UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

A Consistent and Fault-Tolerant Data Store for Software Defined Networks

Fábio Andrade Botelho

DISSERTAÇÃO

MESTRADO EM SEGURANÇA INFORMÁTICA

2013

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

A Consistent and Fault-Tolerant Data Store for Software Defined Networks

Fábio Andrade Botelho

DISSERTAÇÃO

MESTRADO EM SEGURANÇA INFORMÁTICA

Dissertação orientada pelo Prof. Doutor Alysson Neves Bessani
e co-orientado pelo Prof. Doutor Fernando Manuel Valente Ramos

2013

Agradecimentos

Aos meus orientadores — Prof. Doutor Alysson Neves Bessani, Prof. Doutor Fernando Manuel Valente Ramos, e Diego Kreutz que muito mais do que ocuparem títulos e posições, tomaram lugar numa activa equipa de trabalho que tornou este projecto possível. Um muito obrigado por toda a paciência, apoio e confiança ao longo do meu último ano. No contexto do meu percurso na FCUL queria também agradecer ao Prof. Doutor Hugo Alexandre Tavares Miranda, que muito me ensinou sobre sistemas distribuídos, e também aos companheiros do laboratório 25 por serem os melhores. Queria agradecer também aos reviewers da EWSDN pelos comentários que ajudaram a melhorar o artigo que foi resultado deste trabalho. Além disso queria agradecer ao financiamento que foi parcialmente suportado pelo EC FP7 através do projecto BiobankCloud (ICT-317871) e também pela FCT através do programa Multianual do LASIGE.

No campo pessoal queria agradecer a todos que me carregaram até aqui. Nomeadamente a minha família (pais e irmãos), que mais do que ser família, acreditou em mim, e levou-me às costas todos os anos. Eu nunca vos vou conseguir retribuir o quanto fizeram por mim. De seguida à minha companheira de viagem, Maria Lalanda. Foste o melhor deste caminho, espero no futuro perder-me apenas contigo.

Ao pessoal do Jardim, pelas baldas, pelo jardim, por todo o tempo envoltos em ritmo e poesia. São demasiados os nomes para referir aqui, mas não podia deixar de referir as pessoas que mais me apoiaram neste percurso: João Sardinha, Tâmara Andrade, e Miguel Azevedo. Em Braga, a Catarina e Filipe Rebelo, e a senhora Eleanor por terem acreditado em mim. Ao Manuel Barbosa por me alavancar, ao José Silva e a Bárbara Manso por se infiltrarem na minha casa. Vocês três são família. Um especial agradecimento ao José Silva e Manuel Barbosa, companheiros em muitas batalhas — Nunca é tarde para apanhar o comboio Cacilda! Aos Monads, à casa vintage, aos gadgets do Toxa, à bandeira que soprava na direcção de casa, e a todo o restante foklore... Em Lisboa, à Filipa Costa e João Pereira por suportarem o gajo mais chato de sempre. Ao MSI por ter me confrontado com o maior desafio da minha vida, e ao José Lopes, porque sem ele nunca teria conseguido. Fico à espera que me "faças um turing" rapaz. Ao pessoal do FCUL que muito ajudou. Em especial o Emanuel Alves, Juliana Patrícia e Anderson Barretto. À RUMO que me acolheu; é com remorso que não explorei os teus cantos. À Maria e ao José (os Manos) pela casa e paciência; igualmente ao Miguel Costa e Francisco Apolinário. Finalmente a todas as máquinas automáticas do c* que me alimentaram durante as semanas do Globox, ao café, SG Ventil, stack-overflow e lego-coding: um muito obrigado. Ao Lamport por usar t-shirts em apresentações, inventar o LaTeXe um zilião de papers. Por fim, ao Charlie, "*who has always made it out of the jungle!*". Mesmo sem cão!

Às "Torradas", à Lonjura, e às Ilhas! — "Sair da ilha é a pior forma de ficar nela" (Daniel Sá).

A todos “os meus conhecidos” que já levaram porrada.

Resumo

O sucesso da Internet é indiscutível. No entanto, desde há muito tempo que são feitas sérias críticas à sua arquitectura. Investigadores acreditam que o principal problema dessa arquitectura reside no facto de os dispositivos de rede incorporarem funções distintas e complexas que vão além do objectivo de encaminhar pacotes, para o qual foram criados [1]. O melhor exemplo disso são os protocolos distribuídos (e complexos) de encaminhamento, que os *routers* executam de forma a conseguir garantir o encaminhamento de pacotes. Algumas das consequências disso são a complexidade das redes tradicionais tanto em termos de inovação como de manutenção. Como resultado, temos redes dispendiosas e pouco resilientes.

De forma a resolver este problema uma arquitectura de rede diferente tem vindo a ser adoptada, tanto pela comunidade científica como pela indústria. Nestas novas redes, conhecidas como *Software Defined Networks* (SDN), há uma separação física entre o plano de controlo do plano de dados. Isto é, toda a lógica e estado de controlo da rede é retirada dos dispositivos de rede, para passar a ser executada num **controlador** *logicamente centralizado* que com uma visão global, lógica e coerente da rede, consegue controlar a mesma de forma dinâmica. Com esta delegação de funções para o controlador os dispositivos de rede podem dedicar-se exclusivamente à sua função essencial de encaminhar pacotes de dados. Assim sendo, os dipositivos de redes permanecem simples e mais baratos, e o controlador pode implementar funções de controlo simplificadas (e possivelmente mais eficazes) graças à visão global da rede.

No entanto um modelo de programação *logicamente centralizado* não implica um sistema centralizado. De facto, a necessidade de garantir níveis adequados de *performance*, escalabilidade e resiliência, proíbem que o plano de controlo seja centralizado. Em vez disso, as redes de SDN que operam a nível de produção utilizam planos de controlo distribuídos e os arquitectos destes sistemas têm que enfrentar os *trade-offs* fundamentais associados a sistemas distribuídos. Nomeadamente o equilibro adequado entre coerência e disponibilidade do sistema.

Neste trabalho nós propomos uma arquitectura de um controlador distribuído, tolerante a faltas e coerente. O elemento central desta arquitectura é uma base de dados replicada e tolerante a faltas que mantêm o estado da rede coerente, de forma a garantir que as aplicações de controlo da rede, que residem no controlador, possam operar com

base numa visão coerente da rede que garanta coordenação, e consequentemente simplifique o desenvolvimento das aplicações. A desvantagem desta abordagem reflecte-se no decréscimo de *performance*, que limita a capacidade de resposta do controlador, e também a escalabilidade do mesmo. Mesmo assumindo estas consequências, uma conclusão importante do nosso estudo é que é possível atingir os objectivos propostos (i.e., coerência forte e tolerância a faltas) e manter a *performance* a um nível aceitável para determinados tipo de redes.

Relativamente à tolerância a faltas, numa arquitectura SDN estas podem ocorrer em três domínios diferentes: o plano de dados (falhas do equipamento de rede), o plano de controlo (falhas da ligação entre o controlador e o equipamento de rede) e, finalmente, o próprio controlador. Este último é de uma importância particular, sendo que a falha do mesmo pode perturbar a rede por inteiro (i.e., deixando de existir conectividade entre os *hosts*). É portanto essencial que as redes de SDN que operam a nível de produção possuam mecanismos que possam lidar com os vários tipos de faltas e garantir disponibilidade perto de 100%.

O trabalho recente em SDN têm explorado a questão da coerência a níveis diferentes. Linguagens de programação como a *Frenetic* [2] oferecem coerência na composição de políticas de rede, conseguindo resolver incoerências nas regras de encaminhamento automaticamente. Outra linha de trabalho relacionado propõe abstracções que garantem a coerência da rede durante a alteração das tabelas de encaminhamento do equipamento. O objectivo destes dois trabalhos é garantir a coerência depois de decidida a política de encaminhamento. O *Onix* (um controlador de SDN muitas vezes referenciado [3]) garante um tipo de coerência diferente: uma que é importante antes da política de encaminhamento ser tomada. Este controlador oferece dois tipos de coerência na salvaguarda do estado da rede: coerência eventual, e coerência forte. O nosso trabalho utiliza apenas coerência forte, e consegue demonstrar que esta pode ser garantida com uma performance superior à garantida pelo *Onix*.

Actualmente, os controladores de SDN distribuídos (Onix e HyperFlow [4]) utilizam modelos de distribuição não transparentes, com propriedades fracas como coerência eventual que exigem maior cuidado no desenvolvimento de aplicações de controlo de rede no controlador. Isto deve-se à ideia (do nosso ponto de vista infundada) de que propriedades como coerência forte limitam significativamente a escalabilidade do controlador. No entanto um controlador com coerência forte traduz-se num modelo de programação mais simples e transparente à distribuição do controlador.

Neste trabalho nós argumentámos que é possível utilizar técnicas bem conhecidas de replicação baseadas na máquina de estados distribuída [5], para construir um controlador SDN, que não só garante tolerância a faltas e coerência forte, mas também o faz com uma performance aceitável. Neste sentido a principal contribuição desta dissertação é mostrar que uma base de dados contruída com as técnicas mencionadas anteriormente (como as

providenciadas pelo BFT-SMaRt [6]), e integrada com um controlador open-source existente (como o Floodlight[1]), consegue lidar com vários tipos de carga, provenientes de aplicações de controlo de rede, eficientemente.

As contribuições principais do nosso trabalho, podem ser resumidas em:

1. A proposta de uma arquitectura de um controlador distribuído baseado nas propriedades de coerência forte e tolerância a faltas;

2. Como a arquitectura proposta é baseada numa base de dados replicada, nós realizamos um estudo da carga produzida por três aplicações na base dados.

3. Para avaliar a viabilidade da nossa arquitectura nós analizamos a capacidade do *middleware* de replicação para processar a carga mencionada no ponto anterior. Este estudo descobre as seguintes variáveis:

   (a) Quantos eventos por segundo consegue o *middleware* processar por segundo;

   (b) Qual o impacto de tempo (i.e., latência) necessário para processar tais eventos;

   para cada uma das aplicações mencionadas, e para cada um dos possíveis eventos de rede processados por essas aplicações. Estas duas variáveis são importantes para entender a escalabilidade e *performance* da arquitectura proposta.

Do nosso trabalho, nomeadamente do nosso estudo da carga das aplicações (numa primeira versão da nossa integração com a base de dados) e da capacidade do *middleware* resultou uma publicação: Fábio Botelho, Fernando Ramos, Diego Kreutz and Alysson Bessani; *On the feasibility of a consistent and fault-tolerant data store for SDNs*, in Second European Workshop on Software Defined Networks, Berlin, October 2013. Entretanto, nós submetemos esta dissertação cerca de cinco meses depois desse artigo, e portanto, contêm um estudo muito mais apurado e melhorado.

Palavras-chave: Replicação, Coerência Forte, Redes Controladas por Software, Tolerância a Faltas, Máquina de Estados Distribuída, Plano de Controlo Distribuído.

[1] `http://www.projectfloodlight.org/floodlight/`

Abstract

Even if traditional data networks are very successful, they exhibit considerable complexity manifested in the configuration of network devices, and development of network protocols. Researchers argue that this complexity derives from the fact that network devices are responsible for both processing control functions such as distributed routing protocols and forwarding packets.

This work is motivated by the emergent network architecture of Software Defined Networks where the control functionality is removed from the network devices and delegated to a server (usually called controller) that is responsible for dynamically configuring the network devices present in the infrastructure. The controller has the advantage of logically centralizing the network state in contrast to the previous model where state was distributed across the network devices. Despite of this *logical centralization*, the control plane (where the controller operates) must be distributed in order to avoid being a single point of failure. However, this distribution introduces several challenges due to the heterogeneous, asynchronous, and faulty environment where the controller operates.

Current distributed controllers lack transparency due to the eventual consistency properties employed in the distribution of the controller. This results in a complex programming model for the development of network control applications. This work proposes a fault-tolerant distributed controller with strong consistency properties that allows a transparent distribution of the control plane. The drawback of this approach is the increase in overhead and delay, which limits responsiveness and scalability. However, despite being fault-tolerant and strongly consistent, we show that this controller is able to provide performance results (in some cases) superior to those available in the literature.

Keywords: Replication, Strong Consistency, Distributed State Machine, Distributed Control Plane, Software Defined Networking.

Contents

List of Figures

List of Tables

Chapter 1 – Introduction

The nice thing about standards is that you have so many to choose from.

Andrew S. Tanenbaum

1.1 Software Defined Network

Despite its success, current Internet Protocol (IP) networks suffer from problems, which have long drawn attention of the network academic community. Researchers have been tackling those problems with one of two strategies: tailoring the performance of IP based networks and/or providing point solutions to new technological requirements (incremental approach); or redesigning the entire architecture from scratch (clean slate approach). Software Defined Network (SDN) is the pragmatic result of several contributions to the clean slate approach that has the benefit of radically transforming the management and functionality of IP networks, albeit maintaining intact traditional *host-to-host* protocols (i.e., the TCP/IP stack[1]).

In a nutshell, SDN shifts the control logic of the network (e.g., route discovery) from the network equipment, to a commodity server where network behavior can be defined in a high-level control program, without the constraints set by the network equipment. Thus, there is a separation of the control plane, where the server operates, from the data plane, where the network infrastructure (the switches & routers) resides. A fundamental abstraction in SDN is *logical centralization* that specifies that the control plane operates with a logically centralized view of the network. This view enables simplified programming models and facilitates network applications design.

A logically centralized programming model does not postulate a centralized system. Arguably, a less-prone-to-ambiguity definition for "logically centralized" could be "transparently distributed" because *"either you're centralized, or you're distributed"* [7]. In fact, the need to guarantee adequate levels of performance, scalability, and reliability preclude a fully centralized solution. Instead, production-level SDN network designs resort

[1]The TCP/IP stack is a common name for the Internet protocol suite comprising a networking model and set of communication protocols used in the Internet.

to physically distributed control planes. Consequently, the designers of such systems have to face the fundamental trade-offs between the different consistency models, the need to guarantee acceptable application performance, and the necessity to have a highly available system.

In this document, we propose a SDN controller architecture that is distributed, fault-tolerant, and strongly consistent. The central element of this architecture is a data store that keeps relevant network and applications state, guaranteeing that SDN applications operate on a consistent network view, ensuring coordinated, correct behavior, and consequently simplifying application design. The drawback of this approach is the decrease of performance, which limits responsiveness and hinders scalability. Even assuming these negative consequences, an important conclusion of this study is that it is possible to achieve those goals while maintaining the performance penalty at an acceptable level.

1.1.1 Standard Network Problems

Traditional IP networks are complex to manage and control. Researchers argue that this complexity derives from the integration of network control functionalities such as routing in the network devices, which should be solely responsible for forwarding packets [1]. For example, both Ethernet[2] switches and IP routers are in charge of packet forwarding as well as path computation[3]. In fact, control logic is responsible for tasks that go beyond path computation such as discovery, tunneling, and access control. This vertical integration of the control and data plane functions is undesirable for multiple reasons.

First, network devices need to run multiple ad-hoc, complicated distributed protocols to implement a myriad of control functions, which are (arguably) more complex than centralized control based on a global view of the network state. This problem is aggravated if we consider the scale of existent networks. To make matters worse, each of those control functions commonly work in isolation. To exemplify, while routing adapts to topology changes, the same is not true for the access control process (which controls which hosts and services can communicate in the network), thus making it possible for a link failure to cause a breach in the security policy that must be solved by human intervention (see section 2.1 of the 4D paper [1] for an example).

Second, processing the control algorithms consumes significant resources from the network equipment, thus requiring powerful and expensive hardware. The path computation itself is so complex that it may require devices to concurrently execute several control processes and/or maintain a global view of the network (in each device!).

Third, the current state of affairs impairs network innovation since new network protocols must undergo years of standardization and interoperability testing because they are

[2] A family of computer networking technologies for local area networks.

[3] Paths are found by processing routing protocols in the IP case, and by processing the Spanning Tree Protocol (STP) in the Ethernet case, as well as a continuous learning process for associating devices with their correct ports (see section 4.2).

implemented in closed, proprietary software and cannot be reused, modified, or improved upon. As such, researchers have difficulty in testing and deploying new control functionality in real networks [8].

Finally, configuring the data plane is typically performed with low level and device-dependent instructions, a very error-prone process. Empirical studies have found more than one thousand routing configurations faults in routers from 17 different organizations [9]. Moreover, misconfiguration can have severe consequences. As an example, in 2008 Pakistan Telecom took YouTube offline almost worldwide with (arguably) misconfigured routing directives, while following a censorship order [10]. Consequently, administrating large networks requires significant (and expensive) human resources.

In conclusion, coupling the data plane and control plane into a vertical integrated, proprietary solution resulted in networks that are difficult to manage, hard to innovate, and expensive to maintain.

1.1.2 Logical Centralization

SDN is a novel architecture that emerged to resolve the drawbacks set by closely coupling the control and data planes. Fig. 1.1 shows that this architecture physically decouples those planes. In SDN all network control functionality such as routing, load balancing, etc., can be defined in software and performed by a controller with the help of a logically centralized *Network View* containing all the relevant network state (e.g., network topology, forwarding tables, security policy). This state can be present in the controller memory or in a data store. Furthermore, the controller can be a generic software framework that supports multiple applications; each specialized in different control functionalities. These applications can collaborate and coordinate through the *Network View*.

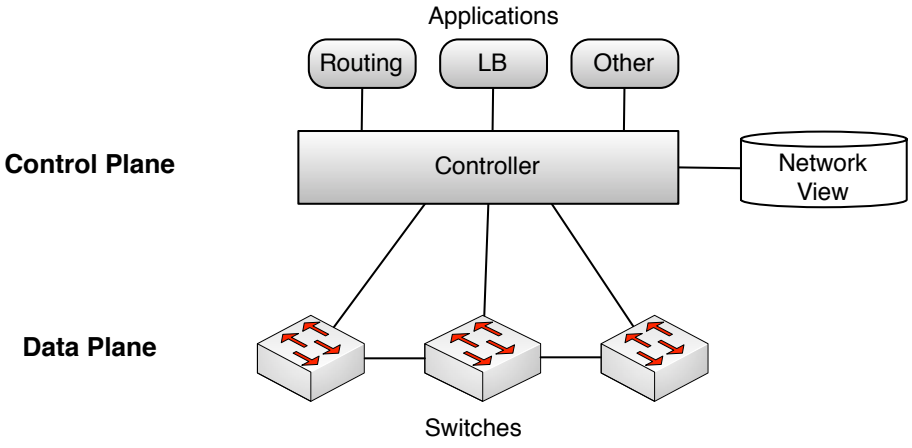


Figure 1.1: SDN architecture: the controller maintains a connection to the network devices residing in the data plane. The *Network View* contains all the relevant network state (e.g., topology information) and configuration (e.g., access control). The controller uses this state to configure the switches (the network devices). Additionally, the controller runs specialized applications that focus in particular control tasks: Routing, Load Balancing, and others.

In order to separate the control plane from the data plane it is crucial that the latter implements an interface to allow the configuration of the network devices. OpenFlow (OF) is the most common protocol that implements this interface [8, 11]. In OF forwarding is based on flows, which are broadly equivalent to a stream of related packets. The control plane manipulates the flow-based forwarding tables present in network devices, such that devices can recognize flows (e.g., *any TCP packet destined to port 80* or *any IP packet destined to 1.1.1.1*), and associate them with actions (e.g., *drop packet*, *forward to port* x, *forward to controller*).

In general, the control plane task can be seen as to implement a function f, representing all control functionality, having the *Network View* as input and the configuration of network devices as output. Alternatively, (and most commonly) the devices can request forwarding "advice" to the controller for a specific packet that represents a flow; in OF, this is called a *flow request*. In this case, function f is expanded to receive both the network state and the specific packet as input, and the output determines a viable configuration for forwarding packets for that particular flow. We clarify this further in section 2.2.

In SDN, the *Network View* should be updated as conditions change. To attain this goal, OF switches update the controller with network events such as topological changes (e.g., *link up*, *link down*). Additionally, the controller can retrieve the switches internal state to obtain traffic-based information, such as packet counters (e.g., *how many packet have been forwarded to port* x), that can amplify the network view. Thus, it is possible to have a dynamic *Network View* with different levels of granularity in order to satisfy different control plane requirements.

When compared to the traditional network architecture SDN can mitigate all the pitfalls identified in the previous section. First, it enables centralized algorithms for control that operate on a coherent global view of the network. Furthermore, the control functions can work in concert (i.e., obtain feedback of one another by collaborative building the network view). Second, since devices only perform forwarding functions, they can run on much simpler (and cheaper) hardware. Third, since the control plane task can be performed in commodity servers, running accessible open-source software, researchers are able to test, deploy, and evaluate innovative control functionality in existent networks. Additionally, these experiments can run side-by-side with production traffic [8] Finally, the configuration is now software-driven, which (arguably) can be much simpler than low level device configuration, and can benefit from well-known software development corroboration techniques such as formal verification [12], debuggers [13], unit testing, etc.

It is worth pointing out that Software Defined Network is not just an artifact for the scientific community, but it is also being adopted by the industry. For example, Google has deployed a Software Defined WAN (Wide Area Network) to connect their datacenters [14]. Additionally, this company and other industry partners (Yahoo, Microsoft, Face-

book, Verizon and Deutsche Telekom, Nicira, Juniper, etc.) have formed the Open Network Foundation (ONF)[4] — a non-profit organization responsible for the standardization process of SDN technology. Finally, several network hardware vendors currently support OpenFlow in their equipment. Examples include IBM, Juniper, and HP.

1.1.3 Distributed Control Plane

Most SDN controllers are centralized, leaving to their users the need to address several challenges such as scalability, availability, and fault tolerance. However, the need for distribution has been motivated recently in the SDN literature [3, 4, 14, 15].

The following arguments support the distribution of the control plane:

Scalability: The controller memory contains the network state, and the CPU processes network events — mainly flow requests. The use of both these resources grows with the size of the network, eventually leading to resource exhaustion. Thus, a scalable control plane requires the distribution of the network state and/or event processing across different machines;

Performance: Scalability may partially be considered for performance reasons also. However, there can be more intransigent performance requirements such as in the case of a Wide Area Network (WAN) where big latency penalties may be observed between the control and data plane communication;

Fault Tolerance: Network control applications built in the controller may require the availability and durability of the service. Even if failures in the control plane are inevitable, it is desirable to tolerate those without disrupting the network.

Scalability is a fundamental reason for distributing a computational system. Although centralized controllers have been reported to handle tens of thousands of hosts [16], and a million events per second (averaging 2.5 ms per event) [17], there are limits in resources that will eventually lead to their exhaustion. These limits are easily reached in current data centers and WANs. Namely, there is evidence of data centers that can easily reach thousands of switches and hundred of thousands hosts [18]. Also Benson et al. show that a data center with 100 edge[5] switches can (in the worst case) have spikes of 10M flow arrivals per second [19]. These numbers strongly suggest distributing the control plane in order to shield controllers from such a large number of network events.

The **Performance** reason presented is also fundamental. At the time of writing only one SDN enabled WAN is known [14], but given its publicized success caused by the

[4] `https://www.opennetworking.org/`

[5] The three tier data center topology is an hierarchical topology with 4 levels. In the bottom levels reside the application servers connected to the edge switches.

cost-effective bandwidth management, one could expect more to follow. Even though the control plane only requires processing the first packet in a flow, the latency established in this communication must be minimal such that network applications are not noticeable affected. Distribution can mitigate the latency problem by bringing the control plane closer to the data plane [15, 20].

Finally, **Fault tolerance** is an essential part of any Internet-based system, and this property is therefore typically built-in by design. Solutions such as Apache' Zookeeper (Yahoo!) [21], Dynamo (Amazon) [22] and Spanner (Google) [23] were designed and deployed in production environments to provide fault tolerance for a variety of critical services. The increasing number of SDN-based deployments in production networks is also triggering the need to consider fault tolerance when building SDNs. For example, the previously mentioned SDN based WAN requires (and employs) fault tolerance [14].

