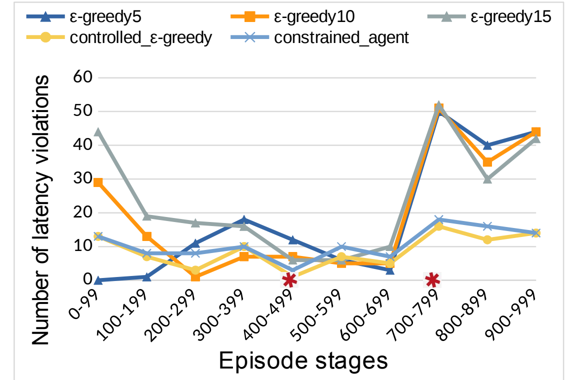
In this section, we aim to assess the ability of our adaptive Quorum-inspired consistency strategy (ONOS-WAQIC) for the C-CDN-like application we developed, to adapt to time-varying application workloads. The dynamic changes in such application workload patterns may indeed affect the observed network and application metrics (e.g. inter-controller overhead, staleness and access latency).

Taking that into consideration, our adaptive consistency model attempts to adjust the consistency level at run-time by continuously tuning the Quorum configuration parameters in order to better match the varying workloads.

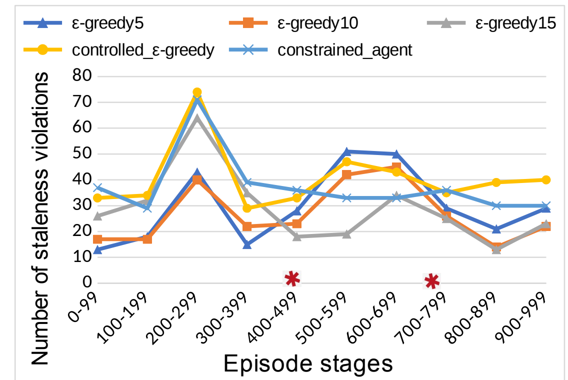
In this context, we consider three workloads as discussed in Section III-B2 (see the `Workload Identifier Module`). In the three studied workloads, our model aims to satisfy the latency and staleness SLA requirements. Additionally, in the read-dominated workload (Workload 1), our model attempts to minimize the read overhead. Conversely, in a write-intensive workload (Workload 3), it focuses on reducing the write overhead. Finally, in a balanced workload, our approach aims to minimize both the read and write overheads (the global overhead).

To experiment with these workloads, we set the application scenario to Scenario 2 (see Section V-C2a) which represents an application scenario with balanced consistency (staleness)/latency SLA requirements. Then, we conduct some tests on our Q-learning simulator. During these tests, we apply different variations in the application workload. More specifically, the first time period of the tests (the first 400 episodes) is characterized by a balanced workload (Workload 2). At episode 400, we run a read-dominated workload (Workload 1). Finally, starting from episode 700, we consider a write-intensive workload (Workload 3).

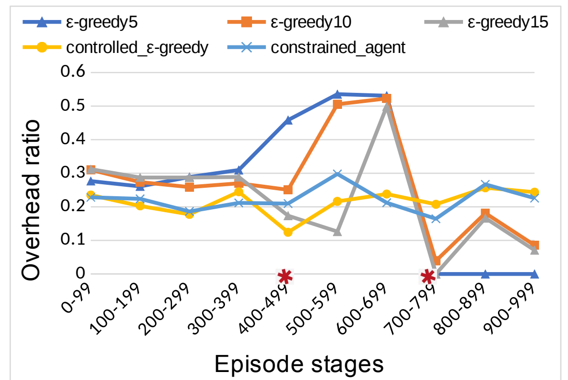
As we can see from Figure 12, our results clearly show that, unlike the simple ϵ-greedy agents, the constrained and controlled agents react quickly to the dynamic workload



(a) Latency violations



(b) Staleness violations



(c) Overhead ratio

Fig. 12: Dynamic changes in the Workload (Workload 2-Workload 1-Workload 3) in a Consistency/Latency-balancing application scenario (Scenario2)

variations. These agents not only offer balanced real-time trade-offs between the performance (latency) and consistency (staleness) application SLA requirements, but also provide minimal overhead at run-time.