SDN fault tolerance covers different fault domains [24]: the data plane (switch or link failures), the control plane (failure of the switch-controller connection), and the controller itself. The latter is of particular importance since a faulty controller can wreak havoc on the entire network. Thus, it is essential that production SDN networks have mechanisms to cope with controller faults and guarantee close to 100% availability.

In summary, there are various reasons for distributing the control plane, but they are multiple challenges in doing so. Namely, the generality of the control plane anticipates the usage of arbitrary control applications that will require different distribution mechanisms, to fulfill their different requirements in terms of scalability, performance, and reliability.

1.1.4 Consistency models

In order to provide fault tolerance most distributed systems replicate data. Different replication techniques are used to manage the different replicas (e.g., servers), relying on the assumption that failures are independent (i.e., the failure of one replica does not imply the failure of another). Thus, with a minimal (varying) number of accessible and correct replicas, the system will be available to its respective clients.

However, from the client viewpoint the techniques used to manage the replicas are irrelevant. The client interest lies on the consistency model describing the exact semantics of the read and write operations performed on the replicated system such as: *"will this update survive replica failures?"* and *"when is this update seen by other clients?"*. This interest is well captured by Fekete and Ramamritham: *"the principal goal of research on consistency models is to help application developers understand the behavior they will see when they interact with a replicated storage system, and especially so they can choose application logic that will function sensibly"* [25]. Indeed, the semantics of the system operations captured by the consistency model is of paramount importance for the client.

The most widely employed models of data consistency are strong and eventual consistency. The eventual model favors availability and performance of the system at the

expense of consistency pitfalls such as stale data and conflicting writes (requiring conflict resolution techniques); the strong model favors consistency at the expense of availability and performance [26, 27]. Thus, the choice of model significantly affects the system characteristics.

In the SDN context, Levin et al. [28] have analyzed the impact an eventually consistent global network view would have on network control applications — in their study, they considered a load balancer — and concluded that state inconsistency may significantly degrade their performance. Similar trade-offs arise in other applications, such as: firewalls, intrusion detection, and routing. This study is a clear motivation for the need of a strongly consistent network view, at least for some applications.

Eventual consistency also affects *distribution transparency* — the ability of the system to hide the fact that it is distributed [29] — since the client can be made aware of the distribution of the system. Namely, the client may be unable to read his own writes or confronted with un-merged conflicting data. This also affects simplicity since the user of the system is forced to compensate the lack of consistency. To do so, the user is limited to external synchronization or data structures that avoid conflicts [30]. In comparison, the strong consistency model favors transparency and simplicity since the user of such system is unable to distinguish it from a centralized service (without replication).

Recent work on SDN has explored the need for consistency at different levels. Network programming languages such as Frenetic [2] offer consistency when composing network policies (automatically solving inconsistencies across network applications' decisions). Other related line of work proposes abstractions to guarantee data plane consistency during network configuration updates [31]. The aim of both these systems is to guarantee consistency *after* the policy decision is made. Onix [3] (an often-cited distributed control plane) provides a different type of consistency: one that is important *before* the policy decisions are made. Onix provides network state (i.e., present in the *Networ View*) consistency — both eventual and strong — between different controller instances. However, the performance associated with strong consistency is limited (in Onix).

In our work, we favor consistency and, consequently, transparency in the distribution of the network state. Our aim is to simplify application development while still guaranteeing acceptable performance and reliability. One of our goals is to show that, despite the costs of strong consistency, the *"severe performance limitations"* [3] reported for Onix's consistent data store are a consequence of their particular implementation and not an inherent property of these systems.

1.2 Goals and Contributions

In this thesis, we argue that it is possible, using state-of-the-art replication techniques, to build a distributed SDN controller that not only guarantees strong consistency and fault

tolerance, but also does so with acceptable performance for many SDN applications. In this sense, the main contribution of this thesis is to show that if a data store built using these advanced techniques (e.g., as provided by BFT-SMaRt [6]) is integrated with a production-level controller (e.g., Floodlight[6]), the resulting distributed control infrastructure could handle efficiently many real world workloads.

The contributions can be summarized as following:

- A distributed controller architecture exhibiting strong consistency and fault tolerance is proposed (chapter 3);
- As the architecture is based on a replicated data store, we performed a study of the workloads produced by three controller applications on such data store (chapter 4);
- To assess the feasibility of our architecture, we evaluated the capability of a state-of-the-art replication middleware to process the workloads mentioned in the previous point (chapter 4). Namely:
  - How much data plane events can such a middleware handle per second;
  - What is the latency penalty for processing such events.

  These two variables are important to understand how such system would scale and perform.

We note that this thesis resulted in the following paper:

- Fábio Botelho, Fernando Ramos, Diego Kreutz and Alysson Bessani; *On the Feasibility of a Consistent and Fault-Tolerant Data Store for SDNs*, in Second European Workshop on Software Defined Networks, Berlin, October 2013.

1.3 Planning

Fig. 1.3 shows the activities that we have performed in order to produce this dissertation. The scheduling deviated from the initially proposed work plan (Fig. 1.2) for multiple reasons including the submission and presentation of the paper. Ultimately the dissertation was delayed since it was in the author interest to do so (for private reasons). This can be seen in Fig. 1.3 — in August, the decision to improve the data store (*New Data Store Functionality*) was taken, instead of writing the dissertation. The total delay accounts for five extra months for the initial prevision of finishing in June 2013.

[6] `http://www.projectfloodlight.org/floodlight/`

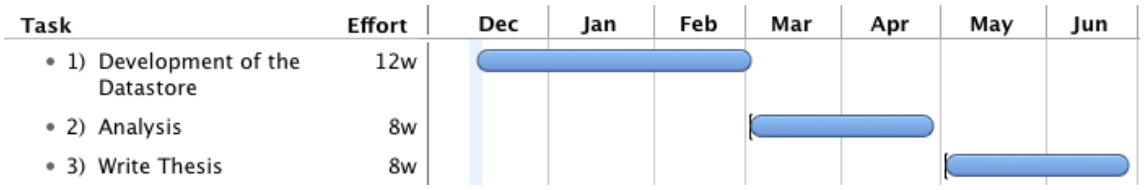


Figure 1.2: The activities (and respective durations) anticipated to produce this dissertation.

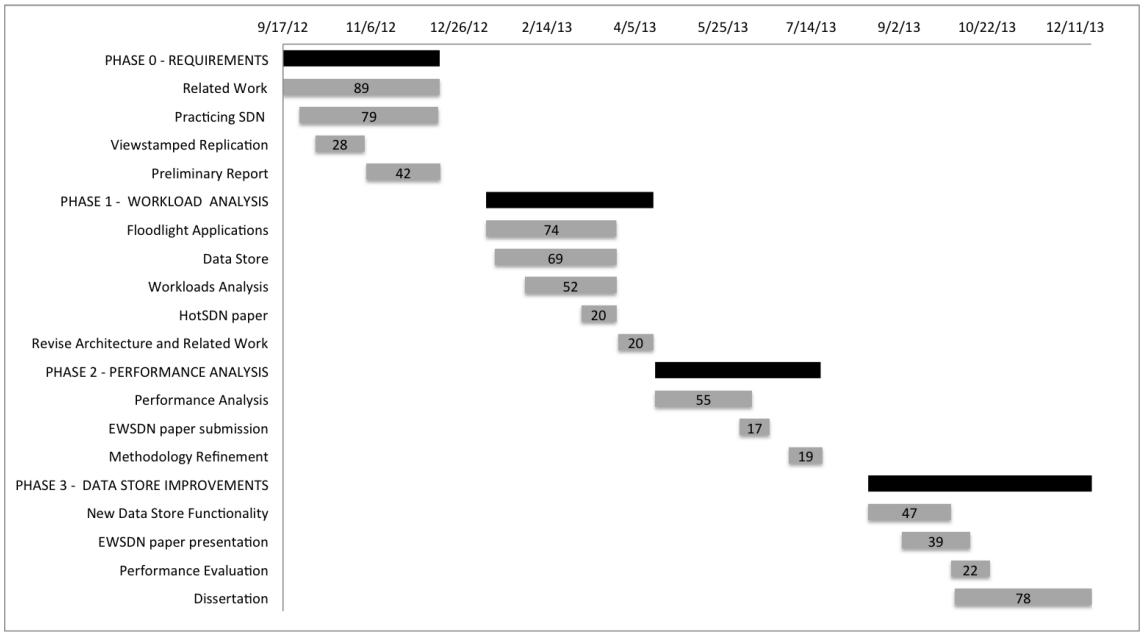


Figure 1.3: The activities (and respective durations) required to produce this dissertation.

We identify four main phases in the production of this dissertation. A summary follows:

1. *Phase 0 - Requirements* — In this phase, we collected information from SDN and state machine replication literature. Concurrently, we also evaluated the existent open source controllers that were candidates for the work covered in chapter 4. We spent considerable time learning the Beacon controller (see section 2.3.3), since we considered that it was the best fit for our requirements. However, in late December, we decided to use Floodlight instead (see section 2.3.4). The Viewstamped Replication protocol study (covered in section 2.5.4) took 28 days, since we planned to implement it, but this turn out to be unnecessary, since our colleagues were kind enough to adapt BFT-SMaRt to this protocol. The results from this phase are shown in chapter 2.

2. *Phase1 - Workload Analysis* — In this phase, we decided that our work would focus in analyzing the workloads produced by existent applications in Floodlight controller. Thus, we developed a data store prototype (covered in section 3.2), and integrated three Floodlight applications (covered in chapter 4) and a third one, the

Topology Manager (responsible for maintaining the topological view of the network in the controller), which was later abandoned, due to its complexity. The main activity during this phase was to analyze the workloads generated by the application (e.g., through statistical analysis of the number of messages per network event, and cumulative distribution of the messages sizes). The results obtained were not included in the dissertation, but the resultant codebase was leveraged to produce the study covered in chapter 4. Ultimately, the goal of this analysis was to publish a paper covering the feasibility of our distributed architecture (covered in chapter 3). We have not submitted the paper since we were confronted with a possible misunderstanding of the field, which was later, found ungrounded.

3. *Phase 2 - Performance Analysis* — In this phase, we took our study a step further, by including the performance analysis of the data store when processing the workloads generated by the applications. This phase resulted in an accepted paper at EWSDN. Right before summer, we revised our methodology to analyze workloads associated to a single data plane event as opposed to a single host-to-host interaction (as seen in the paper).

4. *Phase 3 - Improvements* — In the beginning of this phase we focused in improving the data store (with all the functionalities covered in section 3.2, with the exception of the simple key value table). Additionally, we had to prepare the paper presentation and, finally, write the dissertation.

1.4 Thesis Organization

Chapter 2 covers the background and related work. This is an extensive chapter: we present seminal papers on SDN (section 2.1); discuss the SDN architecture (2.2); survey the literature on open source centralized control planes (2.3); and distributed control planes (2.4); and finally present work on strong consistent data stores (2.5); and consistent data planes (2.6).

Chapter 3 proposes a architecture for a distributed control plane based on a fault-tolerant, strongly consistent data store; and the technical details behind the data store implementation that are crucial to understand Chapter 4.

Chapter 4 covers the integration of three different SDN applications with our data store implementation; the workloads generated by the applications for different network events; and a performance evaluation of a strongly consistent replication middleware to process the aforementioned workloads.

Finally, Chapter 5 summarizes this work, and concludes.

Chapter 2 – Related Work

If the only tool you have is a hammer,
you treat everything like a nail.

Abraham Maslow

2.1 Software Defined Networks History

Traditionally, networking relied and evolved over a non-transparent distributed model for the deployment of protocols and configuration of devices that resulted in complex protocols and intrinsically difficult configuration of network devices. Software Defined Network (SDN) presents a new way of thinking in networking, shifting the complexity of protocols and management functions from the network devices to a general purpose logically centralized service. The motivation behind this decoupling is the following. If the distributed network state can be collected and presented to a logically centralized service then it is simpler both to specify network level objectives as well as to translate these objectives into the appropriate configuration of network devices. These planes, when loosely coupled, can simplify network management.

This section outlines the major contributions that led to the SDN paradigm. In order to understand how SDN works today, it is fundamental to understand how it came to be. Understanding the biggest contributions in the SDN research field allows us to draw a better picture of its composition, benefits, and drawbacks, thus our historical perspective handpicks works that have molded the current SDN architecture. SDN contributions can be decomposed in three phases: the introduction of programmable network hardware; the control and data plane separation; and, finally, the *de facto* standardization of the data plane interface [32]. We only cover the last two since they are the ones directly related to our work. For an overview of all the existing work, we direct the reader to the survey by Feamster et al. [32].

2.1.1 RCP

In 2004, Feamster et al. proposed the Routing Control Platform (RCP) architecture to solve scalability and correctness problems in the interior Border Gateway Protocol (iBGP) protocol — a crucial component of the Internet used between routers inside an Autonomous System (AS)[1] to distribute external routing information [33]. A year later Caesar et al. described and evaluated a functional RCP system [34].

With standard iBGP, one must choose between scalability and correctness. On one hand, a full mesh configuration is correct, but requires a connection between any two routers, which significantly hinders the scalability of the system. On the other hand, the route reflection technique used to mitigate this scalability problem imposes a hierarchic structure between routers that narrows their knowledge of the routing state, which can cause problems such as route oscillations, and persistent forwarding loops. In contrast, in the RCP architecture routers send the routing information to a server that is responsible for routing decisions. As such, this server is able to maintain the global routing state in spite of operating with only one connection per router. This architecture is able to scale better than the previous due to the lower number of connections, and it is correct given that the server has a complete view of the routing information.

We note that the RCP architecture is an SDN whereby the routing state and logic are decoupled from routers (the data plane) and shifted to a server (the control plane). This is possible because routers expose a well-defined interface for the dynamic configuration of the routing tables (the iBGP protocol itself). As a result, the data plane configuration is defined in software by a decoupled server that has access to the global network view (the routing state). This is therefore an SDN, albeit at the time it was not called as such.

The authors of this work advocate the distribution of the RCP architecture in order to avoid having a single point a failure and to amplify its scaling ability. In this setting, each router connects to more than one RCP server that can assign routes to any router under their command. Interestingly, and in opposition to natural reasoning, the paper shows that RCP does not requires a strong consistency protocol between servers in order to guarantee that the routes assigned by two different servers do not conflict (e.g., cause a loop) but, as laid out by the authors, this observation relies on a weak assumption. Namely, it assumes a stable period of the routing protocols whereby each server has a complete view of the routing state (thus ensuring some degree of coordination). Therefore, under transient periods of the routing updates the RCP replicas might install conflicting rules in the routers. Hence, not having a consistency protocol may lead to inconsistencies in these periods. Finally, (and again as stated by the authors), the lack of a consistency protocol affects the liveness of the system because the servers might not assign routes to routers in order to avoid conflicts.

[1] A collection of Internet Protocol (IP) routing devices under the administrative control of a single organisation.

2.1.2 4D

In 2005, Greenberg et al. published a seminal paper proposing a clean slate network architecture, named 4D, to address the root problem of traditional networks: the complexity that derives from intertwining the control and data plane on the network elements [1]. In this work, the authors argue that with the current state of affairs the management of networks is achievable only through dreadful ad-hoc manual intervention (which is error-prone); the specification of network level goals is difficult since the control functions are uncooperative; and the development of control functionalities is complex due to the distributed nature and scale of the data plane. This is the motivation for SDN.

The 4D architecture builds on 3 principles: specification of global network level goals (e.g., routing, security policy), centralization of the network state (e.g., topology, link state), and run-time configuration of the network elements. With those principles one should be able to avoid the hazard of low-level protocol-dependent configuration commands distributed across devices and development of distributed algorithms for control functions. To achieve these goals the authors' propose a network architecture structured in 4 layers (hence the name): the **data** plane to forward packets; the **discovery** plane to collect information from the data plane; the **dissemination** plane to configure the data plane; and the **decision** plane to translate network goals into the configuration of the data plane.

When compared to the RCP architecture, 4D exposes the same patterns of decoupled planes, centralized network state, programmable devices, and so on. However, while RCP only separates routing from routers, the 4D architecture envisions an extreme design whereby all control functionality is separated from the network devices. From the 4D perspective, devices should only be responsible for forwarding packets. Moreover, the authors argue that it should be possible to use the existing forwarding mechanisms (e.g., forwarding tables, packet filters and packet scheduling) as a unified resource at the service of high-level network goals (e.g., *"keep all links below 70% utilization, even under single-link failures"*). Interestingly, this work has *"generated both broad consensus and wide disagreement from the reviewers"* since all reviewers recognized the problem but did not trust on the proposed solution [1]. The SDN architecture is similar to 4D, with the discovery and dissemination plane being tied in a single protocol (see section 2.2).

2.1.3 Ethane

In 2007 Casado et al. proposed a network architecture for the enterprise with an emphasis in security [16]. This architecture, named Ethane, follows the 4D footprints and identifies the same problems in the traditional network management domain. Namely, the configuration errors caused by distributed configuration and lack of cooperating control functions. To address those issues they introduced the controller as a logically centralized server in

charge of all network devices. By maintaining a global network view that includes topology and bindings between users, host machines and addresses, the controller is able to play the interposition role between any communications occurring in the network. As such, it can ensure that a network policy is correctly respected.

In Ethane, network devices (i.e., switches) do not perform regular operations such as learn address, packet filtering, routing protocols, etc. Instead, they are limited to flow-based forwarding with the help of a local flow table that is maintained by the controller. Every packet that fails to match with any of the rules available in the flow table is redirected to the controller. Then the controller takes the packet information, current network state and security policy and decides on a forwarding policy for that packet and the ones that will follow (i.e., the packets belonging to that flow).

Arguably, the most significant contribution of Ethane came from the results obtained with its deployment in a campus network with approximately 300 hosts. According to the reported results, the authors estimate that a single desktop computer could handle a campus network with 22 thousand hosts (generating 10 thousand flows per second at peak time). This is interesting since, at the time, one of the biggest objections to decoupled architectures was its scalability and resilience. Consequently, in the paper we also see a significant discussion regarding the resilience of Ethane.

The authors present different distribution modes to achieve resilience. In the primary-backup (or passive) mode there is one active controller while others standby to take its place in the event of failure. The standby controllers can keep slow-changing state (such as configuration and user registration) consistently, and keep the binding information, that associates network (host) devices to users, eventual consistent. The only problem identified in the paper with this mode is that in the event of failures some users may need to re-authenticate. In the active mode, two or more controllers control the network while switches distribute their requests across the existing controllers. Under this mode, the authors argue that the use of eventually consistent semantics for replication is sufficient in most implementations. To sustain the argument, the authors observe that the use of consistency aware data structures such as Convergent Replicated Data Type (CRDT)[2] or simple partitioning schemes such as the separation of the IP address space across controllers for Dynamic Host Configuration Protocol (DHCP)[3] allocation can be sufficient to avoid conflicts. Albeit arguing for eventual consistency they do state that there is need for further study and stronger consistency guarantees, such as those provided by State Machine Replication (SMR), as used in our distributed control plane.

We partially agree with the authors, but we further believe that in a sensible deployment where the security policy must be strictly respected in every host-to-host communication, it is crucial that the distribution of the control plane is built on top of resilient

[2] A CRDT is an conflict-free data structure such as a set.

[3] A networking protocol used by servers on an IP computer network to allocate IP addresses to computers dynamically (without administrating intervention).

and coherent distribution models. However, due to the asynchronous nature of a network (e.g., as soon as you install a rule in a switch, the network policy or topology may already have changed) we may need more than strong consistency semantics for the control plane. In fact, we may need to expand the strong consistency semantics to the data plane (as covered in section 2.6). Interestingly, our results suggest that our control plane is capable of handling the load aimed by Ethane (a 22 thousand hosts network; see chapter 4).

To summarize, Ethane was probably the first example of a *functional* SDN architecture capable of fully operating with all the control logic separated from the network devices, which were now reduced to simple forwarding equipment configured in run-time by the controller with the help of a logically centralized network state.

2.1.4 OpenFlow

In 2008, McKeown et al. introduced the OpenFlow (OF) protocol that would become the *de facto* standard to program the data plane, thus "enabling" the broad adoption of SDN architectures in the years to come [8]. OpenFlow proposed an open interface to program network devices (that were previously closed by nature). A common analogy in this respect is to compare network devices to processors, and OF to an instruction set that exposes the underlying functional behavior through a well-defined interface. In OF, network devices are limited to flow-based forwarding only (as in Ethane). A flow table resides in the network device and is composed of tuples $\langle match, action \rangle$. The $match$ entry allows the device to match arriving packets (as flows), while $action$ specifies the forwarding behavior. Devices can match packets against standard fields in Ethernet, IP, and transport headers and follow actions that can range from drop, forward to port(s), or forward to the controller. The controller can install and manipulate this table as it sees fit. We cover OF in more detail later (see section 2.2.2).

OpenFlow rises from the success of previous systems, like Ethane. In fact, McKeown et al. do not introduce any groundbreaking novelty with respect to Ethane but rather generalize a technology already seen possible in Ethane switches. Furthermore, the paper can be seen as a "call to arms" to two different communities: the industry community to adopt the protocol in their equipment, and the academic community to push for innovation even outside the SDN context. The protocol was quickly adopted for three reasons. First, it lowers the innovation barrier by "opening" the existent networks (and correspondent devices) to the deployment of new protocols, that could even run in parallel but isolated from production traffic. Second, it also lowers the innovation barrier for the development of SDN architectures. Third, the OF interface contract only requires capabilities that were already present in the hardware thus network devices could support it with a simple firmware upgrade.

In summary, OpenFlow generalizes the technology for the switches already seen feasible in Ethane. Its success, is attested by the several network manufacturers that have

included the OF technology in their equipment.

2.1.5 Network Operating System

SDN gained a significant momentum with the adoption of the OF protocol. Up to this point, the existent work focused in monolithic control planes to satisfy specific network goals (e.g., routing in the Routing Control Platform; security in Ethane). By generalizing the data plane interface OF created the possibility of building abstract and general control planes that consolidate the network control functions.

The first such example, in the context of SDN, was (to our knowledge) the work introduced by Gude et al. [35]. Their proposal is to decompose the control plane in two layers: Management and Control. In the management layer resides the application logic, divided into applications focused in particular network goals (e.g., routing, firewall, load balancing). With this new approach the control plane can be compared to a Network Operating System (NOS) that provides the common functionality that ought to be shared between all applications, such as network device communication, state management, and application coordination. Those tasks are roughly comparable to the ones performed by a regular Operating System (i.e., device drivers, shared memory and scheduling).

The NOS enables the development of a broad spectrum of network applications that can ultimately cooperate and share information. This ability is provided by a programming interface and should allow them to control and observe the state of the network. In section 2.3 we explore how the current state of the art manages this. Interestingly, the architecture of the NOS has not changed drastically, and this paper continues to have a deep impact on the more recent open source controllers that we see today.

2.2 Software Defined Networks Fundamentals

In March 2011, the Open Network Foundation (ONF) was created with the participation and support from several industry partners[4]. ONF is a “non-profit consortium dedicated to the transformation of networking through the development and standardization of a unique architecture called Software-Defined Networking (SDN)”. They have done so, by releasing a white paper defining the design principles of the SDN architecture. They are also responsible for the standardization process of the OpenFlow protocol.

The Software Defined Network (SDN) architecture is not a standard but rather an architectural approach to networking. It should be clear, by now, that SDNs are based on three essential design principles: decoupling of the control plane from the data plane, logically centralized network view, and programmable remote access to network devices.

So far, the SDN architecture has not converged on well-defined interfaces that we could use as reference to define its meta-architecture. Nonetheless, there are common el-

[4]https://www.opennetworking.org/

ements throughout most of the existing SDN literature. Consequently, this section defines a common language used throughout this document to refer to different SDN components. This includes a definition of a meta-architecture for the SDN stack (section 2.2.1) and an overview of the most common data plane interface: OpenFlow (section 2.2.2). This section can (and should) be used as reference throughout the text.

2.2.1 Architecture

The common SDN architecture is a three-layer stack shown in Fig. 2.1. This stack includes three planes: the Management plane to specify network goals through applications, the Control plane to support the Management applications and configure the Data plane, and the Data plane to forward packets. The remaining of this section details all the components seen in the figure.

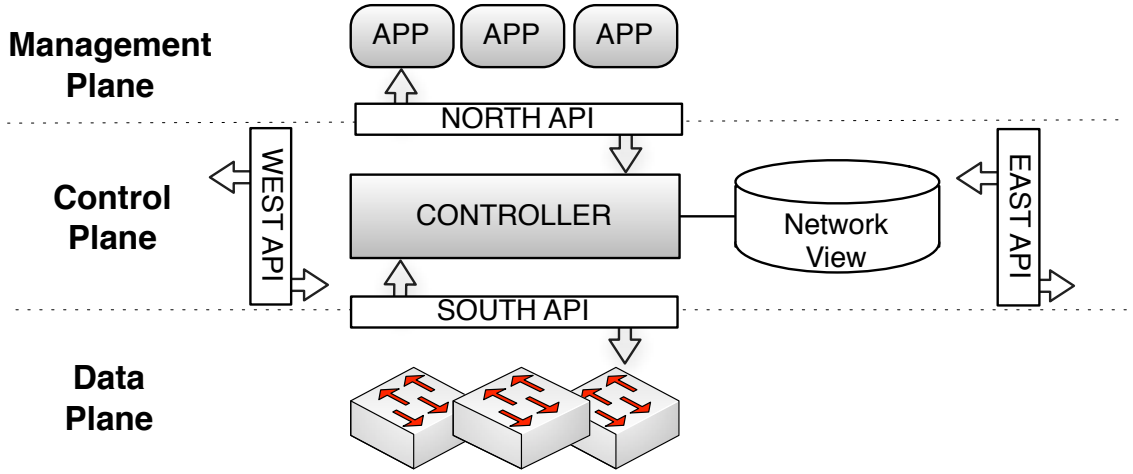


Figure 2.1: General SDN Architecture. Contains three planes: Management where applications reside (APP), Control where the controller resides, and Data where the switches reside. The control plane maintains a logically centralized Network View containing the relevant network state such as the topology. The planes interact through bidirectional API's. The WEST/EASTbound API is the same and enabled only in distributed control planes.

The network logic operates in the **Management** plane. This plane defines the global network level objectives in the form of one or more (possibly) cooperative applications. Examples include load balance, routing, firewalls, among others. From a conceptual point of view, these applications interact with the controller through the Northbound Application Programming Interface (API) (*NORTH API* in the figure). A possible implementation of this interface is a Representational State Transfer (REST) web service[5] [36] implemented in the controller that allows remote applications to read and modify the network state and configuration. It is also common for controllers to enable an event-driven programming model whereby applications subscribe to network events through this API. This

[5] A web API based on HyperText Transport Protocol (HTTP) protocol.

is typically applicable to in-memory applications but does not need to be the case. For this reason, the figure also shows the controller invoking operations on the Management plane through the Northbound API (i.e., the API is bidirectional). All the control planes described in this document support this model. Truly, SDN is event-driven from the very beginning, since network events such as topology changes and forwarding requests are asynchronous in nature. Therefore, the event-driven programming model is present in the entire SDN stack.

In the **Control** plane resides the core logic that glues the stack together. This component has three major responsibilities. First, it is responsible for orchestrating the multiple applications available. This implies setting up the remote Northbound API (e.g., REST web server) and maintaining in-memory applications. Second, it defines how the *Network View* (*view*) is shared between applications. This *view* is updated by the applications, and used to process events such as the triggered by the data plane. Third, it is also in charge of the data plane communication, which is attained through the Southbound API (*SOUTH API* in the figure). Again, this API can be event-driven whereby events are commonly topology information (e.g., link up, link down) or forwarding requests (e.g., flow requests in OpenFlow).

Finally, the network devices responsible for packet forwarding reside in the **Data** plane. Any device can be used (wireless access point, Ethernet Switch, router), as long as it implements the Northbound API[6]. We commonly refer to these elements as switches regardless of the functions they perform in standard mode (i.e., without the intervention of a controller).

There must be connectivity between the Control and the Data plane. This connectivity can be **in-bound** or **out-bound**. In the in-bound case, the connectivity takes place over the network used for data forwarding, whereas in the out-bound case a different and isolated network is used. Connectivity between these two layers require manual configuration of the network devices.

The West/Eastbound API exists only when distributed control is employed (see section 2.4). In that case, the controller uses this API for state distribution and controller-to-controller communication. Again, it has two flows of communication for functions performed by one controller on another controller instance, or to receive asynchronous events from other controller (in event-driven implementations).

Finally, we note that SDNs can operate in two different modes[7]. In the **Reactive** mode a switch *reacts* to traffic that it does not know how to forward. It can react by sending a copy of the packet header to the controller and wait for forwarding configuration from the controller. The controller can then take action by replying to the switch with configuration updates (e.g., forward, drop, log). As shown in Fig. 2.2 the controller (commonly)

[6]In practice, a mixture of both Northbound enabled devices and normal devices is common.

[7]This discussion is based on the OpenFlow protocol, covered in the following section.

updates the switch with rules for both the packet and subsequent packets that will follow it (for the flow). In contrast, in a **Proactive** mode the switch does not bothers the controller with forwarding requests. Instead, the controller *proactively* configures the switch to handle all the traffic that it could possibly receive. This configuration is based on the existent network goals and configuration. Whenever the goals or the network state changes (e.g., link goes down) then the control plane updates the configuration of the switches. The drawback of a reactive mode is that a switch must send one or more packets (in a flow) to the controller, and possibly hold packets in a buffer. These are all tasks that consume the switch resources (memory and CPU), and significantly affect the time to forward the traffic. The drawback of a proactive mode is that it may require occupying significant space in the switches forwarding table (since all the possible traffic is anticipated). Additionally it limits the array of functions that a controller can perform. We note however that the current SDN technology is pliable enough to support both modes simultaneously (i.e., the **Hybrid** mode).

2.2.2 OpenFlow

OpenFlow (OF) is the most common implementation of the Southbound API shown in Fig. 2.1, and the only one we consider throughout this document. Thus, in order to make this document self-contained, it is necessary to detail some technical aspects of the OF protocol.

Table 2.1 shows a representation of an OF table present in the switch. This table is used by the controller to define the forwarding rules for each packet in the network that passes through the switch. As opposed to common devices, which are restricted to a small set of network packet headers (e.g., MAC addresses for switches, IP addresses for routers), OF switches are able to match packets against 13 different headers (that can be combined with logical operators). Furthermore, some headers such as the IP and MAC fields can be matched against bitmasks, which allows a wide range of values for a single table entry[8]. As an example, the table shows that the last two entries match against any host present in the $10.0.0.0/24$ network. Beyond headers, the switch can also match packets according to the port on which they arrive. We note that conflicting matches (i.e., a packet matches more than one entry in the table) are arbitrated by the priority associated with the rule (not show in the table).

For each entry, there is an associated instruction that the switch uses to forward the packet whenever it is able to find a match. Several instructions are available including: forward to port x, forward to controller, and drop the packet. As seen in the table, the instructions can be combined (the first entry forwards to two different ports).

Each entry is removed from the table once one of two private timeouts expires. There is a hard timeout, which is never reset, and an idle timeout that is reset whenever a packet

[8]This is equivalent to routers that perform the longest prefix match on the IP field.

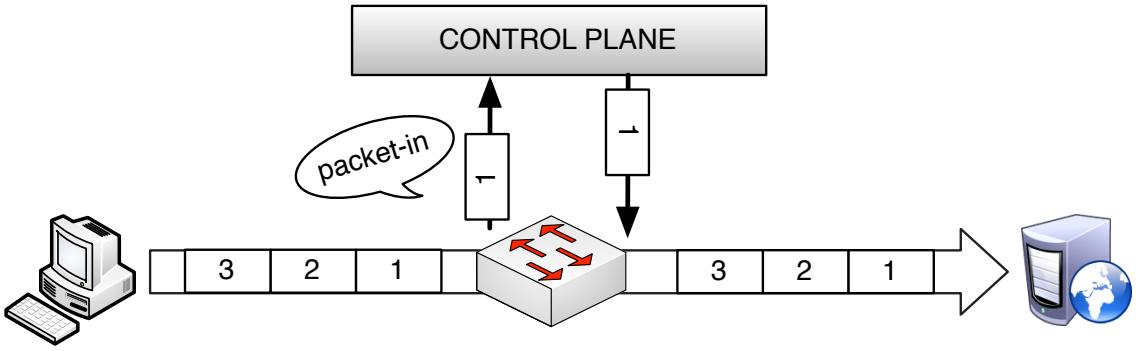


Figure 2.2: Commonly, in reactive mode, the first packet of a flow for which there is no match is forwarded to the controller, which evaluates it and decides the appropriate action. Normally the switch flow table is modified, such that the packets that follow (i.e., other packets for the same flow; packets 2 and 3 in the figure) do not have to go to the controller.

Match Fields	Instructions	Timeouts
source MAC = 10:20:. *AND* protocol = ICMP	port 2,3	5,10 ms
source IP = 10.0.0.0/24	port 1	0/0
source IP = 10.0.0.0/24 *AND* protocol = TCP	controller	0/0
any	controller	0/0

Table 2.1: Simplified representation of a flow table in OpenFlow switches.

is matched against the entry. The controller recycles and controls the entries in the switch table using both timeouts. The switch can be configured to send `flow-removed` messages to the controller whenever an entry expires. It is also possible that a rule is persistent (never expires). In this case, the timeout will be set to 0.

An OF switch table can be configured to forward non-matching packets to the controller (e.g., the last entry in the table). In reactive mode, the switch forwards the first packet of a flow (stream of one or more packets in a communication) to the controller. To exemplify, a flow could be all the packets sent from an HTTP client to a server to download a web page, or all the Internet Control Message Protocol (ICMP) packets sent to a host.

Fig. 2.2 illustrates how the switch processes packets in reactive mode. The first packet (1) for some "flow" (e.g., download of a web page) reaches the switch but is redirect to the controller since there is no rule that matches it in the switch table. We call this message a flow request or `packet-in` (the original name in the protocol). This request must contain (at least) the packet header from packet 1 that will be used by the controller to determine the reply. For this reason, we usually abuse the language and say something like "an IP packet arrives at the controller". Getting back to our example, when the controller receives a flow request, it processes it and then replies to the switch with instructions on how to process this packet. Commonly, the controller will use a `flow-mod` to configure the switch table such that future packets for this flow are acted upon accordingly to the

switch (1 and 2 in the figure). However, a controller can also forward the packet directly without changes to the switch flow table, or even inject packets in the data plane (e.g., to reply to an ARP request[9]).

The OF protocol can support multiple connections to different controllers. In the context of our work, it suffices to known that each switch can have one *master* controller, and multiple *slave* controllers. Only the master controller can affect the switch table rules, whereas all controllers can receive messages from a switch such as `packet-in` requests. This behavior can be modified. Namely, slave controllers can specify the type of events that they want to receive from each switch. A slave controller can be promoted to master by sending a message to the switch. However, there can only be one master, thus each switch processing a `role-change` request from a slave controller to master demotes the current master controller to slave.

2.3 Centralized Controllers

A simplified view of the architecture of centralized controllers is shown in Fig. 2.3. In this architecture, the controller supports multiple in-memory applications listening to different events that are triggered by the data plane. For each event e_x (e.g., link up, new switch, flow request), the switch sends an OpenFlow (OF) message to the controller, which parses the message to analyze its type and triggers an event to the appropriate pipeline. A pipeline is composed of several stages. In each stage, a different application processes the event and chooses to pass it further along in the pipeline, or aborts event processing. At any stage in the pipeline, an application can configure the switch with explicit OF configuration messages. We note that this architecture commonly presents a tight-bound between the management and data plane since the Northbound Application Programming Interface (API) exposes the Southbound API through OF events.

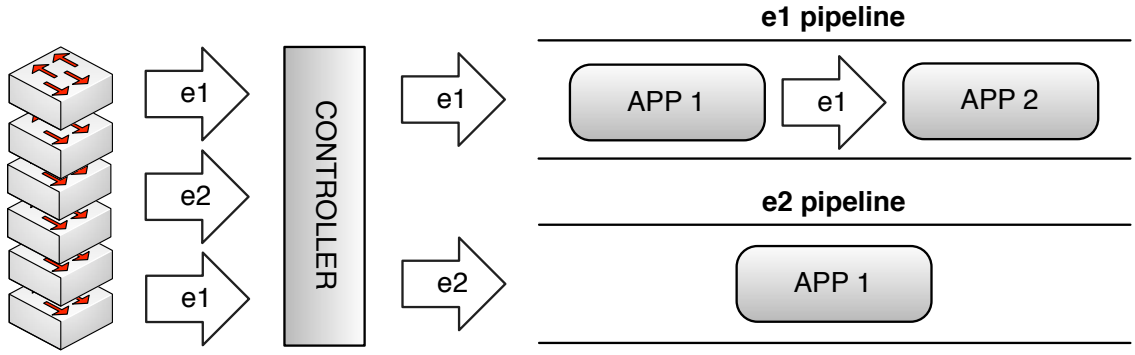


Figure 2.3: The common (meta) controller architecture is event-driven. The data plane triggers events (e1, e2) that are dispatched by the controller to a pipeline of interested applications.

[9]The Address Resolution Protocol (ARP) is used to translate Internet Protocol (IP) addresses into Media Access Control (MAC) addresses, which is essential for communication in Ethernet networks. Hosts broadcast ARP requests to find out the MAC of other hosts.

2.3.1 NOX

NOX[10] was published and publicly released under the GNU Public License (GPL) in 2008 [35]. It was developed in both C++ and Python, and enabled a standard interface for the integration of management applications in the controller. Even though one of the main contributions of the paper is the introduction of the Network Operating System (NOS) abstraction (see section 2.1.5), NOX itself is tightly bound to the OF API.

The NOX programming model is event-driven, as shown in Fig. 2.3. Applications register in a priority-based pipeline with event handlers associated to either OpenFlow or application based events. The NOX core is equipped with applications that build a host and switch level topology in its *Network view* (*view*).

In NOX, applications define network policies (i.e., control the network), and cooperate to define the current *view*. The *view* can consist of the switch-level topology; the location of users, hosts, middleboxes and other network elements; and the services (e.g., HTTP, FTP) being offered. The authors' point out that this choice of granularity for state is adequate for many network applications, since it changes slowly enough that it can be scalable. They also argue that the *view* can be distributed with standard replication techniques, for resilience. In fact, the *view* is a set of indexed hash tables with support for local caching. In this regard, the paper vision is very similar to ours, but has not been effectively implemented.

Initially, NOX was a single threaded application not focused on performance. However, from its publishing date several improvements have taken place that have significantly improved NOX performance [17]. Under the set of improvements, we highlight the natural evolution to a multi-core design that statically binds each thread to one or more switches. In the time of writing, NOX is publicly available but ramifies into two different controllers: A C++ based controller available in Linux and a Python based controller (POX) available in multiple Operating Systems.

2.3.2 Maestro

A NOS has (at least) two functions: introduce a layer of abstraction between the network and the applications and control the interaction between applications. Maestro, focus is in the second [37]. In particular, its authors' recognize that management components (i.e., the applications) do not operate independently and in isolation. Instead, they operate concurrently but with inter-dependent state. With this in mind, Maestro exploits parallel computing techniques to enhance the control plane performance.

Maestro splits the regular pipeline execution such that it can be concurrently executed (as NOX-MT). As seen in Fig. 2.3, events may follow different execution paths since singular applications are not interested in every single event. Three major design goals

[10] `http://www.noxrepo.org/`

shape Maestro: fair distribution of work across cores; minimal overhead introduced by cross-core and cache synchronization; and minimal memory consumption. In addition, it exploits throughput optimization through batching. The results published show that Maestro linearly scales the throughput with the number of cores available on the controller.

Currently Maestro is available[11] under the LGPL 2.1 license. It ships with the usual switching and routing capabilities.

2.3.3 Beacon

Beacon[12] is an open source controller built in Java by David Erickson, during his academic studies in Stanford University [38]. Beacon is also based on the event-driven model shown before. Applications register for specific type of events, and process these in the order configured by the user. Any application processing an event chooses to forward the event further in the pipeline, or terminate its execution. It is also multi-threaded, binding switches to particular threads.

Applications in Beacon are implemented as *bundles*. A bundle is the unit of abstraction in the OSGI[13] framework — a component and service platform for the Java programming language with dynamic capabilities — allowing features such as *hot-swapping* (i.e., deploy, start and stop modules in run time). Beacon provides a central service for registration of bundles as services (the registry). Each bundle implements a service, exports it to the registry and other bundles may consume it. Application events take place through the service abstraction: bundles may register in other bundles as listeners to be notified when specific events take place.

2.3.4 Floodlight

Floodlight is an open source Apache licensed controller. It is developed by an open community mainly composed of Big Switch[14] employers. It is written in Java, but applications can be implemented in either Java, Jython or any other language through the REST API available.

Floodlight follows the common event-driven programming model of the previous controllers. Although Floodlight was originally forked from Beacon, the OSGI support was abandoned for performance and deployment reasons. The overall functionality is based on modules (i.e., applications) that implement services that can be consumed by other modules. It is similar to Beacon in this regard, however the module/service functionality is directly provided by Floodlight instead of delegated to a third-party framework.

[11] `https://code.google.com/p/maestro-platform/`

[12] `http://goo.gl/pqMqh` [stanford.edu]

[13] `http://www.osgi.org/Main/HomePage`

[14] An SDN vendor with a commercial distributed controller named Big Controller. See `http://goo.gl/s4X53q` [bigswitch.com]

Floodlight is also multi-threaded. It accomplishes this through an asynchronous event based multithreaded library named Netty[15] that manages input/output communication with the managed switches.

We choose Floodlight as the base for our distributed controller study for two reasons: the programming language and the variety of applications already developed to it and made publicly available. The programming language — Java — is a priority since it simplifies the integration with the replication library that we use to build a consistent data store (see section 3.1.2). In addition, Floodlight has a very active community of developers and users.

2.4 Distributed Controllers

In this section, we present an overview of relevant distributed control planes architectures. A distributed controller is, by our definition, a controller that uses more than one controller to administrate the data plane. According to our general architecture (see section 2.2.1), the controller has an *East*/*West* Application Programming Interface (API).

The control planes that we will cover are focused in scalability, simplicity, and generality. They employ different techniques to manage the distributed access to the network state, and the distribution of the state itself such that it may become resilient to failures (in some cases). A common characteristic between all of them is that the data plane is partitioned across different controllers such that event processing can be distributed across different machines.

2.4.1 Kandoo

In 2012, Yeganesh et al. presented Kandoo [15] — a hierarchical controller for scalable Software Defined Network (SDN) infrastructures. The main contribution comes from the deployment of isolated controllers near the switches that shield a parent controller from processing all flow requests triggered by the network. It is implemented in a mixture of C, C++, and Python and is not publicly available.

In Kandoo — see Figure 2.4 — the control and management plane is split in two levels: in the top level resides the root controller, responsible for normal operation; in the bottom level, local controllers that are located closer to the managed switches (they may even be implemented in the switches themselves). This design is motivated by the idea of bringing specific control functionality towards the data plane for performance and scalability reasons.

Global applications reside in the top level and operate on of the global *Network View* (*view*), whereas *Local* applications reside in the bottom level and operate without direct access to the *view*, since they should be able to take decisions without it. Thus, specific

[15] `http://netty.io/`

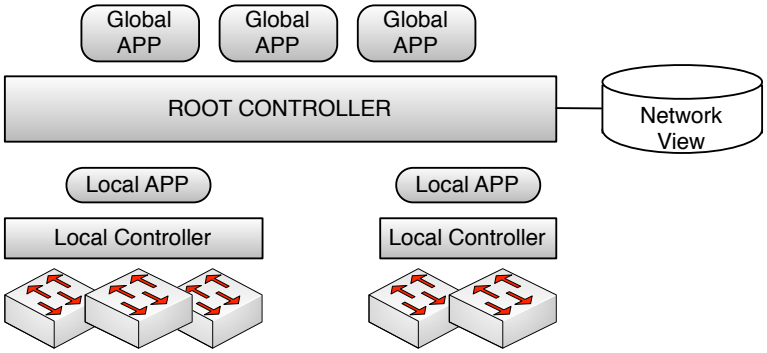


Figure 2.4: Kandoo decomposes the control plane into in two levels. The root controller and global applications have access to the global network state (*view*). The local controller and applications remain close to the data plane and do not require access to the global state.

events can be processed by local applications without the root controller collaboration. These events benefit from a lower latency penalty. However, if a local application decides that a particular event either affects the *view*, or requires accessing it, then the application can redirect the event to the root controller.

In theory, Kandoo can effectively scale the control plane. However, the paper does not specify the class of applications that can operate with local state only. This is something still under investigation.

2.4.2 HyperFlow

In 2010, Tootoonchian et al. introduced HyperFlow, a distributed SDN controller [4]. This work was motivated by the lack of scalability in centralized controllers. The authors aim was to provide scalability without sacrificing the simplicity that centralized controllers offer when building network applications. It is built as a *C++* application on top of the NOX controller [35], requiring minor modifications to the controller applications. HyperFlow is not publicly available.

Fig. 2.5 shows HyperFlow overall architecture. The control plane is distributed over different controller instances (or replicas). Each instance maintains, under its administrative domain, the connection to a subset of the data plane. HyperFlow supports the crash-recovery fault model where controllers can fail (e.g., crash, disconnect) and recover later on without affecting the network state.

In HyperFlow, each controller instance contains a replica of the entire network state. To replicate state, HyperFlow intercepts any event that causes changes to it. In such case, it distributes this event to other controllers that also process the event in the application pipeline. This way — and assuming that all controller instances run the same set of deterministic applications — all controller instances eventually arrive at the same state (if we assume a sufficient long period with no pending events). The benefit of having a view of the entire network is that each controller can take decisions based on local state.

Figure 2.5: HyperFlow architecture. The two major components are the controller which intercepts events and configuration commands that must be distributed, and the Publish/Subscribe system used for all communication between controllers.

Still, the controller has to communicate with others to distribute state changing events and configuration directives (i.e., commands) to switches controlled by other instances.

The inter-controller communication system is based on the well-known *Publish/Subscribe* model, whereby one defines publishers as senders of messages and subscribers as the receivers. Publishers do not send messages directly to receivers. Instead, messages are published in a medium, and interested receivers selectively receive them through some form of subscription logic. This model allows the decoupling of both space and time in the communication between publishers and subscribers. The authors have implemented a *Publish/Subscribe* system (on top of a distributed file system dubbed WheelFS [39]) that provides: persistent storage of the events published, *FIFO* (First-In-First-Out) ordered delivery of the messages published by the same controller, and resilience against network partitioning.

HyperFlow addresses scalability of the control plane by filtering the number of events that an instance replicates to others. To this end, it forces applications to tag local events that affect the network state. Furthermore, each application that triggers an event (e_2) must associate the event (e_1) that has caused it. Then, the distribution of e_1 is enough since applications in remote controllers will also trigger e_2 when processing e_1. As one single event can cause several application events to be triggered this can reduce the volume of traffic in the Publish/Subscribe middleware. Thus, HyperFlow is focused on the scalability of the CPU. It does not address state scalability since each controller contains the entire network state.