Besides, when analyzing the generated Quorum configurations during the conducted tests, we observe that, in Workload 1, the constrained agent converges to Quorum configurations where R is minimal in order to reduce the application's read inter-controller overhead. On the other hand, in Workload 3, the Quorum configurations where W is small are eventually selected. Finally, in Workload 2, the constrained agent converges to balanced Quorum configurations where R=W=2 to reduce the application's read and write inter-controller overheads.

VI. Conclusion

In this paper, we studied the use of an adaptive and continuous consistency model for the distributed SDN controllers following the notion of partial Quorum consistency at scale. Our consistency adaptation strategy was implemented for a Content-Centric Delivery Networks (C-CDN) application developed on top of ONOS. It mainly consists in turning ONOS's optimistic replication technique into a more scalable and intelligent Quorum-inspired replication strategy using various online Q-learning RL approaches. Our experiments showed that the constrained ϵ-greedy approach we tested in a 5-node ONOS cluster proved efficient in helping our C-CDN-like application find the appropriate read an write Quorum replication parameters at run-time. In fact, the adjustable and time-varying partial Quorum configurations determined by our strategy at run-time have achieved, under changing network and application workload conditions, balanced trade-offs between the application's continuous performance (latency) and consistency (staleness) requirements. Besides, these real-time trade-offs ensured a substantial reduction in the application's inter-controller read and write overhead (especially in a large-scale ONOS network) while satisfying the application-defined thresholds specified in the given application SLA.

Moreover, our proposed adaptive and Quorum-inspired consistency model could be further enhanced by leveraging the compulsory Anti-Entropy reconciliation mechanisms proposed in the previous work [9] (*expanding partial Quorums*). Such mechanisms are indeed useful in particular cases (e.g. failure scenarios, controller crashes) where the system consistency observed by the applications is at high risk, and cannot be fixed only by adjusting the Quorum parameters.

Our proposed C-CCN application might obviously be leveraged by several ICN use-cases. For example, it can meet the needs of video-on-demand services, and more particularly, the requirements of live and pre-recorded streaming where video files are versioned based on current time [32]. Another interesting use-case could be vehicular networks. Our proposal can indeed provide solutions to frequent changes in network topology state and content (e.g. degree of pollution and traffic conditions) [33].

Finally, it is worth noting that our self-adaptive and automated consistency mechanisms for the distributed SDN controllers might be applied to many other concrete distributed network applications. The latter should have consistency adaptability requirements at scale such as the need for a dynamic adaptation of the replication style given the varying patterns of application behavior and network context. These applications might include Cloud data storage services, Website visitors (e.g. discussion forums), e-commerce and media-service providers (e.g. Amazon and Netflix), security applications but also smart-home services.

References

[1] Dirk Kutscher, Suyong Eum, Kostas Pentikousis, Ioannis Psaras, Daniel Corujo, Damien Saucez, Thomas C. Schmidt, and Matthias Wählisch. Information-Centric Networking (ICN) Research Challenges. RFC 7927, July 2016.

[2] Boubakr Nour, Kashif Sharif, Fan Li, Sujit Biswas, Hassine Moungla, Mohsen Guizani, and Yu Wang. A survey of Internet of Things communication using ICN: A use case perspective. *Computer Communications*, 142-143:95 – 123, 2019.

[3] B. Nour, F. Li, H. Khelifi, H. Moungla, and A. Ksentini. Coexistence of ICN and IP Networks: An NFV as a Service Approach. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec 2019.

[4] Maroua Meddeb. *Information-Centric Networking, A natural design for IoT applications?* Theses, INSA de Toulouse ; Ecole Nationale des Sciences de l'Informatique, September 2017.