One of the advantages of HyperFlow is that the existing centralized applications are barely modified in order to work in such distributed model. Anyway, applications do have to account for the restrictions of the underlying distribution system. Otherwise, network state might diverge. Notice that even with FIFO based channels some controller c might receive event e_i followed by event e_j (sent from controllers i and j), while a controller c' perceives e_j followed by e_i. If the change of state is dependent of the ordering of such events (i.e., operations are not commutative) then the state from c and c' will be different. Consistency problems can thus emerge.

Another benefit of HyperFlow is that under read intensive workloads the controller

can achieve low response times since events are processed locally. However, in our experience with centralized applications (as the ones that HyperFlow targets), read intensive workloads are not too common. In fact, Chapter 3.2.5 shows that all three applications we analyzed show at least one write operation for each event processed in the controller.

2.4.3 Onix

Onix improves on the NOX legacy with multiple contributions [3]. It provides an improved controller architecture on which the Northbound API does not reflect the Southbound API (the first to do so). Management applications are programmed against a network graph very similar to the Object Oriented paradigm and are not aware of the Southbound characteristics (e.g., the use of OpenFlow). The graph is implemented in the Network Information Base (NIB) — an in-memory data structure that contains the network state — and can be distributed across a cluster of Onix controllers. Applications have the choice to specify consistency and durability requirements per network entity present in the NIB. Onix was the result of a joint effort between Google, NEC, Nicira, ISCI, and Berkeley, and (at least) both Google and Ericson have developed their controllers from Onix [14, 40]. Thus, it is the first production level controller. At the time of writing Onix is not publicly available.

Onix architecture is presented in figure 2.6. The NIB is the only element in Onix Northbound interface. The management layer directly modifies the NIB and subscribes to changes on it. The data plane indirectly modifies the NIB (through the controller). The controller has to guarantee that changes in the data plane are reflected in the NIB and vice-versa. For this, it translates network events into changes in the NIB and changes in the NIB to changes in the data plane configuration. Onix supports OpenFlow (OF) but it could transparently move to another Southbound API.

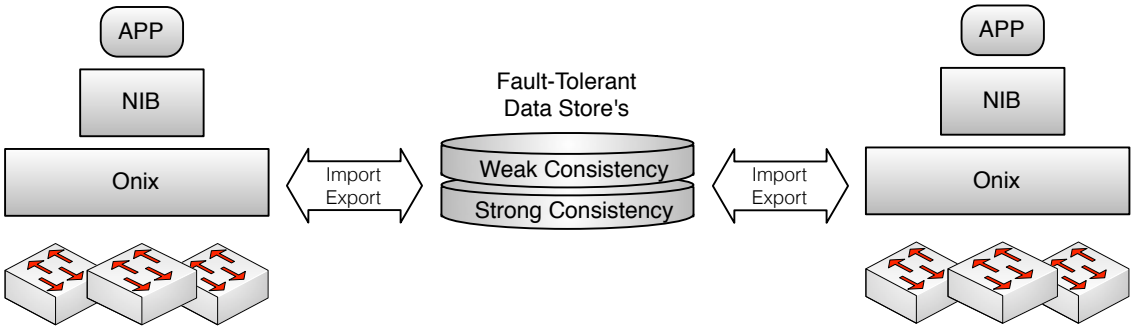


Figure 2.6: **Onix achitecture.** The NIB is an in-memory network graph of typed entities (i.e., objects in a statically typed language). It is supported by two replicated data store's accessible across Onix instances. The data store's interface is bidirectional, the controller exports information from the NIB to the data store and vice-versa.

Each Onix instance independently manages a subset of the data plane. However, each instance can be exposed to the entire network state through the NIB. Under such scenario,

each time a local Onix instance alters the NIB, these changes are reflected in all other instances. Thus, the NIB is also the distribution mechanism of the Onix controller. The distribution itself is done by the data stores that back up the NIB state (seen in figure 2.6). However, in concept the NIB is not built to contain all the network state. In fact, the NIB may be best seen as a kind of in-memory cache which imports (exports) particular state from (to) the fault-tolerant distributed data stores.

Onix defines a flexible distribution model for the NIB whereby it offers the application designer the choice of consistency guarantees. Two replicated data stores are present: one offers strong, another eventual consistency. Strong consistent is supported through a transactional persistent database backed up by a replicated state machine (see section 2.5). This data store is favored for data with low-frequency events (e.g., topology changes) as its performance limitations are significant. The eventually consistent data store consists in a memory based Distributed Hash Table (DHT) (as Dynamo [22]) favored for volatile data with high update rates. Both the integration of multiple data stores and the inconsistency characteristics of the DHT can lead the NIB to an inconsistent state where reads performed in some entity may return more than one result. Thus, Onix provides primitives for the integration of inconsistency resolution logic as well as direct integration of a distributed coordination framework (such as Apache' Zookeeper [21]) in the Northbound interface.

Onix is intended for large-scale network infrastructures where scalability is fundamental. However, in each Onix instance the NIB size reflecting the network state could lead to memory exhaustion; and the processing of both network events and subscriptions to changes in the NIB can lead to CPU exhaustion. Thus, the management layer must employ *partition* and *aggregation* techniques in order to guarantee scalability. Partition avoids full replication of both data and workload such that additional instances do not only replicate overall work but also relief it (e.g., hiding inaccessible network servers from other controllers). Aggregation combines several entities in the NIB of a controller before exporting them to the data stores' (e.g., a collection of switches appears to be a single "big" switch).

The application only interacts with the NIB graph data structure that is composed of *Typed Entities* supporting the Object Oriented paradigm (i.e., encapsulation of data, functions over entities, hierarchy, etc.). Onix supports extensible representations of network entities. The NIB API provides essential functions to search, inspect, create, destroy, and modify the network entities present in the NIB. It is also possible to register notifications for creation, removal, and updates of data entities. When network events of other Onix instances update the data stores those changes can be reflected on the local NIB and the application can be notified through a callback function. All operations are asynchronous, with eventual delivery and no ordering or latency guarantees given. Therefore, a *barrier* synchronization primitive is available allowing the application to wait as updates are translated and applied in the network devices and/or other controllers.

Onix is focused in generality; and it does a great job at it. In fact, the authors emphasize that "when faced with a tradeoff between generality and control plane performance we try to optimize the former while satisfying the latter". This choice is revealed in the strong consistent data store. One of our main goals is to show that this performance limitation is merely a performance characteristic of the particular implementation considered in the paper.

2.5 Consistent Data Stores

The key idea of our distributed controller architecture is to make multiple controller instances coordinate their actions through a dependable data store in which all relevant state of the network and of its control applications is maintained in a consistent way. This data store is implemented with a set of servers (replicas) to avoid any single point of failure, without impairing consistency.

Our data store provides one of the strongest consistency models known in distributed systems: linearizability [41]. This model is completely transparent for the client of the system: it is indistinguishable from a centralized system (i.e., non-replicated). Such strong consistency is achieved at the expense of performance and, more importantly, availability, since it requires a majority of the system's replicas to be accessible. In contrast, the eventually consistency model relaxes such constraints in order to improve availability and performance. This model has multiple variations and lacks a clear definition. Loosely, it is a particular form of weak-consistency, grounded on a liveness guarantee: eventually the replicas of a system converge to the same state [39, 42]. Sections 2.5.1 to 2.5.3 further clarify the differences between these two models.

One of the most popular techniques for implementing a strongly consistent data store is State Machine Replication (SMR) [5, 43]. Practical fault-tolerant replicated state machines are usually based on the Paxos agreement algorithm for ensuring that all updates to the data store are applied in the same order in all replicas (thus ensuring strong consistency) [43]. However, since the original Paxos describes only an algorithmic framework for maintaining synchronized replicas with minimal assumptions, we instead describe — in section 2.5.4 —- the Viewstamped Replication (VR) protocol, a similar (but more concrete) state machine replication algorithm introduced at the same time [44].

State Machine Replication is a fundamental technique that is used by real world systems in a wide array of applications. Furthermore, and contrary to popular believe, it can be used in systems where availability and performance are crucial requirements. To support this claim, in section 2.5.5 we give an overview of state of the art SMR based systems.

2.5.1 Trade-offs

With the proof of the CAP theorem in 2002, by Gilbert and Lynch, a significant interest arisen for eventual consistent distributed systems [27]. This theorem establishes that a distributed system can only be qualified with two of the following properties: consistency, availability, and partition-tolerance. In this context, consistency is equivalent to a system supporting linearizability; availability is equivalent to the guarantee than any request performed on the system eventually receives a reply; and partition-tolerance implies tolerating arbitrary message loss. Since networks often exhibit message loss, systems are forced, in practice, to choose between availability and consistency.

However, in practice systems are not this black or white. One of the outcomes of the dichotomy established by CAP is a panoply of systems with different consistency models that attempt to position themselves at a different optimal point between the availability and consistency spectrum. Due to space constraints, we cannot cover this topic in depth and instead defer the interested reader for surveys on the topic [25, 30, 42].

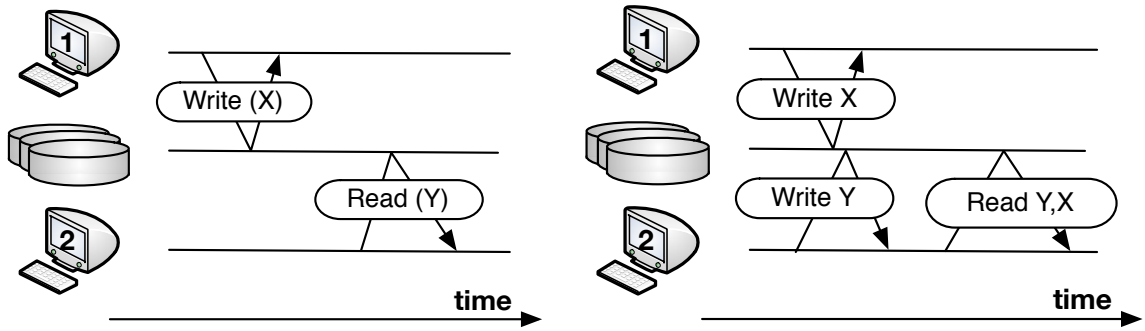
2.5.2 Eventual Consistency

Under the eventual consistency model, when a write to the data store is finished, the replicated system guarantees that, at some point in time, all replicas will converge to the same value (in the absence of pending client operations). A benefit of this model is that it is straightforward to implement. A replica can accept client requests, process them, reply to the client, and only later spread the outcome to other replicas. The process of spreading values to others is commonly known as anti-entropy and can take several forms, but is in essence an asynchronous process (i.e., it only takes place *"some time"* after replying to the client).

Several undesirable outcomes come out of this model. For example, consider a replicated system with three replicas and a single register containing some arbitrary value. There are two clients of the system: client 1 and client 2 that can both perform write and read operations on this register. Fig. 2.7 shows the system composition with two different histories of client requests.

First, Fig. 2.7a shows that after a write operation clients may see older values of the register. Namely, client 1 performs an update on the data store ($write(X)$), and later client 2 stills sees the older value of the register (Y). The time between a write being applied in a data store replica and all clients being able to see that value is named the **window of inconsistency**. A recent paper on eventually consistent systems states that this window of inconsistency varies between 200 milliseconds to 15 seconds in different types of systems [30].

It may be the case that for some clients of the data store, the window of inconsistency does no represents a problem. However, another subtle problem can emerge from the



(a) Stale data: reading an old value.

(b) Conflicting data.

Figure 2.7: Two clients of a replicated data store experiencing some of the eventual consistency model pitfalls. In the stale data case Client 2 is not able to see the last write done to the data store. As for the conflicting data case the concurrent update done by two clients causes the data store to become "confused" as to what value it should keep. Consequently, it keeps both.

asynchronous anti-entropy process — there is no guarantee that an update will reach other replicas. Consequently, clients cannot be certain that the outcome of operations such as $write(X)$ will eventually be seen by other clients. This is possible in the case the server that process the update fails before spreading it to other replicas. If so, either he recovers and the update was stored on disk, or the update is lost forever. Truly, eventual consistency ensures convergence only if replicas survive long enough to spread updates to others[16].

The last problem we identify in eventual consistency systems is conflicting values caused by concurrent updates. Fig. 2.7b shows two clients concurrently updating the register. Consider (not shown in the figure) that two different data store replicas (out of three) process and reply to the clients independently. Then, the anti-entropy process takes place and each updated replica spreads the update to the others. The question that must be answered then is the following: how can the replicas decide which update should persist between the two: $write(X), write(Y)$? Eventually consistent systems diverge in how they arbiter conflicts but, in practice, either the system solves the conflict "arbitrarily", or it applies conflict resolution logic provided by the user.

In general, eventually consistent data stores are better equipped to deal with volatile data that does not have stringent requirements such as consistency and reliability. Notwithstanding the evidence that systems can also be equipped to deal with the problems mentioned above, they do so at the expense of possible incorrectness, or the extra effort required from the developer. Still, this extra effort can be justified, in systems where scalability and performance is crucial as in the case of Dynamo, an eventually consistent data store used by Amazon's core services [22]

[16]Getting back to the consistency spectrum comment performed in the previous section: some eventually consistent systems could update more than one replica before replying to the client, thus obtaining a higher reliability.

2.5.3 Strong Consistency

The most significant advantage of the strong consistency model is that developers can relax their concern towards the semantics of the replicated system since a strong consistent system is not any different from a centralized (single site), un-replicated system.

The strong consistency property is formally introduced as linearizability in a seminal paper by Herlihy et al. [41]. Informally, this property establishes that every operation performed on the replicated system appears to have effect instantaneously. The practical consequence is that clients can be made sure that after receiving a reply from the system for an update operation, every other client will be able to see the effect of that write. In practice, this condition is true as soon as the data store processes the client update, but it makes more sense, from the client point of view, to base this assumption only as soon as the reply arrives since it has no other mean to determine that the data store has processed his operation. Fig. 2.8 exemplifies this condition.

With strong consistency, clients can rest assured that their operations survive failures as soon as they receive a reply from the system. To exemplify, the SMR technique that we use (covered in the following section) blocks client requests until they are processed by a majority of replicas. Given that this technique assumes that there will always be a majority of replicas "alive" and reachable, the client is made sure that the aforementioned operation will never be lost in the data store.

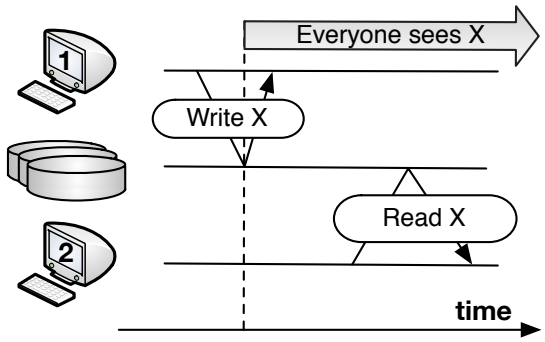


Figure 2.8: Strong Consistency Semantics. A client can be sure that every client sees its operation outcome as soon as the data store replies.

2.5.4 ViewStamped Replication

The Viewstamped (VR) protocol provides a transparent State Machine Replication (SMR) behavior that can be leveraged to implement a replicated data store with strong consistency semantics [44]. SMR allows a set of machines to provide a replicated service, usually to support availability and reliability characteristics in the system [5]. Informally, the model states that a distinguished set of machines who start in the same state and execute the same set of instructions (in the same order) achieve the same final state. SMR is

suitable for distributed applications, such as file systems, lock managers, and data stores which can be built on top of the distributed state machine.

The algorithm is built over a partial-synchronous model (i.e., the system eventually progresses) with a crash-recover failure model for the participants (*replicas*). In practice, this failure and timing model establishes that replicas may stop (due to crash) and later recover, perform computations arbitrarily slow, and/or be isolated from the rest of the group. The remaining group suspects the replica has failed as a reaction to its lack of feedback. VR allows replicas to re-join the group as long as they are updated on the missed executions by others. The group is composed of $n = 2f + 1$ replicas, assuming that only f replicas can fail simultaneously. Thus, as long as $f+1$ replicas (i.e., a majority) are accessible the protocol guarantees liveness (which translates to availability from the client point of view).

VR is based on *total-order*, an ordered delivery guarantee between a set of machines communicating in-group. The protocol is structured around the notion of a leader (a distinguished replica named primary) who chooses and imposes the order of delivery of executions on the group. This approach has the drawback of overloading the primary with client requests but simplifies the protocol. Fig. 2.9 shows the messages exchanged in VR for an update operation: the client sends a message (`Request`) to a primary replica (the leader) that disseminates the update to all other replicas. These replicas write the update to their log and send an acknowledgment message to the primary (`PrepareOk`). In the final step, the leader executes the request and sends the reply to the client. If the primary fails, messages will not be ordered and thus a new primary will be elected to ensure the algorithm makes progress. When read-only operations are invoked, the leader can answer them without contacting the other replicas. Strong consistency is ensured because the leader orders all requests before executed by each replica.

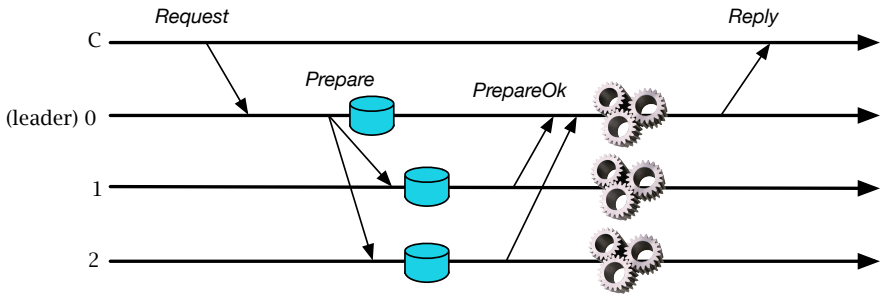


Figure 2.9: The normal execution of the ViewStamped Replication protocol. Each client request must be acknowledged (PrepareOk) by a majority of the replicas before being executed.

This short description covers only the normal execution of the protocol. The interested reader is forwarded to the original work for a description of the details on re-electing a new leader, and for recovering failed replicas [44].

2.5.5 State Machine Replication Performance

The Paxos/VR algorithm has served as the foundation for many recent replicated (consistent and fault-tolerant) data stores, from main-memory databases with the purpose of implementing coordination and configuration management (e.g., Apache' Zookeeper [21]), to experimental block-based data stores or virtual discs [45–47], and even to wide-area replication systems, such as Google Spanner [23]. These systems employ many implementation techniques to efficiently use the network and storage media.

Although not as scalable as a weakly consistent data store, these systems grant the advantages of consistency for a large number of applications, namely those with moderate performance and scalability requirements. To give an idea of the performance of these systems, Table 2.2 shows the reported throughput for read and write operations of several state-of-the-art consistent data stores.

System	*Block Size*	*kRead/s*	*kWrite/s*
Spanner [23]	4kB	11	4
Spinnaker [45]	4kB	45+	4
SCKV-Store [47]	4kB	N/R	4.7
Zookeeper [21]	1kB	87	21

Table 2.2: Throughput (in thousands data block reads and writes per second) of consistent and fault-tolerant data stores based on state machine replication (N/R = Not Reported).

Given the differences in the design and the environments where these measurements were taken, we present these values here only as supporting arguments for the possibility of using consistent data stores for storing the relevant state of SDN control applications. Interestingly, these values are of the same order of magnitude of the reported values for *non-consistent* updates in Onix (33k small updates per second considering 3 nodes), and much higher than the reported values for their consistent data store (50 SQL queries/second) [3]. The Onix paper does not describe how its consistent database is implemented but, as shown by these results, its performance is far from what is being reported in the current literature.

2.6 Consistent Data Planes

A couple of recent works on Software Defined Network (SDN) consistency has appeared recently. However, these works target consistency at a different level. In essence, they target consistent flow rule updates on switches, dealing with overlapping policies and using atomic-like flow rule installation in SDN devices. The aim of both these systems is to guarantee consistency *after* the policy decision is made. Our work provides a different type of consistency: one that is important *before* the policy decisions are made (as with Onix, but we want better performance also).

2.6.1 Abstractions for Network Updates

Reitblatt et al. introduce two abstractions capable of preserving well-defined behaviors while updating the data plane configuration [31]. In their work they show, that the transition of data plane forwarding rules from one state to another may lead to violation of network policies and disturb network services.

To avoid such pitfalls, the authors ague that the update of the data plane configuration should be done step-by-step (rule by rule) in each switch until completion, while making sure that any intermediary state of the network configuration is valid. Reitblatt et al. proposed an abstraction that is capable of transitioning the network from one state to another while maintaining the following property: **per-packet consistency** — "every packet traversing the network is processed by exactly one consistent global network configuration". This is important since it guarantees that packets are processed by the initial policy, or the final policy but not an intermediary (potentially invalid) policy. To implement such abstraction the authors suggest a mechanism based on stamping packets entering the network with version numbers corresponding to the network policy used to take the forwarding decision. Additionally, the authors build on this abstraction to provide **per-flow consistency**, an property analogous to the previous, but this time applied to flows.

2.6.2 Software Transactional Network

Software Transactional Networking focuses in consistency policy composition in the context of a concurrent (i.e., distributed) control plane [48]. Contrary to the previous work, consistency policy composition deals with overlapping policy updates. To exemplify imagine two control applications that deal with forwarding and traffic monitoring. The forwarding application establishes that all packets addressed to a particular address such as 10.0/16 should follow a specific path along the data plane (i.e., a composition of different switches and ports), and the monitor application establishes that all HyperText Transport Protocol (HTTP) packets should be counted. Since the monitor application policy applies to a subset of the forwarding application policy, they must be composed. The only *consistent* solution to this composition is that all the HTTP traffic destined to 10.0/16 should be counted and follow the forwarding application specified path. The nitty-gritty of this composition is that in the switches, the rules installed by both applications must have the correct priorities in order to avoid, for example, the case where the HTTP traffic rule has a higher priority than the forwarding rule. The result would be that HTTP traffic to 10.0/6 would be counted but not forwarded since the policies did not compose.

In essence, this problem was already solved by Frenetic [2] in the context of a centralized controller (leveraging the central point to resolve policy conflicts and order policy updates). However, in STN the authors take this notion of consistent composition and

confront it with concurrent policy updates whereby different controllers that execute the same set of control applications can update and compose the network policy in different points in time, and with different ordering. The authors show that: *"without synchronization it is impossible at any single point in time for a particular controller instance to ensure that its policy specification does not conflict with that of any others"*. Indeed, given the asynchronous nature of the policy updates in each controller, conflicting policies can be applied to the data plane.

The authors distinguish between *weak-composition*, where policies updates can be composed arbitrarily but eventually reach the same final composition for every controller, and *strong-composition* where policies updates are composed in the same global order by independent controllers. Furthermore, they propose a transactional interface implementation, for the *weak-composition* case, whereby policy updates either commit or are aborted, and single packets are processed by a single configuration of the network (either the configuration before the update, or after the update, but not an intermediary configuration) In essence, this proposal is an atomic update of the network configuration that is susceptible to global conflicting policies caused by the *weak-composition* pitfall — different controllers can employ different policies at a single point in time. However, it does guarantee that eventually the network reaches the coherent, consistent, and correct state.

This work is fundamental to understand the pitfalls of distributed updates to the data plane. As observed by the authors, the strong-composition semantics of concurrent updates can be implemented by resorting to the state-machine-replication technique (as used by our data store) that could be used to impose a global order in the policy update. However, this will certainly be limiting the number of different policies and rate of policy update supported. We are not certain that this would be a serious problem since we do not have any evidence of specific requirements in real-world deployments (we plan to address this issue in the future). Finally, we note that the *weak-composition* technique proposed in the paper is orthogonal to our distributed controller design.

Chapter 3 – Architecture

All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection.

Kevlin Henney's

Before the appearance of SDNs, control functions such as routing required their own distributed protocol that would span over the data plane elements. With Software Defined Network (SDN), the logically centralized architecture enables control functions based on protocols that can leverage the logical centralization of the network state. However, as already covered in the previous chapters, this does not postulate a centralized system. In this chapter we present a distributed control architecture that relies on standard distributed techniques to preserve the primordial logical centralization abstraction of SDN.

3.1 Shared Data Store Controller Architecture

Building a distributed system is a complicated task. There are plenty of problems to address: failures are inevitable and must be considered, implementing the "right" consistency model right is hard; and testing can be unwieldy. Ideally, the development of control applications should not be exposed to such a complex environment. In traditional networks, control applications (e.g., routing protocols) have to solve the problem of distributing the system and solve the control problem. In this section we present a architecture that can mitigate the complexity of distributed systems.

The fundamental requirements that have driven our design are:

Transparency – the distribution of the system is invisible to applications;

Simplicity – the distribution of the system allows a centralized programming model;

Generality – the system should be as general as possible providing useful distributed constructs for the use of the client;

Reliability – the system should be prepared to handle failures;

Scalability – the system should anticipate scaling requirements.

The transparency and simplicity requirements protect the developer of network applications from the idiosyncrasies of a distributed system. Distributed systems can be built transparently offerring a centralized programming model whereby the user is only faced with the distribution in the event of particular network exceptions. The next requirement (generality) establishes the need of robust distributed systems tools such as locks, barriers, and leader election that are indispensable in a distributed context. We aim to provide the essential building blocks to develop these tools under the same Application Programming Interface (API) used for state distribution.

Reliability for a distributed control plane is an obvious requirement for production networks. As stated before, faults are the norm and not the exception in real-world applications. Thus, it is crucial for the system to handle failures from any of its composing components.

The insightful reader may found curious that we set scalability as one of our requirements given that we already established that, when faced with the tradeoff between consistency and scalability, we prefer the former to the latter. However, this does not rule out the consideration of scalability mechanisms.

3.1.1 General Architecture

We propose a novel Software Defined Network (SDN) controller architecture that is distributed, fault-tolerant, and strongly consistent. The central element of this architecture is a key value data store that keeps relevant network and application state, guaranteeing that SDN applications operate on a consistent network view, ensuring coordinated, correct behavior, and consequently simplifying application design.

The architecture is based on a set of controllers acting as clients of the fault-tolerant replicated data store, reading and updating the required state as the control application demands, maintaining only soft state locally. This architecture is data-centric — it is through the data store that we support distribution, and the data store, being strongly consistent, is versatile enough to satisfy most of the control plane requirements with the exception of scalability (when contrasted with eventual consistent data stores). The data store mimics the centralized shared memory model existent in concurrent centralized controllers such as Floodlight. Therefore, other controllers can easily be integrated as a component of our architecture (see chapter 4).

Fig. 3.1 shows the architecture of our shared data store distributed controller. The architecture comprises a set of SDN controllers connected to the switches in the network. All decisions of the control plane applications running on the distributed controller are based on data plane events triggered by the switches and the consistent network state the controllers share on the data store. The fact that we have a consistent data store makes

the interaction between controllers as simple as reading and writing on the shared data store: there is no need for code that deals with conflict resolution or the complexities due to possible corner cases arising from weak consistency.

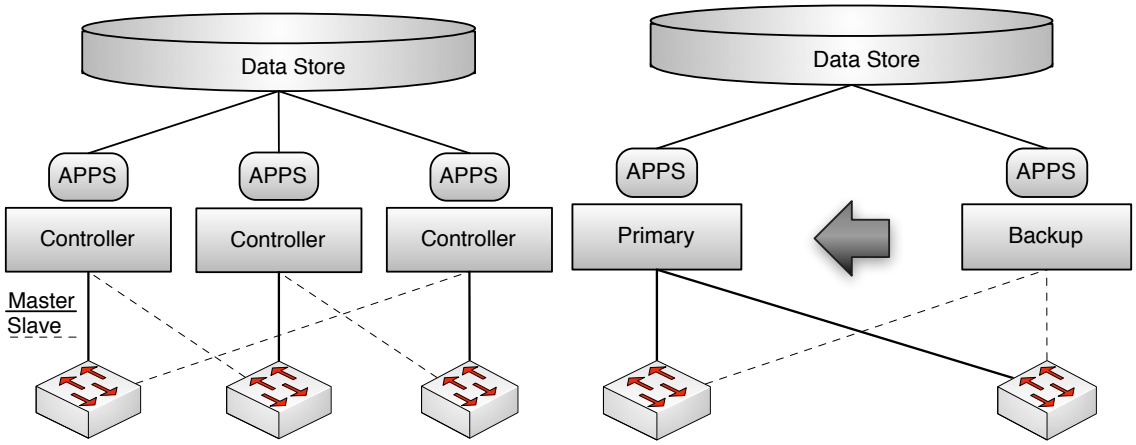
The control plane is stateless and cares only about processing the data plane events. The only state kept is soft-state[1], which can easily be reconstructed after a crash. The hard-state is kept in the data store. Thus, once a controller fails, any of the existent controllers can take over its place based on the network state that always survives in the data store. The switches can tolerate controller crashes using the master-slave configuration introduced in OpenFlow 1.2 [11], which allows each switch to connect to $f+1$ controllers (being f an upper bound on the number of faults tolerated), with a single one being master for each particular switch. The master is constantly being monitored by the remaining f controllers, which can takeover its role in case of a crash. Because the data store provides strong consistency, the controllers can leverage it to assure coordination while "fighting" over the switches in the network (e.g., leader election, locks, etc.).

Interestingly, our architecture can be used in two different models. Fig 3.1a shows that in the active model the control plane is distributed and each controller takes over a different subset of the network (coordinating through the data store). In this model, each controller can serve as master for a subset of a network and as slave for any other subset. Once a controller fails, any controller can take over. In this model, only the master controller processes the events of the switches.

The second model shown in Fig. 3.1b is the Primary-Backup. In this model, the control plane is "centralized" since only the primary controller controls the network while others stand by. Still, the fault-tolerant data store can be used to store the pertinent controller state, making it extremely simple to recover from the primary crash. In this case, the applications deployed on the primary controller manage the network while a set of f backup controllers keep monitoring this primary, just as in the active model. If the primary fails, one of the backups – say, the one with the highest IP address – takes the role of primary and uses the data store to continue controlling the network. In this model, the primary controller can contain in cache the frequently accessed data without impairing consistency since it is the only reader and writer of data. In such case, all writes can be performed on the data store, and reads can be performed locally (if available in cache).

Our distributed controller architecture covers the two most complex fault domains in an SDN, as introduced in [24]. It has the potential to tolerate faults in the controller (if the controller itself or associated machinery fails) by having the state stored in the fault-tolerant data store. It can also deal with faults in the control plane (the connection controller-switch) by having each switch connected to several controllers (which is ongoing work). The third SDN fault domain — the data plane — is orthogonal to this work since it depends on the topology of the network and how control applications react to

[1] State used for efficiency such as precomputed tables in hashing algorithms.



(a) Active (b) Primary-Backup

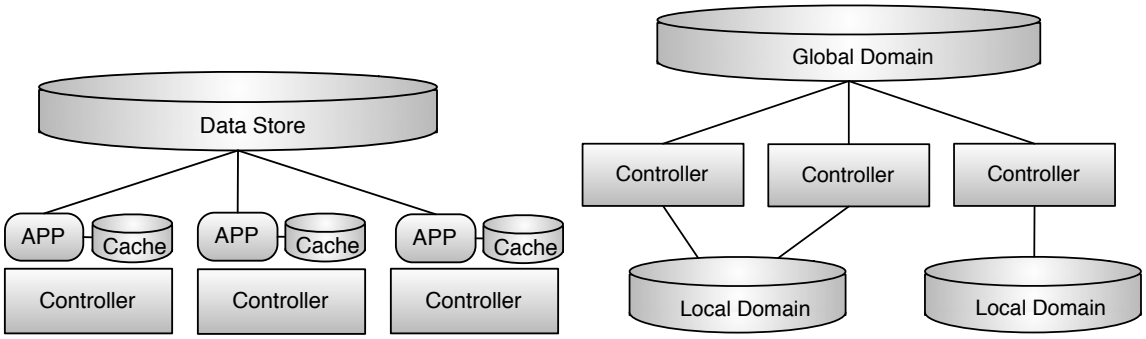
Figure 3.1: General Architecture: The controllers coordinate their actions using a logically centralized data store, implemented as a set of synchronized replicas (see Figure 2.9). The architecture comprises two models. In the Active model, each controller is actively responsible for a subset of the network. In the Primary-Backup model, a single controller is active, and another is prepared to take its place in case of failure.

faults. This problem is being addressed in other recent efforts [24, 49].

To increase the overall scalability and generality of this architecture we consider, from the outset, two fundamental components: cache and domains. Cache offers latency benefits at the expense of consistency and memory space; domains offers scalability and latency benefits at the expense of more hardware.

Fig. 3.2a shows each controller with a cache that may store a frequently accessed subset of the data store data, thus enabling local read operations that do not experience the latency penalty of the data store. In order to be coherent with our design — that favors consistency — it is of the utmost importance for the cache to be exposed as a functional component to the applications that reside on the controller. In order words, it is the clients that have explicit control over the values present in cache and whether each operation allows a cached value or not. For this purpose, we propose a time-based cache validity scheme (see section 3.2.5).

Fig. 3.2b shows that, in a multi-domain setting, each domain is a single data store instance in isolation, and controllers can connect to multiple domains. This setting is inspired by Kandoo [15]. Our proposal is to have a single domain for global information that is made accessible to every controller. In Wide Area Network (WAN) SDN deployments the global domain data store could use resilient protocols optimized for that environment [50]. Then, multiple local domains could be positioned closer to the controllers exploiting the amount of data shared between them (say, in a single building of a campus). This scheme can improve the latency of the overall system since frequently accessed data is closer to the controllers. Additionally, it can also improve the global throughput since the data store is partitioned across multiple systems. With this configuration, controllers



(a) Cache (b) Domains

Figure 3.2: Performance and Scalability improvements. The cache is a subset of the data store present in the controller memory thus enabling local read operations. Domains allow the data store to be partitioned across different configurations enhancing scalability.

can selectively choose how their network state is exposed to others. To clarify, in the global domain they can expose an aggregated topology view of their own network subset (i.e., more succinct), while in the local domain they can keep the entire topology.

3.1.2 Data Store

Our architecture orbits around a data store that is replicated to guarantee fault tolerance, and strongly consistent to guarantee transparency. As previously covered (in section 2.5), a solution that covers all these requirements is the State Machine Replication (SMR). In this section, we explain how we exploit this technique to implement our data store.

Fig. 3.3 shows the architecture of our data store. In order to be fault tolerant, the data store is composed of a set of servers (replicas) that are initialized with the same state. Then, for each client request (e.g., read, write) the *SMR* component is responsible for running an ordering protocol between the different replicas that ensures that all replicas receive requests in the same order. An example of such protocol is Viewstamped Replication [44], which was specified in section 2.5.4. When the protocol finishes, the *SMR* component transfers the request to the *Data Store* component responsible for processing it. When the *Data Store* finishes it can reply to the *SMR* component, which in turns replies to the client. In the meanwhile, the *Data Store* moves to process the next request. If the *Data Store* actions are deterministic, this technique ensures that all replicas achieve the same state (in the absence of pending client requests[2]).

We use the Byzantine Fault-Tolerant State Machine Replication (BFT-SMaRt) — an open source Java-based library for state machine replication — to implement the *SMR* component. Among other things, this library supports a tunable fault model, durabil-

[2]Due to the asynchronous nature of the model (network and computational), replicas can process the same requests in different moments of time. Replicas only achieve the same state when they have processed the same set of client requests. Such moment is guaranteed in the absence of pending client requests.

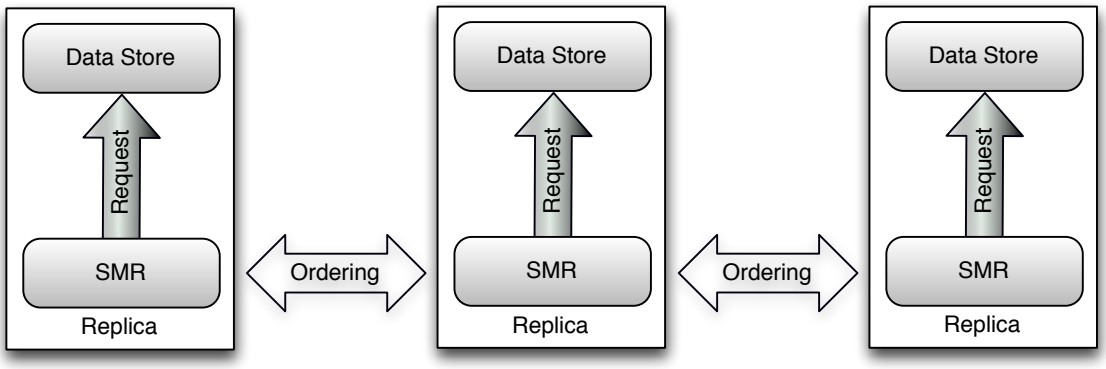


Figure 3.3: Data Store Architecture: each replica is composed by two main components: BFT-SMaRt, responsible for the ordering protocol; the Data Store responsible for the structure of data (e.g., tables, columns), and processing of client requests. Each client request that reaches a replica, is ordered by the BFT-SMaRt protocol which guarantees that every Data Store execute requests in the same order, thus achieving a coherent state.

ity, and reconfiguration. The fault model can be either Byzantine[3] or crash-recovery. For performance reasons, we consider the crash-recovery model whereby a process (i.e., replica) is considered faulty if either the process crashes and never recovers or the process keeps infinitely crashing and recovering [51]. The library operates under an eventually synchronous model for ensuring liveness, which informally states that at some point in time, the system progresses (i.e., computations finish and messages get delivered). For durability, a state transfer protocol guarantees that state survives the failure of more than f replicas (the number of replicas that can fail simultaneously). Finally, BFT-SMaRt possesses a reconfiguration protocol that allows the system to shrink and grow in run-time.

Our BFT-SMaRt-based data store is therefore, replicated and fault-tolerant, being up and running as long as a majority of replicas is alive [43]. Under partition scenarios where replicas can be temporarily isolated from each other either due to network partitions or more obscure conditions that inhibit the replicas from participating in the protocol, at least a majority (i.e., half plus one) of the replicas must be available in order to guarantee progress. More formally, $2f+1$ replicas are needed to tolerate f simultaneous faults (e.g., a cluster of three servers supports one fault).

BFT-SMaRt also enables transparency via strong consistency. Namely, it guarantees linearizability (i.e., an operation appears to execute instantaneously, exactly once, at some point between its invocation and its response) [41]. The reader may easily understand how this is achieve by our system, since each ordered operation is in fact executed in isolation (as explained above). Therefore, our data store is in congruence with concurrent shared memory systems. The user of the data store is not aware of its distribution unless: (i) the network connection to the majority of replicas that compose the data store fails; or

[3]In a Byzantine fault model, processes can deviate from the protocol in any way. Namely, they can lie, omit messages, and crash.

(ii) the performance penalty of the network is noticeable (by measurement). Otherwise, it behaves just as a centralized system (it is transparent).

3.2 Data Store Prototype

In this section, we present the details of the key value data store prototype that we have built to evaluate the feasibility of the architecture propose before. The implementation of this prototype required two components: the data store server (i.e., *Data Store* component of Fig. 3.3), and the data store client (i.e., the client code necessary to interact with the data store). To connect the two, we developed a simple Remote Procedure Call (RPC) protocol over BFT-SMaRt. This protocol establishes the format of the messages sent such that the data store is able to identify the appropriate operation and extract the relevant arguments (i.e., keys, values, etc.).

We started by developing simple key value data store where the client can define an arbitrary number of tables (uniquely identified by their name). Each table is a key value table (i.e., an hash table) mapping unique keys to an arbitrary value (i.e., raw data). The server has no semantic knowledge of the data present in the data-store and supports simple operations such as create, read, update, and delete[4].

Fig 3.4 shows the class diagram for the client side interface with the data store. The reader familiar with Java will quickly identify that from the client perspective the data store client interface is similar to existent constructs (i.e., the *java.util.Map* interface). In production code we could use any of the multiple off-the-shelve data stores (either SQL or NoSQL) available, given that they offer functionalities akin to the ones that we present in this section. However, in the context of our study (in chapter 4) it was simpler to use these constructs.

As the figure shows all the interfaces are statically typed. We relied on the client to provide marshalling and unmarshalling code[5] for each table *instance*. Our initial version of the data store (i.e., the key value) relied only in the *IKeyValueTable* interface. However, our data store prototype was iteratively refined to incorporate new data store functions (and interfaces), required to increase the performance of Software Defined Network (SDN) applications.

A brief overview of the interfaces identified in the figure follows[6]:

[4]This was enough for the purpose of the study performed, but we intend to explore the possibility of building or incorporating richer query languages.

[5]The process where values are transformed into byte streams (marshalling) and vice-versa (unmarshalling).

[6]For the sake of brevity, we simplify the inheritance design of the interfaces. To conform to a proper separation of concerns each table extending an *ICrossReferenceTable* should have a counterpart interface extending an *IKeyValueTable*.

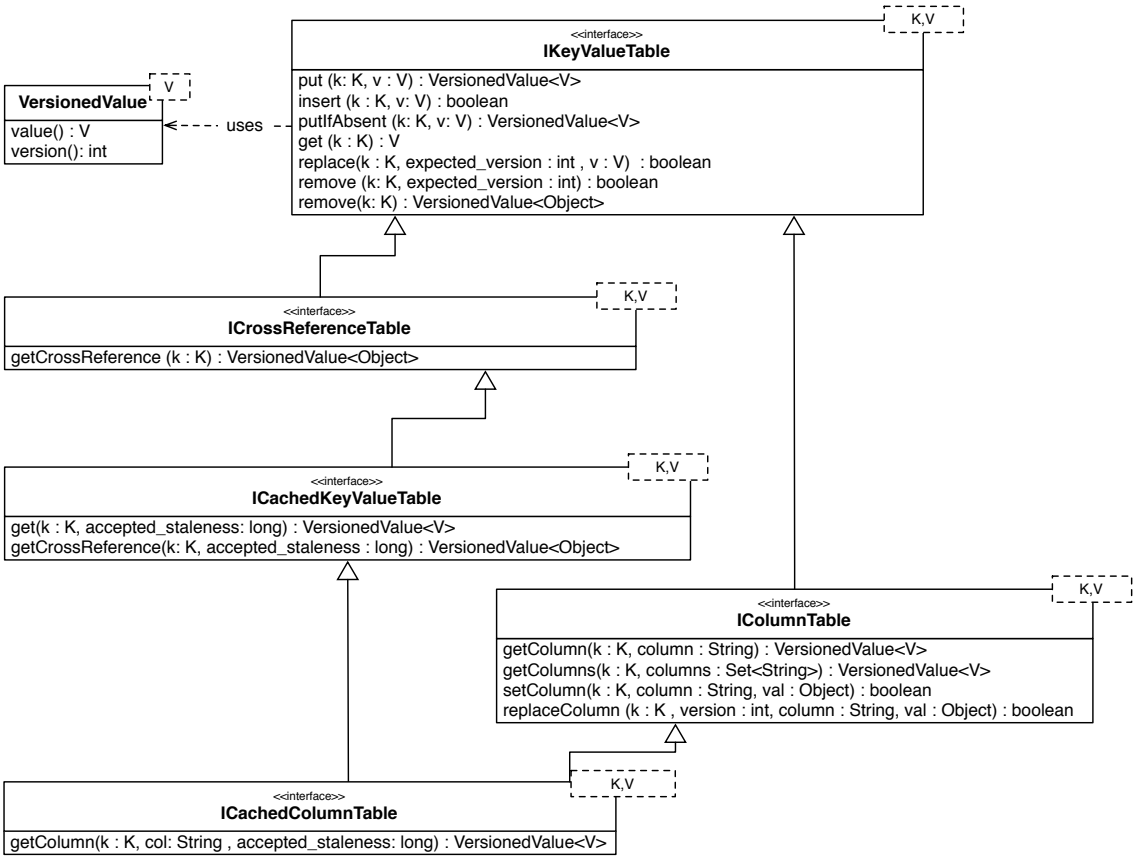


Figure 3.4: Class diagram of the client interface to data store tables.

- *IKeyValueTable* is a key-value (i.e., hash table) interface. You can manipulate the key-to-value association in different ways (create, update, remove, replace);
- *ICrossReferenceTable* extends an *IKeyValueTable*. It is used for tables that contain values that can be used as keys in another table (section 3.2.1);
- *IColumnTable* is the extension of an *IKeyValueTable* into a bidimensional table where two keys access an individual value (section 3.2.3);
- *ICachedKeyValueTable* handles caching of the entries retrieved/updated from/to the data store. In addition it allows explicit control over the window of inconsistency accepted in cached entries (section 3.2.5);
- *ICachedColumnTable* does the same as the previous interface, but for an *IColumnTable* interface;
- *VersionedValue* is a container for versioned values obtained from the data store (section 3.2.2).

The remaining of this section further clarifies the functions of those interfaces.

3.2.1 Cross References

Commonly, a hash table is restricted to a single key to identify a value despite the number of unique attributes that are associated with the value. Furthermore, the asymptotical complexity to obtain a value with a particular key is $\mathrm{O}(1)$ as opposed to searching for one, which at best has $\Omega(\log n)$ complexity for n entries (using balanced trees). To circumvent those limitations, one can use an additional table that relates a "secondary" key of a value to its "main" one. As an example, imagine that for the purpose of tracking hosts in a network we consider that a device is uniquely identified by an IP or MAC address. Therefore, we could use two tables: table `IPS` relating IPs (key) to MACs (value), and table `MACS` relating MACs (key) to devices (value). This is a reasonable scheme in a local environment (in memory hash table) given that the asymptotical cost to obtain a device with a MAC address or its IP is equal ($\mathrm{O}(1)$). However, in a distributed environment, this requires two round trips to the data store just to obtain a single device with an IP address (one to fetch the MAC, and another to fetch the device), thus incurring in a significant latency penalty.

To address this problem (i.e., two round trips to obtain a single value) we implemented a Cross Reference table (*ICrossReferenceTable* interface in Fig. 3.4), which is able to obtain the device with a single access to the data store. Fig. 3.5 clarifies how our Cross Reference table works. In this example, the client (*controller*) configures the `IPS` table as a cross reference to the `MACS` table. In practice, this is understood as: the values of the `IPS` table can be used in the `MACS` table. With this setting, the client can fetch a device from the `IPS` table, with a single data store operation (the *getCrossReference* method). Thus, this operation reduces in half the latency penalty required to obtain the device.

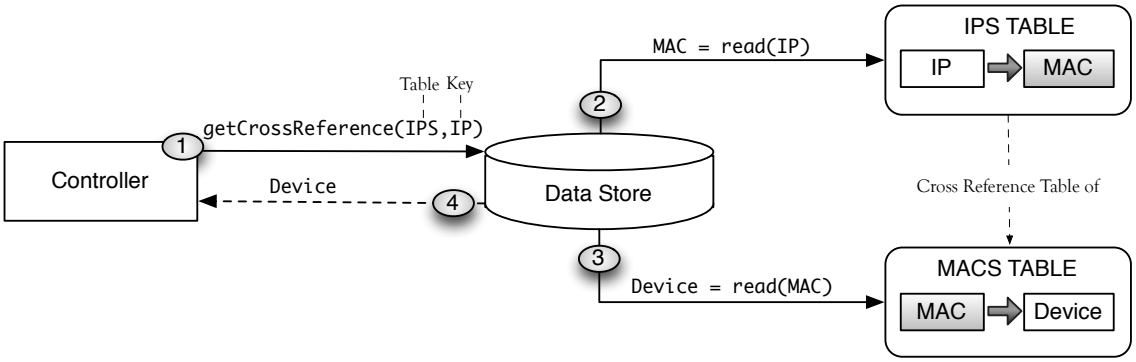


Figure 3.5: Cross Reference table example with Table *IPS* configured as a cross reference to table *MACS*. First, the controller sends a cross reference read request to the data store for table `IPS` and key *IP* (1). Then, the data store performs a read in table *IPS* to obtain the key *MAC* (2), that is used in table `MACS` (3) to finally reply to the client the *Device* (4).