[5] M. A. Salahuddin, A. Al-Fuqaha, M. Guizani, K. Shuaib, and F. Sallabi. Softwarization of Internet of Things Infrastructure for Secure and Smart Healthcare. *Computer*, 50(7):74–79, 2017.

[6] A. H. Shamsan and A. R. Faridi. Network softwarization for IoT: A Survey. In *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1163–1168, March 2019.

[7] Akbar Rahman, Dirk Trossen, Dirk Kutscher, and Ravi Ravindran. Deployment Considerations for Information-Centric Networking (ICN). RFC 8763, April 2020.

[8] F. Bannour, S. Souihi, and A. Mellouk. Distributed SDN control: Survey, taxonomy, and challenges. *IEEE Communications Surveys Tutorials*, 20(1):333–354, Firstquarter 2018.

[9] F. Bannour, S. Souihi, and A. Mellouk. Adaptive State Consistency for Distributed ONOS Controllers. In *2018 IEEE Global Communications Conference(Globecom)*, pages 1–7, 2018.

[10] ONOS. https://onosproject.org/.

[11] ODL. http://opendaylight.org/.

[12] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[13] Swaminathan Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM.

[14] Mohamed Aslan and Ashraf Matrawy. A clustering-based consistency adaptation strategy for distributed SDN controllers. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018.

[15] Ermin Sakic and Wolfgang Kellerer. Impact of adaptive consistency on distributed SDN applications: An empirical study. *IEEE Journal on Selected Areas in Communications*, page 13, 2018.

[16] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, 2013.

[17] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. Adaptive Quorum-inspired SLA-Aware Consistency for Distributed SDN Controllers. In *International Conference on Network and Service Management (CNSM 2019) (CNSM 2019)*, Halifax, Canada, October 2019.

[18] Haifeng Yu and Amin Vahdat. 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*, OSDI'00, Berkeley, CA, USA, 2000.

[19] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012.

[20] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with pbs. *Commun. ACM*, 57(8):93–102, August 2014.

[21] H. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Pérez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *2012 IEEE International Conference on Cluster Computing*, pages 293–301, Sept 2012.

[22] Sathiya Prabhu Kumar. *Adaptive Consistency Protocols for Replicated Data in Modern Storage Systems with a High Degree of Elasticity*. Theses, CNAM, March 2016.

[23] Canh Son Nguyen Ba. *Adaptive control for availability and consistency in distributed key-values stores*. Theses, University of Illinois, 2015.

[24] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying eventual consistency of optimistic replication systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 285–296, 2014.

[25] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[26] Voldemort project. http://www.project-voldemort.com/voldemort/design.html.

[27] Jing Zhong, Roy D. Yates, and Emina Soljanin. Minimizing content staleness in dynamo-style replicated storage systems. *CoRR*, abs/1804.00742, 2018.

[28] H. Ge, Y. Song, C. Wu, J. Ren, and G. Tan. Cooperative Deep Q-Learning With Q-Value Transfer for Multi-Intersection Signal Control. *IEEE Access*, 7:40797–40809, 2019.

[29] Maria Couceiro, Gayana Chandrasekara, Manuel Bravo, Matti Hiltunen, Paolo Romano, and Luís Rodrigues. Q-opt: Self-tuning quorum system for strongly consistent software defined storage. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 88–99, 2015.

[30] OpenAI Gym Project. https://gym.openai.com/.

[31] Hai-Anh Tran, Sami Souihi, Duc A. Tran, and Abdelhamid Mellouk. Mabrese: A new server selection method for smart SDN-based CDN architecture. *IEEE Communications Letters*, 23:1012–1015, 2019.

[32] Derek Kulinski and Jeff Burke. NDNVideo : Random-access Live and Pre-recorded Streaming using NDN. 2012.

[33] E. Kalogeiton, Z. Zhao, and T. Braun. Is SDN the solution for NDN-VANETs? In *2017 16th Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, pages 1–6, 2017.