3.2.2 Versioning

Despite being strongly consistent, our data store is still exposed to the pitfalls of concurrent updates performed by clients. Namely, the loss of data caused by overlapping writes. To clarify, imagine an HTTP network logger running in a controller that maintains a key-value table (in the data store) to map each Uniform Resource Locator (URL)[7] observed, in the network packets, to the set of IP addresses that have visited it. In other words, whenever a host visits a web site, the controller adds the IP address of that host to that URL set. However, for the data store a set is merely an opaque binary object. Hence, the controllers need to fetch the set, add an element locally, and finally write the new set in the data store[8]. If two controllers do this concurrently, it is possible to loose values added to the set.

Fig. 3.6a shows an example to clarify this further. First, controllers 1 and 2 fetch the same `visitors` set for a particular web site (uniquely identified by the URL), and then they replace it by a new set that includes IP2 and IP3 respectively. In this case, the lack of concurrency control results in the loss of the write operation that includes the IP2 visit to the site (visitors=IP1,IP2) since the last write (visitors = IP1, IP3) overwrites the previous.

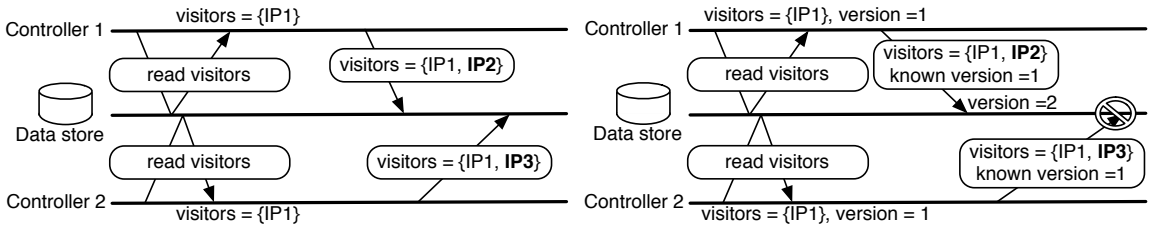
With Versioning, each table entry (i.e., key value pair) is associated with a monotonically increasing counter — the version number — that is incremented in every mutation operation. Doing so, we empower the data store with the capability to detect and prevent conflicting updates that otherwise could result in the loss of data.

Fig 3.6b shows that with Versioning, the write from controller 1 results in an increase of the version number of the visitors set at the data store (to 2), which prevents any update done by controllers unaware of the most recent version of the set. Therefore, by the time the second write request (from controller 2) arrives at the data store it is aborted, since the version number included in the operation is not consistent with the data store. We dub this kind of write operation as *conditional write*. It succeeds only if the version included by the client matches the version present at the data store.

Whenever the data store denies a request (as in the example above) the client can only repeat the entire process since, in order to complete its write, it must obtain the most recent version from the data store. It is important to realize that the data store has no mechanism to guarantee that a stubborn client will eventually succeed. Indeed, it is possible that one client loops indefinitely if another client constantly out-wins him in every write attempt. Clients are solely accountable for guaranteeing the progress (liveness) of their updates. This process is commonly termed of *Optimistic Concurrency Control* [52]. Clients are optimistic in the sense that they hope that no one else updates the value while they perform

[7] A uniform resource locator, also known as web address, constitutes a reference to a resource such as a web page or an email.

[8] This can be considered a limitation of the data store, since it could support append operations to sets, however the problem stills exists for arbitrary raw data.



(a) No Versioning. (b) Versioning.

Figure 3.6: Both figures show the history of interactions in a system composed of a data store and two controllers (1 and 2). Time flows from left to right. The concurrent update to the `visitors` set for a particular site can result in loss of data. In Fig. 3.6a, the update from controller 1 is forgotten when replaced by the update from controller 2 (last-write-wins). Conversely, in Fig 3.6b, the use of versioning in the data store prevents controller 2 from overwriting the last update.

the entire process (read, modify and write).

Concerning implementation, the Java common library provides a concurrent hash table[9] with concurrent control primitives equivalent to the ones we include in our data store Application Programming Interface (API). However, the control is based on the logical equivalence of objects (as established by their `equals` contract[10] [53]) instead of version numbers. That is to say, instead of providing the version number in a conditional write, the client must provide the value that it expects to find in the hash table. Then, the hash table implementation can perform a logical test to assert if the client provided value is logically equivalent to the one that it holds. If so, the write is allowed, otherwise it is "aborted".

While adapting existing applications to our API that used the Java concurrent hash table, we have chosen to modify them slightly to use the version number mechanism instead of the logical equivalence method refereed to above (using method *replace* in *IKeyValueTable*). We have done so for two reasons. First, while equivalence tests work well with objects, the same is not true for the raw bytes that result from the marshalling process. In fact, we found cases where two objects where logically equivalent, but their byte representation was disparate. Second, with version numbers we reduce the size of messages (sent to the data store) significantly.

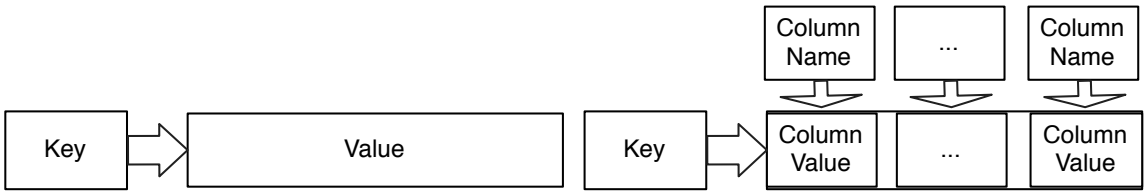
3.2.3 Columns

With a key value data model (Fig. 3.7a), clients are able to map a unique key to any arbitrary value with no syntactical meaning for the data store (it is just raw data). This is a quite limited data model since values are often composed of multiple attributes. Consequently, when the client interest lies towards a small portion of the value (e.g., a single attribute), this model can be a bottleneck, since both the update messages (sent to the

[9] *java.util.concurrent.ConcurrentHashMap* ; see `http://goo.gl/avpkVb` [oracle.com]
[10] `http://goo.gl/5ZVbE` [oracle.com]

data store), and reply messages (received from the data store) may contain unnecessary attributes (thus increasing the latency penalty for the client). Therefore, we expanded the key value table (Fig. 3.7b) to allow clients to access the individual components of a value with an additional key (i.e., the column name). With Columns, we enhance the unidimensional model of a key value table to a bidimensional one whereby two keys (as opposed to one) can access an individual attribute of a value inside a table.

Despite the fact that a Column table decomposes a value into columns, the client is still able to manipulate the entire value. In fact, the class diagram introduced before (Fig 3.4) shows that the client API for an *IColumnTable* inherits all the *IKeyValueTable* methods. Namely, the client is still able to retrieve or update a value "entirely" (i.e., a Java object) even if he is not aware of the column names that compose a value. Furthermore, the column names are not static, not even in the context of a table. Each key-value entry may have different columns, and clients can add and delete columns from a value as they see fit (in run-time).



(a) Key Value store. (b) Column store.

Figure 3.7: From a Key Value model to a Column model. In the Key Value model a key references an entire value (raw data from the data store viewpoint). As for the Column model, the client can still manipulate the entire value (using only the key), or use two keys to address the value (key and Column name).

A column based table model is beneficial because reading an entire value introduces considerable overhead, as the message returned by the data store may be considerable big. To address this the client can create a column table, and obtain a selected set of columns (using the *getColumns* method in *IColumnTable*). Furthermore, before introducing the column model, the updates performed by the clients on the data store required the entire value, despite the number of attributes changed (remember that attributes are meaningful for the client, not the data store). Again, this can result in considerable big messages. To solve this, the client can use the *replaceColumn* method in *IcolumnTable* to replace a single column inside a value, thus reducing the size of the update message when compared to updating the entire value. This operation is also conditional (see previous section).

3.2.4 Micro Components

So far, we have focused in particular client use cases (i.e., multiple keys to obtain a value, concurrent updates, and manipulation of attributes) to introduce techniques that reduce

the number or size of messages in the client-to-data store interaction. However, for an arbitrary number of operations that have no explicit connection to each other we need a more general and powerful abstraction. To exemplify imagine that the client intends to: *"read two value from different tables"*, *"add them"*, *"update other table with the result"*, and *"return the result to the client"*. Clearly, with the current interface, this set of operations will require multiple client-to-data store interactions, thus revealing a significant latency penalty for such a simple task. To address these types of problems, we introduce the Micro Components functionality.

Micro Components are equivalent to the stored procedure functionality existent in Structured Query Language (SQL) databases. Truly, they are a powerful abstraction that can be used to implement any of the features seen before. In essence, a micro component is an arbitrary long method that is executed in the data store and triggered by the client. This method has semantic knowledge of the data that is contained in the data store. It knows what to do with the data kept in the data store, which implies that it knows the marshalling and un-marshalling process used for the tables that it manipulates. The most significant advantage of a micro component is performance since it allows the client to merge multiple operations in a single method. This diminishes the latency impact that the data store has in the client.

In our prototype, we developed multiple micro components that were statically (i.e., prior to compilation) included in the data store along with the Classes that each micro component required to operate (i.e., we included the client code in the data store server). This is undesirable, mainly because it forces the re-deployment of the data store code in order to add new micro components. We plan to address this issue in the feature.

3.2.5 Cache

With a Cache table (*ICachedTable*), the client can keep frequently accessed values locally, for a particular data store table. In this table, each value that is read or written from and to the data store is added to the local cache. Consequently, each read performed by the client *can* avoid the latency penalty in the data store interaction. However, the client has the choice to reject cached values if he wishes so, since we explicitly provide the Cache interface (*ICachedKeyValueTable* and *ICachedColumnTable*) as a functional element of our interface design, for which the client has absolute control. Thus, the cache is not an implementation detail hidden beneath the data store interface. To clarify, the client is able to define if he accepts a cached value as well as the bound on the window of inconsistency that he is willing to tolerate. We note this can only be strictly guaranteed in a synchronous system model[11].

[11] A synchronous model specifies a known limit on processing and message delivery delays. Under other models clock errors and undefined time limits on computations can result in the client obtaining a value outside the specified window of inconsistency bound

In order to define the inconsistency bound, the client can use the `get` method in an *ICachedKeyValueTable* that requires an argument (`accepted_staleness`) defining the upper time bound for accepting the value present in the cache. Then, for each client request, the cache returns the local value, if it has been added to the cache within the time specified by the client (i.e., the time passed between adding the value to the cache and the current time is lower than the `accepted_staleness`). Otherwise, the cache retrieves the value from the data store. This is shown in Fig. 3.8. It is worth pointing out than if the bound specified by the client is 0, then the cache must forcibly fetch the value from the data store (hence providing consistency).

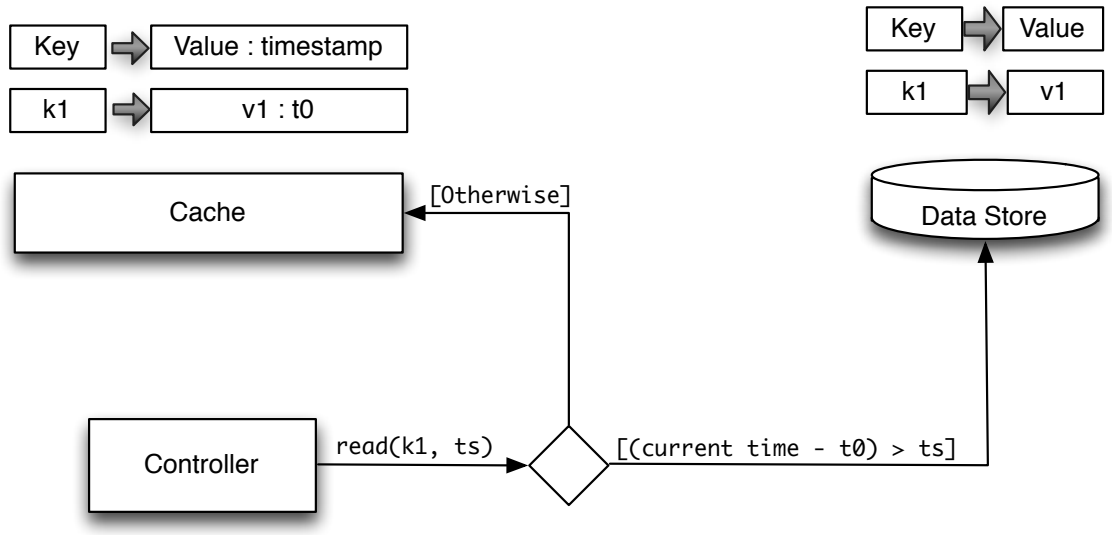


Figure 3.8: Reading Values from the Cache: the client performs a read on the data store for key *k1* and accepted staleness *ts*. The cache returns a local value iff: it was added to the cache for less than *ts* time. Otherwise, it obtains the value from the data store (and updates the cache).

To be fair, the use of cache may (arguably) break the transparency characteristic of our design. However, it does not need to break the consistency semantics if the clients wish so. Moreover, this design still has multiple advantages when compared to an eventually consistent data store. First, clients have the freedom to choose whether they are willing to accept a possibly stale value present in cache or a consistent value retrieved from the data store. Furthermore, they have explicit control over the window of inconsistency that they are willing to accept. Second, as long as writes are performed consistently (i.e., in the data store) there is no risk of conflicting values. Finally, clients still have a strong consistency data store on which they can rely upon to evaluate the consistency of their cached values.

The previous point is of fundamental importance. Clients can always evaluate the staleness of their cached values in the data store. To clarify, imagine some task that involves reading a value from the data store, modify it, and update the data store. If the value read is present in cache, then the client can save a trip to the data store, and evaluate

the staleness of the cached value on the write. To evaluate staleness, the client can use the Versioning technique, introduced in section 3.2.2, to ensure that the cached value used is in fact consistent with the data store version. Thus, a task composed of two round-trips to the data store may be performed with a single data store request.

Chapter 4 – Evaluation

There are only two hard things in Computer Science:
cache invalidation, naming things and off-by-one errors.

Variation of Phil Karlton quote

To evaluate the feasibility of our distributed controller design we implemented a prototype of the previously described architecture by integrating applications from the Floodlight controller[1] with the data store built over a state-of-the-art State Machine Replication (SMR) library, BFT-SMaRt [6] (which was described in the previous chapter). We considered three SDN applications provided with Floodlight: *Learning Switch* (a common layer 2 switch), *Load Balancer* (a round-robin load balancer), and *Device Manager* (an application that tracks devices as they move around a network). The applications were slightly modified to use the data store efficiently (i.e., always trying to minimize communication) instead of the controller volatile memory.

Our main goal was to analyze the workloads generated by these applications to thereafter measure the performance of the BFT-SMaRt library when subject to the realistic demand caused by real applications. We measured our data store focusing in BFT-SMaRt, as it is the bottleneck of our architecture due to the expensive consistency and fault tolerance guarantees.

In the remaining of this chapter we introduce the methodology to evaluate our system in section 4.1. Then, we report our results for each individual application, from section 4.2 to Section 4.4. Finally, we report our results for caching in section 4.5.

4.1 Methodology and Environment

To evaluate our design, we consider each application in isolation. Namely, we focus in the workload that they apply to the data store. A workload is a trace (or log) of data store requests that result from processing a data plane event by a particular application.

Fig. 4.1 shows that whenever a switch triggers some message to a controller application, the latter uses the data store for an arbitrary number of operations (e.g., associate the source address of the some flow request to the switch port at which it arrived, as in the Learning Switch application). Then, as soon as the application finishes, it may reply to the switch with a message (named "controller reaction" in the figure).

[1] `http://www.projectfloodlight.org/floodlight/`

In our evaluation we analyze the operations performed between the time that a data plane event is received at the controller and the time the application replies to the event. This log of operations (i.e., the workload) is later used to analyze how each individual application acts according to the different data plane events. In addition, we also consider changes in the workload caused by variations of the state present in the data store (returning to the Learning Switch example, the case when the source address of a packet is already "known" by the data store).

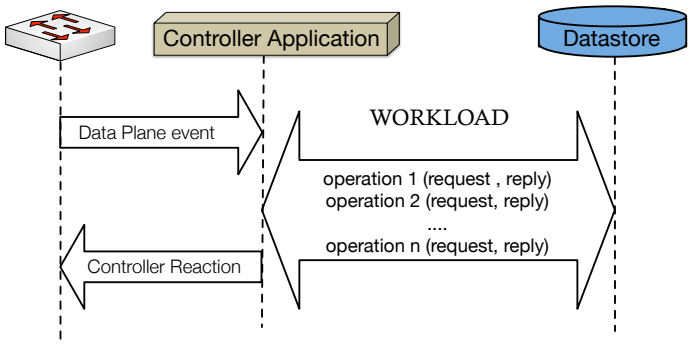


Figure 4.1: Each data plane events trigger a variable number of operations in the data store. The trace of those operations and their characteristics is our workload.

The workload analysis is thus performed in two distinct phases: first, we generate data plane traffic and record the respective workloads, and second, we use the workloads to measure the performance of our data store. The following two sections describe both phases.

4.1.1 Workload Generation

For the first phase of our study we emulated a network environment in Mininet [54]. Briefly, Mininet is a network emulation platform that enables a virtual network (a real kernel, switch and application code) on a single machine, and we use it to emulate the network devices (switches and hosts). As far as our study is concerned, we use it to trigger the appropriate OpenFlow (OF) data plane events messages sent from the switch to the controller (recall Fig. 4.1)[2].

Our network environment for each application consists of a single switch and at least a pair of host devices. After the initialization of the test environment (e.g., creation of a switch table, configuration of the Load Balancer application, etc.) we generated Internet Control Message Protocol (ICMP) requests between two devices. The goal was to create OF traffic (`packet-in` messages) from the ingress switch to the controller.

Then, for each OF request, the controller performs a variable, application-dependent number of read and write operations, of different sizes, in the data store (i.e., the *work-*

[2]We could have used an OF library to trigger the data plane events, but we choose Mininet given its simplicity as well as our familiarity with it.

load). In the controller (the data store client), each data store interaction is recorded entirely (i.e., request and reply size, type of operation, etc.) and associated with the data plane event that has caused it.

4.1.2 Data Store Performance

In this phase, the previously collected workload traces were used to measure the performance of the BFT-SMaRt-based data store.

As shown in Fig. 4.2, we set up an environment in a cluster composed of four machines: one for the data store client (that simulates several controllers), and three for the data store (to tolerate one crash fault ($f = 1$), wee need three replicas, as explained in sections 2.5 and 3.1.2).

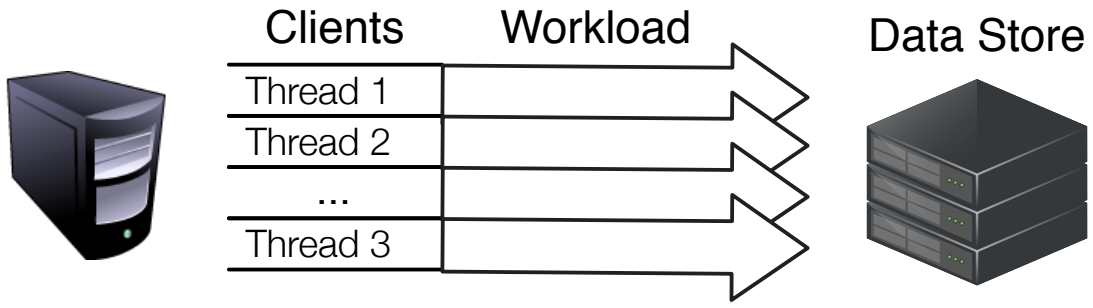


Figure 4.2: Illustration of the data store performance test. The machine at the left simulates multiple clients — equivalent to multiple controller applications — that replay the workload over several iterations. The data store is composed of 3 BFT-SMaRt replicas (see section 3.1.2).

The data store client runs in a single Java process, but executes multiple threads that replay a simulation of the recorded workload with an equal number of messages and payloads (i.e., same message type and size). We emphasize that in order to replay a workload composed of $op_1, op_2, ...op_n$ operations, a thread must send operation op_1, wait for a reply from the data store and, only after, send operation op_2 (and so on, until op_n).

In the simulation we use a special data store implementation for two reasons: $i)$ to reply to each client request with an message equivalent to the one recorded by the workload, and $ii)$ measure the number of workloads processed. Since the data store server is unaware of the workload composition it relies on the client to: include the expected reply size in each request, and signal the data store when all the operations of a workload complete[3].

This simulation is repeated for a variable number of concurrent data store clients (representing different threads in one controller and/or different controllers). From the measurements we obtain throughput and latency benchmarks for the data store under different realistic loads.

[3]This is why our reports show a decrease in performance, when compared to the existent BFT-SMaRt analysis [6].

Each workload was run 50 thousand times, measuring both latency and throughput. We calculated the average, minimum, maximum and standard deviation at the 90, 95 and 99th percentile. In this document we only show the averages at the 90th percentile. Appendix A (available online [55]) contains all the benchmark information (in graphical and raw format) and the traces (i.e., data plane events and respective workloads) for each workload shown in this chapter. We also made available the scripts that automate the data plane events in Mininet (used in our experiments), as well as the original codebase that can be used to reproduce all the work presented.

4.1.3 Test Environment

Each machine in the performance benchmarks had two quad-core 2.27 GHz Intel Xeon E5520 and 32 GB of RAM memory, and they were interconnected with a gigabit Ethernet. The software environment was Ubuntu 12.04.2 LTS with Java(TM) SE Runtime Environment (build 1.7.0_07-b10) 64 bits. For the applications, we used Mininet 2.0[4], a Floodlight fork[5] and BFT-SMaRt[6].

4.2 Learning Switch

The Learning Switch application emulates the hardware layer 2 switch forwarding process[7] based on a switch table that associates Media Access Control (MAC) addresses to the switch ports where they were last seen. The switch is able to populate this table by listening to every incoming packet that, in turn, is forwarded according to information present in the table.

Similarly, in the application, for each switch a different MAC-to-switch-port table is maintained in the data store. Each table is populated using the source address information (i.e., MAC and switch port) present in every OpenFlow (OF) `packet-in` request, which is used for maintaining the location of devices. After learning this location, the controller can install rules in the switches to forward packets from a source to a destination. Until then, the controller must instruct the switch to *flood* the packet to every port, with the exception of the ingress port (where the packet came from).

Fig. 4.3 shows the detailed interaction between the switch, Learning Switch application, and data store for the only two possible cases of an OF `packet-in` request. First, the case for broadcast packets that require one write operation to store the switch-port of

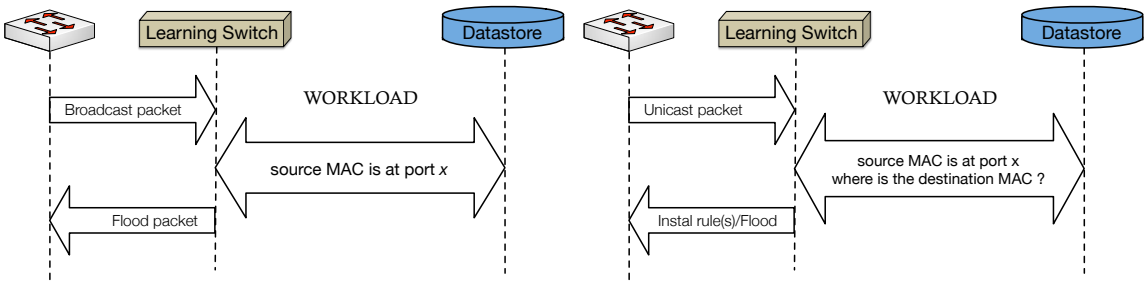
[4]Available at `http://mininet.org` (mininet-2.0.0-113012-amd64-ovf). We had an issue with this version and corrected it by following online instructions available at `http://goo.gl/DQ7FQF` [stanford.edu].

[5]`http://goo.gl/RbBXag` [github.com] commit 9b361fbb3f84629b98d99adc108cddffc606521f

[6]`http://code.google.com/p/bft-smart`, revision 334

[7]If the reader is not aware of how this process works do not worry, it is identical to the one we will describe for the Learning Switch application.

the source address (Fig. 4.3a). Second, the case for unicast packets, that not only stores the source information, but also reads the (possibly) known switch port for the destination address (Fig. 4.3b). If the port is not known, the packet is flooded through all the switch ports (with the exception of the incoming port).



(a) Broadcast packet. (b) Unicast packet.

Figure 4.3: Operations in the data store vary based on whether the OF `packet-in` is triggered by a broadcast or unicast packet.

It is critical, both for the original and the distributed versions of this application, that each switch table is limited in size due to resource exhaustion (each table can potentially keep an entry for each host present in the network!). For this reason the application limits a table to a fixed number of hosts (1024 by default). When this limit is reached the least recently used entries are replaced by new ones. This eviction policy favors (i.e., avoids deleting) active devices over inactive ones. Each access to the table (either a read or a write) promotes the key to the top of the list making it the most recently used. After the table is full, newly added entries replace the bottom entry of the same list (the least referenced).

The Least Recently Used (LRU) tables are not the only way to control the table entries. The Learning Switch application also applies timeouts (hard and soft — see section 2.2.2) to the flows installed in the data plane. When they expire, a switch triggers an OF `flow-removed` message (containing a source and a destination address) to the control plane that, in turn, deletes the associated entry from the data store. Consequently, the application constantly recycles both switch flow rules and data store entries.

4.2.1 Broadcast Packet

This workload is defined by the operations performed in the data store when processing broadcast packets in an OpenFlow (OF) `packet-in` request (Fig. 4.3a). Table 4.1 shows that for the purpose of associating the source address of the packet to the ingress switch-port where it was received, the Learning Switch application performs one write (W) operation with a request size (Request) of 113 bytes and reply size (Reply) of 1 byte (reporting success).

Operation	Type	Request	Reply
1) Associate source address to ingress port	W	113	1

Table 4.1: Workload lsw-0-broadcast operations and sizes (in bytes).

Notice that in order to tag the *source-to-port* entry as the most recently used (in the LRU table) the Load Balancer has to perform this write regardless of the entry being already known or not.

4.2.2 Unicast Packet

This workload adds an operation to the previous one, since for every unicast packet we must also fetch the known switch port location of the destination address. Table 4.2 shows that this second operation requires 36 bytes for the request payload (sent to the data store) and a 77 byte response size containing the known switch port.

Operation	Type	Request	Reply
1) Associate source address to ingress port	W	113	1
2) Read egress port for destination address	R	36	77

Table 4.2: Workload lsw-0-unicast operations and sizes (in bytes).

4.2.3 Optimizations

The Learning Switch operations are simple, so there is not much to be done to improve them. Still, there is an overhead in the messages exchanged considering their content: a MAC address (6 byte standard); and a switch port identifier. This is justified by the fixed overhead of the Java Object Serialization Stream Protocol [56] used to transform the object values into byte arrays (as required by the data store prototype). By doing it "manually" (i.e., convert the MAC address and switch port identifier to their underlying byte representation) we lower the total size of the messages in the unicast workload by 72% (see Table 4.3). The same goes for the broadcast workload (first line of the same table).

Operation	Type	Request	Reply
1) Associate source address to ingress port	W	29	1
2) Read egress port for destination address	R	27	6

Table 4.3: Workload lsw-1-unicast operations and sizes (in bytes).

Finally, we merge the two operations that compose the unicast workload (lsw-1-unicast) into one, as shown in Table 4.4. This is possible using micro components (recall

section 3.2.4), which enables the Learning Switch application to perform more than one operation with a single request to the data store. Notice that in the broadcast case it is irrelevant to use micro components since it is already composed of a single operation.

Operation		Type	Request	Reply
1)	Associate source address to ingress port; and read egress port for destination address	W	56	6

Table 4.4: Workload lsw-2-unicast operations and sizes (in bytes). The two operations are performed as a single one by resorting to micro components (recall section 3.2.4).

4.2.4 Evaluation

Fig. 4.4 shows the results of the performance analysis made to the data store using the five workloads described above. The reported metric for the average throughput is *flows per second*, with each flow being the equivalent to the execution of all the workload steps. Similarly, the measured latency is taken per flow. The resulting values follow an exponential growth as we increase the load on the system by adding more clients.

Surprisingly, the difference in performance between the original versions (with workload names prefixed by lsw-0) and the optimized size versions (prefixed by lsw-1) is unnoticeable (Fig. 4.4a). Indeed, we will soon verify that size optimizations are bearably unnoticeable in all our examples, except when they differ at least one order of magnitude.

Furthermore, Fig. 4.4a shows a significant difference between unicast and broadcast workloads. This is due to the different number of message exchanges. For the broadcast workload (1 message) the data store can support up to 20k Flows/s under 3 ms latency, whereas for the unicast workload (2 messages) the data store only supports 12k Flows/s, with the same 3 ms latency penalty. The results shown in Fig. 4.4b show an analagous result for the reduction of messages in the unicast workload (lsw-2-unicast).

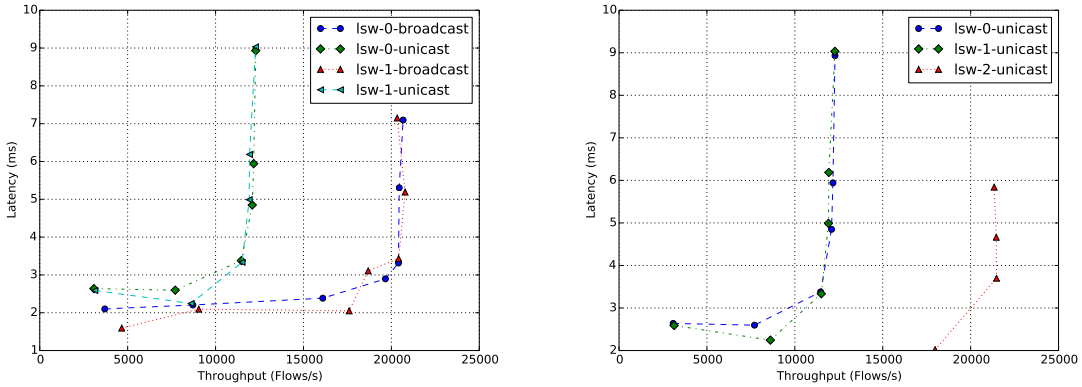
4.3 Load Balancer

The Load Balancer application employs a round-robin[8] algorithm to distribute the requests addressed to a Virtual IP (VIP) address across a set of servers.

Fig. 4.5 shows the entities relevant to understand this application. The VIP entity represents a virtual endpoint with a specified MAC, IP, protocol (ICMP, TCP or UDP), and port. Each VIP can be associated with one or more *Pools*[9]. Each *Pool* has a current assigned server (`current-member` attribute), which changes every time the round-robin algorithm is executed. Finally, the third entity — *Member* — represents a real server.

[8]This algorithm distributes each request to a different server in a circular fashion.

[9]The original application associates each vip to more than one pool since it considers enhancing the round-robin model in the future. As with all other applications, we prefer to maintain its original behavior.



(a) lsw-1-X vs. lsw-0-X

(b) lsw-X-unicast

Figure 4.4: Learning Switch workloads performance comparison.

Table 4.5 shows the different tables required by the Load Balancer application. The first three track entities by their key attributes. An additional table (`vip-ip-to-id`) links IPs to VIPs.

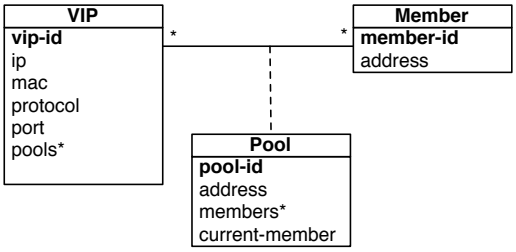


Figure 4.5: Simplified Load Balancer entity model. Only the attributes relevant to our discussion are shown.

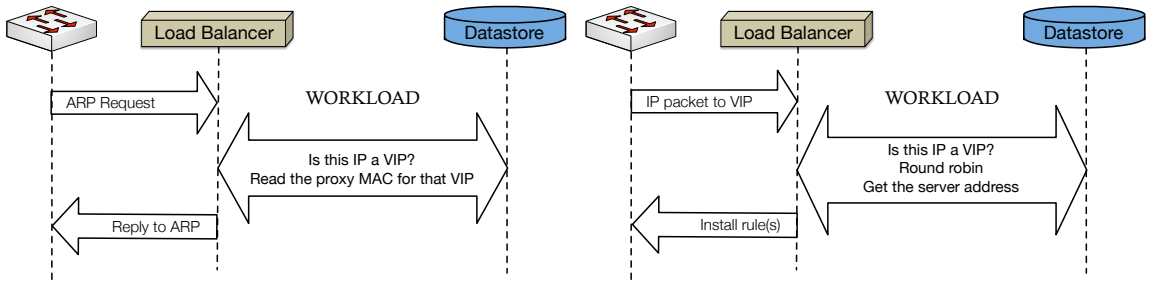
Name	Key	Value
vips	vip-id	vip
pools	pool-id	pool
members	member-id	member
vip-ip-to-id	ip	vip-id

Table 4.5: Load Balancer key-value tables.

The Load Balancer application asserts if any `packet-in` request triggered by a switch is addressed to a VIP. If so, two different executions flows are possible:

1. (Fig. 4.6a) when the event is caused by an Address Resolution Protocol (ARP) request, the Load Balancer application must fetch the `mac` address attribute of the VIP to reply to the source host.

2. (Fig. 4.6b) if the event is caused by Internet Protocol (IP) data packets the application must: (i) fetch the VIP information; (ii) choose and fetch a *Pool*; (iii) rotate the `current-member` attribute of the *Pool* (to perform the round-robin algorithm); and (iv) fetch the chosen *Member* data (to install the appropriate rule in the switch).



(a) ARP packet address at a VIP. (b) IP packets addressed at a VIP.

Figure 4.6: Load Balancer workloads by events.

4.3.1 ARP Request

Table 4.6 shows the operations that result from an OF `packet-in` caused by an ARP request querying the VIP MAC address. In the first operation, the Load Balancer application attempts to retrieve the `vip-id` for the destination IP. If it succeeds (since the association exists and the reply is different than 0), then the retrieved `vip-id` is used to obtain the related VIP entity in operation #2 (we surround the operation description with brackets to mark it as optional being that it is only executed when the first succeeds). Despite the fact that only the MAC address is required to answer the ARP request; the VIP entity is read entirely. Notice that the size (509 bytes) is two orders of magnitude bigger than a standard MAC address size (6 bytes).

Operation	Type	Request	Reply
1) Get `vip-id` for the destination IP	R	104	8
2) [Get VIP]	R	29	509

Table 4.6: Workload lbw-0-arp-request operations and sizes (in bytes). Bracketed operations is optional.

4.3.2 Packets to a VIP

Table 4.7 shows the detailed operations triggered by IP packets addressed at a VIP. The first two operations fetch the VIP entity associated with the destination IP address of the packet. From the VIP we obtain the `pool-id` used to retrieve the *Pool* (operation #3). The next step is to perform the round-robin algorithm by updating the `current-member` attribute of the retrieved *Pool*. This is done locally. Afterwards, the fourth operation aims to replace the data store *Pool* by the newly update one. If the *Pool* has changed between the retrieve and replace operation this operation fails (reply equal to 0) and we must try again by fetching the *Pool* one more time (repeating operation #3 and #4, hence the $\star$ mark). In order to check if the versions have changed, the replace operation contains both the original and updated *Pool* to be used by the data store. To

succeed the original client version must be equal to the current data store version when processing the request. If successful (reply equal to 1), we can move on and read the chosen *Member* (server) associated with the `member-id` that has been determined by the round-robin algorithm.

Operation	Type	Request	Reply
1) Get `vip-id` for the destination IP	R	104	8
2) [Get VIP]	R	29	509
3) [Get the chosen pool]*	R	30	369
4) [Conditional replace pool after round-robin]*	W	772	1
5) [Read the chosen Member]	R	32	221

Table 4.7: Workload lbw-0-ip-to-vip operations and sizes (in bytes). The * symbol signals that the operation may be repeated due to concurrent updates.

4.3.3 Optimizations

Table 4.8 shows all the optimizations done to the workload triggered by a packet addressed to a VIP. It is similar to previous workload description tables but this time, we show how the data store optimizations affect the workloads. The reader can attest the improvement in the workloads reading from left to right. To simplify our discussion we prefix each workload with a different name: lbw-0, lbw-1,..., lbw-4. For reference, Table 4.9 relates prefixes to data store optimization techniques. Prefix lbw-0 refers to the initial key value store implementation already presented (in Table 4.7).

Operation	Type	(Request, Reply)				
		lbw-0	lbw-1	lbw-2	lbw-3	lbw-4
1) Get VIP id of destination IP	R	(104,8)	(104,509)	(104,513)	(62,324)	-
2) Get VIP info (pool)	R	(29,509)				
3) Get the chosen pool	R	(30,369)	-	(30,373)	-	
4) Conditionally replace pool	W	(772,1)	-	(403, 1)	-	(11,4)
5) Read the chosen Member	R	(32,221)	-	(32,225)	(44,4)	

Table 4.8: Load Balancer lbw-*X*-ip-to-vip workload operations and respective sizes (in bytes) across different optimizations. Sizes marked with − are equal to the previous.

In the first improvement (lbw-1) we eliminate the double step required to obtain a VIP (first two operations). This can be done with the Cross Reference functionality by stating, when creating the `vip-ip-to-id` table, consulted in the first workload operation that its values can be used as keys in the `vips` table. Then, the Load Balancer application can fetch the VIP for the provided IP address in a single operation.

Prefix	Data store	Section
lbw-0	Simple Key-Value	3.2
lbw-1	Cross References	3.2.1
lbw-2	Versioning	3.2.2
lbw-3	Column Store	3.2.3
lbw-4	Micro Components	3.2.4

Table 4.9: Name guide to Load Balancer workloads.

Next, in workload lbw-2 we reduce operation #4 size in half, by "upgrading" the conditional replace (after round-robin) by a similar operation based on version numbers that are provided by the data store while reading the VIP information (notice the 4 byte increase in operation #1-2 caused by adding the version number of the VIP to the reply).

Following (lbw-3), we modify the `members` and `vips` tables to keep values as columns. As such, we can now replace the existing read of a VIP and Member by "partial" reads. Only a slight improvement (from 513 to 324 byte) is seen for reading the VIP (operation #1-2) since the application requires most of the VIP attributes. On the other hand, with a column-based `members` table, we reduce the return value of the last operation by a factor of 56 in the return value because we only require reading its IP attribute.

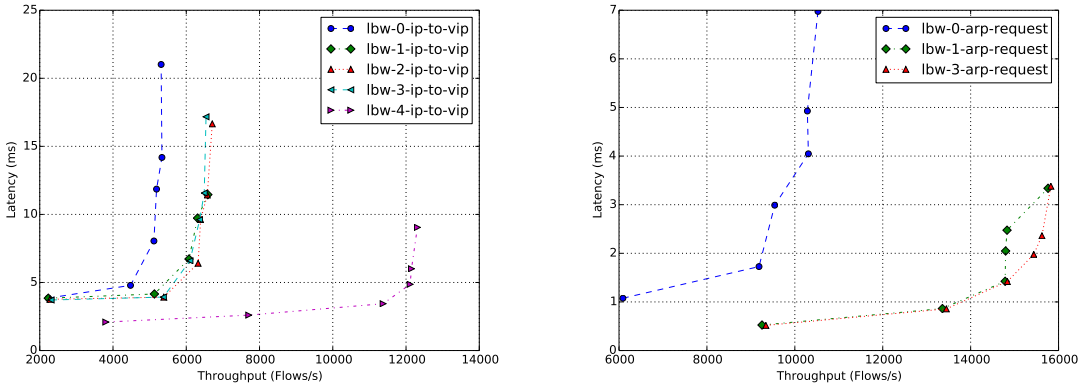
Finally (lbw-4), the most significant improvement consists in setting up a method in the data store equivalent to the local round-robin operation that also returns the Member IP in a single step (i.e., merges operations #3 through #5). Note also that there is an additional benefit that is not shown in Table 4.8. In the previous versions, we fetch and update a *Pool* in two separate steps, and as we explained, the conditional replace can fail in case of concurrent updates. This can become a potential bottleneck under peaks of traffic. This problem disappears in the final optimization phase (lbw-4), since load balancing is performed as a single operation in the data store (exploiting the linearizability property referred to in section 2.5).

4.3.4 Evaluation

Fig. 4.7a shows the performance results of our analysis of the different workloads optimizations. In the figure we can identify similar patterns to the previous analysis.

Again, the message size reduction optimizations have little to no effect, as can be attested from workloads lbw1 to lbw-3. This was expected since the relative improvements have smaller impact when compared with the Learning Switch case.

As before, it is the reduction of the number of messages that has the greatest impact. From workload lbw-0 to lbw-1 we see a clear, albeit small improvement. The improvement is much more significant with the final optimization (lbw-4) where we obtained a throughput of 12k Flows/s under 5 ms latency — at the same latency penalty (5 ms), we triple the throughput (4k to 12k) from lbw-0 to lbw-4.



(a) ip-to-vip. (b) arp-request

Figure 4.7: Load Balancer workloads performance comparison.

Regarding Fig. 4.7b, it shows the performance analysis to the arp request workload described previously. This workload is equivalent to the first two operations of the ip-to-vip workload in Table 4.8. Again, the message size improvements from lbw-1 to lbw-3 have no considerable effect as opossed to the reduction in the number of messages from lbw-0 to lbw-1.

4.4 Device Manager

The Device Manager application tracks and stores host device information such as the switch-ports attachment points (ports where devices are connected to). This information is retrieved from all OF packets that the controller receives. For each new flow, the Device Manager application retrieves the known switch ports for the source and destination addresses for later use by other applications.

The Device Manager application requires three data store tables, listed in Table 4.10. The first table (`devices`) keeps track of known devices. The second (`macs`), tracks the devices by their MAC address and Virtual Local Area Network (VLAN) identifier pair. Finally, a third table named `ips` links IPs addresses to one or more devices.

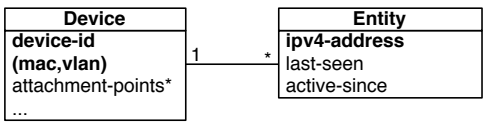
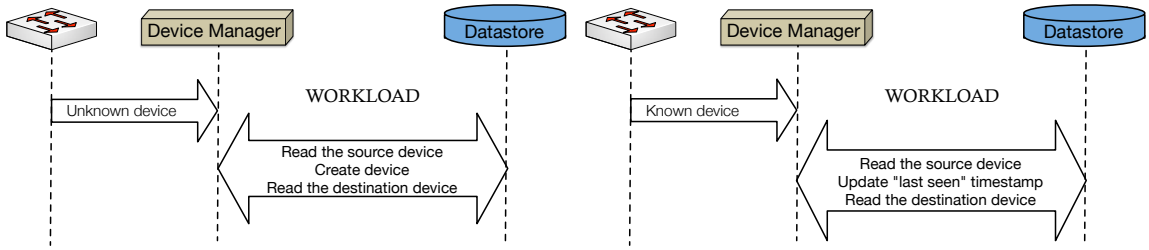


Figure 4.8: Simplified Device Manager class diagram. Only the attributes relevant to our discussion are shown.

Name	Key	Value
devices	device-id	device
macs	(MAC,VLAN)	device-id
ips	IP	device-ids*

Table 4.10: Device Manager key-value tables.

This application extracts the switch port, MAC and VLAN information for the source address from every OF `packet-in` request processed by the controller. Then it updates



(a) Packet from an unknown device. (b) Packet from a known device.

Figure 4.9: Workloads for this application heavily depend on the state of the data store. Unknown devices trigger several operations to their creation, while known devices only require an update of their ”last seen” timestamp. No matter the case, the source and destination devices are retrieved if they exist.

or creates new devices based on that information. A device (see Fig. 4.8) is uniquely identified by its `device-id` or (MAC,VLAN) pair (represented in bold in the figure). It is also composed of one or more entities (*Entity* class in the figure). Each entity is a “visible trace” of a device activity. The application always creates two entities for each device: one associated with the device IP address, that is created (or updated) on every ARP request, and a generic one (with IP 0.0.0.0), that is created (or updated) on every IP data packet seen from that device. The *last-seen* timestamp of the each entity is updated on every packet seen, and is later used to age out inactive devices.

We analyze and improve two workloads generated by this application. The first (Fig. 4.9a) is caused by an ARP packet from an unknown device, and the second (Fig. 4.9b) by an IP packet from a well-known device. In the former case, the application must create the device information and update all tables. As for the latter case, the Device Manager updates the source device `last-seen` timestamp. In both cases, the known attachment points of both source and destination devices are fetched to be made available to other applications.

4.4.1 Unknown Device

This workload is triggered in the specific case in which the source device is unknown and the OF message carries an ARP reply packet. As seen in Table 4.11, eight data store operations are required in order to create a device. The first operation reads the source device key. Being that it is unknown (notice, in the table, that the reply has a size of zero bytes corresponding to `null`) the application proceeds with the creation of the device. For this, the following write (second operation) atomically retrieves and increments a device unique `id` counter. Afterwards, the third and fourth operation updates the `devices` and `macs` tables respectively. Then, since the `ips` table links an IP to several devices, we need to first collect a set of devices (operation #5) in order to update it (operation #6). This *read-modify-write* operation can fail in case of concurrent updates.

However, this should not be an issue since it will be unusual to have a device updated by different controllers concurrently. If successful, the Device Manager has created the new device info and can, finally, move to the last two operations that fetch the destination device information. If unsuccessful, the process is repeated from step #5.

Operation	Type	Request	Reply
1) Read the source device key	R	408	0
2) [Get and increment the device id counter]	W	21	4
3) [Put new device in device table]	W	1395	1
4) [Put new device in `(MAC,VLAN)` table]	W	416	0
5) [Get devices with source IP]*	R	386	0
6) [Update devices with source IP]*	W	517	0
7) Read the destination device key	R	408	8
8) [Read the destination device]	R	26	1378

Table 4.11: Workload dm-0-unknown operations and sizes (in bytes).

4.4.2 Known Devices

When the devices are known to the application, a `packet-in` request triggers the operations seen in table 4.12. The first two operations read the source device information. Then an update is required to update the "last seen" timestamp of the device generic `entity`. Notice that the size of this request message is nearly twice that of a device (1444 bytes). This is due to the fact that this is a standard replace containing both the original device (fetch in step #2) and the updated device. This operation will fail if other data store client has changed the device. If so, the process is restarted from the beginning. Otherwise, the last two operations can fetch the destination device.

Operation	Type	Request	Reply
1) Read the source device key*	R	408	8
2) [Read the source device]*	R	26	1444
3) [Update "last seen" timestamp]*	W	2942	0
4) Read the destination device key	R	408	8
5) [Read the destination device]	R	26	1369

Table 4.12: Workload dm-0-known (Known Devices) operations and sizes (in bytes).

4.4.3 Optimizations

Table 4.13 summarizes the optimizations done to the *know devices* workload. As before, we use different optimization prefixes (dmw-0, dmw-1, ..., dmw-4) that are described in Table 4.14.

In the first optimization (dmw-1) the two-step operation required to fetch a device is replaced by a single read with the help of Cross References tables (just as we have done

Operation	Type	(Request, Reply)				
		dmw-0	dmw-1	dmw-2	dmw-3	dmw-4
1) Get source key	R	(408, 8)	(408,1274)	(408,1278)	(486,1261)	(28,1414) [a]
2) Get source device	R	(26,1444)				
3) Update timestamp	W	(2942,0)	(2602,0)	(1316,1)	(667,1)	(36,0)
4) Get target key	R	(408,8)	(408,1199)	(408,1203)	(416,474)	N/A
5) Get target device	R	(26,1369)				

[a)] This operation also fetches the target device.

Table 4.13: Device Manager dmw-X-known operations and respective sizes (in bytes) across different optimizations.

Prefix	Data store	Section
dmw-0	Simple Key-Value	3.2
dmw-1	Cross References	3.2.1
dmw-2	Versioning	3.2.2
dmw-3	Column Store	3.2.3
dmw-4	Micro Components	3.2.4

Table 4.14: Name guide to Device Manager workloads.

in lbw-1 in Table 4.8). Then, in dmw-2, we upgrade the replace operation used to update the Device timestamp (operation #3) by a versioned based replace operation.

Following these optimizations, in dmw-3 we use a column table to store devices. The update timestamp operation (#3) is reduced to half its previous size. This improvement is caused by using only the updated column in the replace operation. We could improve this further if we kept the timestamp in a single column on the data store. However, the timestamp is kept in an array of `entities`, and our current data store prototype cannot break each array element in a different column. Additionally, with columns we reduce the return size of the last operation (#4-5) by a factor of 3 since we only need to read the switch attachment points of the destination (target) device.

As before, the final optimization step is to use micro-components (dmw-4). Two components are created for the known device workload. First, one that merges the two operations required to fetch the source and destinations device into a single operation (notice the Not Applicable in operation #4-5). Second, one to update the timestamp in the data store. In this operation, only the device key, `entity` index, the new timestamp, and version number of the device are sent. This significantly improves the message size.

However, the most significant improvement of micro components is in creating devices. Table 4.15 shows that the different optimizations to the Device Manager applications make no significant improvements up to dmw-3. However, the introduction of a micro component to create a device replaces 5 operations (from #2 to #6) with a single

one. In addition, the source and destination device are read simultaneously, just as in the known-device workload. Thus, in the final optimization the workload is composed of two operations (#1 and #2-6).

Operation	Type	(Request, Reply)				
		dmw-0	dmw-1	dmw-2	dmw-3	dmw-4
1) Read source key	R	(408,0)	-	-	(486,0)	(28,201)[a]
2) Increment counter	W	(21,4)	-	-	-	
3) Update device table	W	(1395,1)	(1225,1)[b]	-	(1183,1)	
4) Update MAC table	W	(416,0)	-	-	-	(476,8)
5) Get from IP index	R	(386,0)	-	-	-	
6) Update IP index	W	(517,0)	-	-	-	
7) Get target key	R	(408,8)	(408,1208)[b]	(408,1212)	(416,474)	N/A
8) Get target device	R	(26,1378)				

[a] This operation also fetches the destination device.
[b] Differences in sizes caused by a marshalling improvement.

Table 4.15: Workload dm-0-unknown operations and sizes (in bytes).

4.4.4 Evaluation

In Fig. 4.10 we present the results from the performance analysis for the two workloads previously covered.

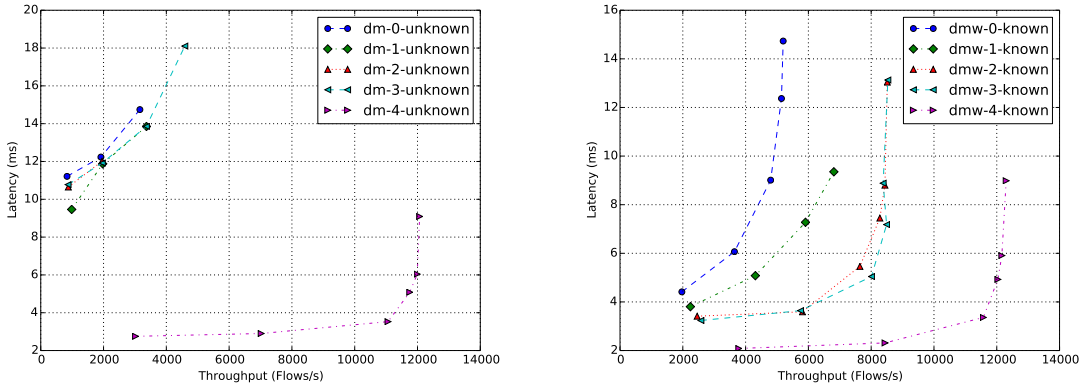
The figure confirms that the most significant improvement comes from using a micro component to create a device (dmw-4-unknown in Fig. 4.10a). With it we can reduce the latency penalty significantly while increasing the data store throughput from 2k Flows/s (and a penalty of around 12 ms) to nearly 12k Flows/s (with a reduced penalty of around 4ms). This significant improvement is also observed, albeit to a less degree, in the known device workload (Fig. 4.10b).

Also, note that under adequate workloads the data store has a significantly low latency penalty (the use of micro-components in both cases shows a steady latency penalty between 2 to 3 ms until 8k Flows/s).

4.5 Cache

In the workloads shown in the previous sections, the applications perform all operations in the data store. However, it is possible to perform some of the operations of each workload locally (in the controller) by integrating the applications with our cache interface (see section 3.2.5).

We should start by explaining why we “isolated” this optimization technique from the others. This is simply due to the differences in the evaluation process used in the



(a) Unknown device

(b) Known device

Figure 4.10: Device Manager performance comparison

cache optimization. This process differs from the others since it is theoretical and not experimental, as the reader will see.

In this section we show how we modified the workloads with the cache integration, the effect that it can have on the staleness of the data used by the clients (i.e., the applications), and if any consistency problems can arise. We conclude with a theoretical analysis of the performance of the cache optimization for the considered workloads.

4.5.1 Learning Switch

Given that Learning Switch is a single reader, single writer application[10], we can introduce caching mechanisms without impairing the consistency semantics or the staleness of the data. To clarify, a cached entry in the Learning Switch application is *always* consistent with the data store since no other controller modifies that entry. Therefore, with cache we can potentially avoid the data store while processing data plane events, thus avoiding the two operations in the unicast packet workload (lsw-1-unicast).

Table 4.16 shows the operations that can be cached (in gray background) from the lsw-1-unicast workload (in Table 4.3)[11].

	Operation	Type	Request	Reply
1)	Associate source address to ingress port	W	29	1
2)	Read egress port for destination address	R	27	6

Table 4.16: Workload lsw-3-unicast operations and sizes (in bytes). Operations in gray background are cached. Operation #1 can only be performed in cache, iff the source address to ingress port association is already known (in cache).

[10]For each switch table only a single thread, in a particular controller (the one responsible for the switch) reads and performs writes in the data store.

[11]We choose lsw-1-unicast as opossed to lsw-2-unicast, since our current implementation of the cache is based on the former.

First, we avoid re-writing the source address to source port association when we already now it, because it is present in cache (operation #1). Second, we can also avoid the read operation #2, which queries the egress port of the currently processed packet, if that entry is available in cache. With this improvement, we no longer have to read values from the data store as long as they are available in cache, and we still get consistent values because when we update a value we also update the cache. Note however, that the cache is also limited in size, thus entries are refreshed over time. In the case of cache-misses (i.e., entry is not available in cache), then the operation is performed in the data store.

By performing operation #1 in cache we affect the behavior of the original application. In the original Learning Switch application this re-write is not expensive (since it is local) and tags the source device entry as the most recent one in the LRU table. However, if we avoid updating the data store, then the active host will be removed from the LRU table sooner, even if active. Still, we choose to avoid the data store since an active host will (arguably) benefit in latency while being processed by the Learning Switch application until being removed from the cache and data store.

The reader may think of an exception where we find stale data: when a host moves from a switch to another, the tables from the first switch will have incorrect data and devices will be unreachable from that switch, for some time. However, this also happens with the centralized version of the application. In fact, this is why rules installed in the switches must have an idle timeout set (recall section 4.5.1). When one of the timeouts expires, the switch triggers a `flow-removed` message to the controller that, in turn, deletes the respective information in the data store. This type of problems resides on the data plane consistency side.

4.5.2 Load Balancer

In the Load Balancer case we use the cache to maintain VIP entities locally. The reader may think that by doing this we will affect the consistency of the application, since in the worst case the local VIP information can be stale. Namely, the application can install rules in the data plane to reach a VIP address server when the VIP has been removed (from the data store) or possibly changed the pool of servers. Note that this may also happen in the strong consistent version since a VIP fetched from the data store can be invalid by the time it arrives at the controller. With cache the probability of such an event is higher but we recall that this is tunable, since our implementation has a time-base cache validity interface (recall section 3.2.5).

Table 4.17 builds on the lbw-4-ip-to-vip workload (see Table 4.8) to illustrate how caching can increase the system performance. Only the first operation can be cached since it is the only read. For the write, we must rely on the data store to accurately perform the round-robin algorithm and return the address of the next server chosen. Otherwise, consistency problems could occur (i.e., conflicts). We leverage on this mandatory trip

to the data store, to evaluate the staleness of the VIP present in the cache. If the VIP changed between the time it was added to the application cache and the time the write is performed, then the data store aborts the operation and the application can restart from scratch. This time the value is obtained from the data store.

	Operation	Type	Request	Reply
1)	Get VIP pool for the destination IP*	R	62	324
2)	[Round-robin pool and read chosen Member]*	W	21	4

Table 4.17: Load Balancer lbw-5-ip-to-vip workload operations and respective sizes (in bytes). Greyed operations are cached.

4.5.3 Device Manager

As before (in the Load Balancer case), we can introduce the cache to service a portion of the reads operations locally. This time we keep devices[12] in cache which, in the worst case, can lead to path decisions based on outdated device locations. Again, this can also happen in the non-cached implementation of the application since devices read from the data store can already be outdated by the time the reply arrives at the controller (e.g., a mobile device moves from one access point to another).

First, we analyze how cache can affect the workload for a known device. Table 4.18 builds on our last workload (dmw-4-known in Table 4.13) to show that only a single operation (out of two) can be local. This operation (#1) reads the source and target devices based on the IP's addresses present in the packet. If any of the two are not available in cache, the application fetches both from the data store. Also, notice that we rely on the second operation (the write updating the timestamp) to validate the cached data, but in our current implementation, this operation only validates the source device. If the cached source device has been modified in the data store, the operation fails and the process must be repeated. If repeated then the first operation forcibly fetches values from the data store. Off course, this validity check could be expanded to include the destination device, thus narrowing the inconsistency window of all the cached information used to install flows.

In the unknown device workload (see dmw-4-unknown in Table 4.15) caching has no effect, because the cache will definitely need to consult the data store when the source device does not exists (thus cannot be in cache). In practice, the first operation of this workload is the same of the known device workload from Table 4.18, which must be performed in the data store when any of the devices is not available locally.

[12]For the thorough reader: we keep the cross reference tables that obtain devices.

	Operation	Type	Request	Reply
1)	Get source and target devices*	R	28	1414
2)	[Update "last-seen" timestamp of source device]*	W	36	0

Table 4.18: Device Manager dmw-5-known workload operations and respective sizes (in bytes). Greyed operations are cached.

4.5.4 Theoretical Evaluation

Table 4.19 shows the results the performance analysis to the cached workloads shown in the previous sections. The best case of each workload refers to when all the cache-enabled operations are performed locally. In contrast, the worst case refers to when all operations that compose the workload are performed in the data store. Of course, these values can only be used as a broad reference to understand the impact of cache. The true results may be far from the best case, since the frequency of cache-hits is dependent of the accepted staleness, the frequency of data plane events, the size of the cache, etc.

Regarding the results, in the lsw-3-unicast workload we show that the best case has an infinitive throughput and zero latency since no operation is performed in the data store. Thus, only the controller throughput and latency would have impact in such best case.

Naturally, the results of the device manager and load balancer best case are very similar since they have identical workloads. Additionally, the best case of each workload has roughly the double of the throughput when compared to the worst case. This was expected since the best case in each workload reduces the number of message sent to the data store in half.

Workload	Case	Throughput (kFlows/s)	Latency (ms)
lsw-3-unicast	best	∞	0
	worst[a]	21.5	4.7
lbw-5-ip-to-vip	best	21.4	4.8
	worst	11.4	3.4
dmw-5-known	best	21.4	3.6
	worst	11.1	3.5

Table 4.19: Bounded Analysis to Cache Workloads

[a]We consider the lsw-2-unicast as reference for the worst case

4.6 Discussion

The introduction of a fault-tolerant, consistent data store in the architecture of a distributed SDN controller has a cost. Adding fault tolerance increases the robustness of the system,

while strong consistency facilitates application design, but the fact is that these mechanisms affect system performance. First, the overall throughput will decrease to the least common denominator, which will in most settings be the data store. Second, the total latency will increase as the response time for a data path request now has to include i) the latency to send a request to the data store; ii) the time to process the request; and iii) the latency to reply back to the controller. Starting by assuming the inevitability of this cost, our objective was to show that, for some network applications at least, the cost may be bearable and the overall performance of the system remain acceptable.

First, we note that the performance results of our data store are similar to those reported for the original NOX and other popular SDN controllers [17]. The average throughput for the Learning Switch application (the only application considered in [17]) is not far from that reported by the original NOX (30kReq/s), so our data store would not become a bottleneck in this respect. In addition, the latency is equivalent to the reported for the different SDN controllers analyzed in that work (including the high-performance, multi-threaded ones), so the additional latency introduced, although non-negligible, can (arguably) be considered acceptable. We consider this result to be remarkable given that our data store provides both strong consistency and fault tolerance. Of course, the insightful reader will note that the results become quite distant from what is obtained with a controller that is optimized for performance, such as NOX-MT [17], in terms of throughput.

As the second part of the argument, it is important to understand that every update to our data store represents a similar execution of the protocol of Fig. 2.9, while in NOX-MT we have simply OF requests being received by a controller with the data store kept in main memory. Even if NOX-MT (or any other high-performance controller) synchronously writes particular data to disk (something that takes around 5ms), no more than 200 updates/second can be executed. This unequivocally shows that if some basic durability guarantees are required (e.g., to ensure recoverability after a crash), then the impressive capabilities of these high-performance controllers will be of little use.

Another goal of our work is to show that even with strong consistent semantics our data store can perform with equal or superior performance when compared to the existent distributed control platforms, namely HyperFlow [4] and Onix [3]. The HyperFlow evaluation results reveals that it can only maintain a bounded[13] window of inconsistency between controllers if the network state is updated at a rate lower than 1000 events per second [4]. Our results show that a strong consistent data store is superior by at least an order of magnitude (even when considering that an event causes more than one data store interaction). The limited performance in HyperFlow derives from the replication middleware used which, despite being eventually consistent, only supports 300 writes/s with 3kB messages [4].

[13]There is no information regarding the bound.

Onix, on the other hand, uses two data stores to maintain the network state: one eventually consistent, and the other strong consistent. The consistent data store, supports 50 SQL queries per second without batching. Even with batching (grouping more than one operation in a single request), the data store only supports 500 operations/s. As our results show, this performance limitation is associated with the implementation of the data store itself, not with the use of a state machine replication technique. The eventual consistent data store used in Onix can support 22 thousand "load attribute" updates/s with 5 replicas and 33 thousand with 3 replicas. These values are equivalent to ours and we support a strong, coherent view of the network state, contrary to Onix's eventual data store.

As a final note, we would like to give some context to these numbers *vis-à-vis* a real network deployment. According to Casado et al. [16], a Stanford network campus with 22 thousand hosts can have peaks of 9 thousand new flow requests per second (at maximum). Our results suggest that our distributed control plane would be able to handle that load for the application we considered. However, our architecture is not suitable for large-scale networks such as big data centers. Empirical studies have shown that data centers can have peaks of millions of new flows per second [19]. Clearly, our numbers show that our data store is incapable of handling such load. We leave as future work considering proactive SDN models to offer consistency guarantees in such environments.

Chapter 5 – Conclusions

The Free Lunch Is Over.

Herb Sutter

5.1 Conclusions

In this work we have proposed a distributed, highly-available, strongly consistent controller for SDNs. The central element of the architecture is a fault-tolerant data store that guarantees acceptable performance. We have studied the feasibility of this distributed controller by analyzing the workloads generated by representative SDN applications and demonstrating that the data store is capable of handling these workloads, not becoming a system bottleneck. Our results suggest that this architecture is capable of handling a medium-sized network.

We have chosen a simple design that emphasizes dependability and simplicity. One of the fundamental aspects of this design is that by using our data store, current centralized applications may easily become distributed and fault tolerant without affecting codebase significantly. However, our study also shows that a seamless integration process falls short in performance when compared to a fine-tuned adaptation of the data store to applications requirements. This is demonstrated by our ability to improve on the overall performance of the system by adopting well-known techniques borrowed from the database community.

The drawback of a strongly consistent, fault-tolerant approach for an SDN platform is the increase in latency, which limits responsiveness; and the decrease in throughput, which hinders scalability. Even assuming these negative consequences, an important conclusion of this study is that it is possible to achieve those goals while maintaining the performance penalty at an acceptable level (for a medium-size network). As the number of SDN production networks increase the need for dependability becomes essential. The key takeover of this work is that dependability mechanisms have their cost, and it is therefore an interesting challenge for the SDN community to integrate these mechanisms into scalable control platforms.

5.2 Future Work

As future work, we plan to expand our analysis to be thorough. Namely, we want to: $i)$ account for the impact that multiple applications can have in the control plane (running in parallel); and $ii)$ perform experiments with realistic network traffic simulations.

The impact of running multiple applications in parallel is crucial, since applications do not operate in isolation. Therefore, by including other applications, and optimizing the overall workload, we expected to have a practical Software Defined Network (SDN) controller in the future.

A realistic network traffic is also crucial to understand the limitations of our design. Despite our experience being focused in the more frequent type of data plane events (flow requests), we must take into consideration the hybrid workloads generated by the data plane in realistic scenarios.

Finally, as previously stated, we plan to study the feasibility of a proactive distributed SDN controller supported by a strong consistent data store to address the stringent requirements of large-scale networks such as big data centers.

Glossary

API Application Programming Interface.
ARP Address Resolution Protocol.
AS Autonomous System.

BGP Border Gateway Protocol.

CPU Central Processing Unit.
CRDT Convergent Replicated Data Type.

DHCP Dynamic Host Configuration Protocol.
DHT Distributed Hash Table.

FIFO First In,First Out.

HTTP HyperText Transport Protocol.

iBGP interior Border Gateway Protocol.
ICMP Internet Control Message Protocol.
IP Internet Protocol.

LRU Least Recently Used.

MAC Media Access Control.

NIB Network Information Base.
NOS Network Operating System.

OF OpenFlow.
OS Operating System.

RCP Routing Control Platform.
REST Representational State Transfer.
RPC Remote Procedure Call.

SDN Software Defined Network.
SMR State Machine Replication.
SQL Structured Query Language.

URL Uniform Resource Locator.

VIP Virtual Enpoint Internet Protocol.
VLAN Virtual Local Area Network.

References

[1] Albert Greenberg et al. “A Clean Slate 4D Approach to Network Control and Management”. In: *SIGCOMM Comput. Commun. Rev.* 35.5 (Oct. 2005), pp. 41–54 (cit. on pp. vii, 2, 13).

[2] Nate Foster et al. “Frenetic: A High-level Language for OpenFlow Networks”. In: *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*. PRESTO ’10. Philadelphia, Pennsylvania: ACM, 2010, 6:1–6:6 (cit. on pp. viii, 7, 35).

[3] Teemu Koponen et al. “Onix: A Distributed Control Platform for Large-scale Production Networks”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–6 (cit. on pp. viii, 5, 7, 27, 34, 72).

[4] Amin Tootoonchian and Yashar Ganjali. “HyperFlow: A Distributed Control Plane for OpenFlow”. In: *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*. INM/WREN’10. San Jose, CA: USENIX Association, 2010, pp. 3–3 (cit. on pp. viii, 5, 25, 72).

[5] Fred B. Schneider. “Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial”. In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319 (cit. on pp. viii, 29, 32).

[6] Alysson Bessani, João Sousa, and Eduardo Alchieri. *State Machine Replication for the Masses with BFT-SMaRt*. Tech. rep. 2013;07. DI-FCUL, Oct. 2013 (cit. on pp. ix, 8, 52, 54).

[7] Martin Casado. *The Scaling Implications of SDN [networkheresy.com]*. June 2011 (cit. on p. 1).
URL: `http://goo.gl/zILFm` (visited on 12/13/2013).

[8] Nick McKeown et al. “OpenFlow: Enabling Innovation in Campus Networks”. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74 (cit. on pp. 3, 4, 15).

[9] Nick Feamster and Hari Balakrishnan. “Detecting BGP Configuration Faults with Static Analysis”. In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 43–56 (cit. on p. 3).

[10] Declan McCullagh. *How Pakistan knocked YouTube offline [cnet.com]*. Feb. 2008 (cit. on p. 3).
URL: `http://goo.gl/bVWOSa` (visited on 12/13/2013).

[11] Open Network Foundation. *OpenFlow Switch Specification (version 1.2) [opennetworking.org]*. Dec. 2011 (cit. on pp. 4, 39).
URL: \url{http://goo.gl/tKo6r} (visited on 12/13/2013).

[12] Ahmed Khurshid et al. "VeriFlow: Verifying Network-wide Invariants in Real Time". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 15–28 (cit. on p. 4).

[13] Nikhil Handigol et al. "Where is the Debugger for My Software-defined Network?" In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: ACM, 2012, pp. 55–60 (cit. on p. 4).

[14] Sushant Jain et al. "B4: Experience with a Globally-Deployed Software Defined Wan". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 3–14 (cit. on pp. 4–6, 27).

[15] Soheil Hassas Yeganeh and Yashar Ganjali. "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: ACM, 2012, pp. 19–24 (cit. on pp. 5, 6, 24, 40).

[16] Martin Casado et al. "Ethane: Taking Control of the Enterprise". In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '07. Kyoto, Japan: ACM, 2007, pp. 1–12 (cit. on pp. 5, 13, 73).

[17] Amin Tootoonchian et al. "On Controller Performance in Software-defined Networks". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. Hot-ICE'12. San Jose, CA: USENIX Association, 2012, pp. 10–10 (cit. on pp. 5, 22, 72).

[18] Robert Colin Scott et al. *What, Where, and When: Software Fault Localization for SDN*. Tech. rep. UCB/EECS-2012-178. EECS Department, University of California, Berkeley, 2012 (cit. on p. 5).

[19] Theophilus Benson, Aditya Akella, and David A. Maltz. "Network Traffic Characteristics of Data Centers in the Wild". In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC '10. Melbourne, Australia: ACM, 2010, pp. 267–280 (cit. on pp. 5, 73).

[20] Brandon Heller, Rob Sherwood, and Nick McKeown. "The Controller Placement Problem". In: *SIGCOMM Comput. Commun. Rev.* 42.4 (Sept. 2012), pp. 473–478 (cit. on p. 6).

[21] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, pp. 11–11 (cit. on pp. 6, 28, 34).

[22] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220 (cit. on pp. 6, 28, 31).

[23] James C. Corbett et al. “Spanner: Google’s Globally-Distributed Database”. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI’12. Hollywood, CA, USA: USENIX Association, 2012, pp. 251–264 (cit. on pp. 6, 34).

[24] Hyojoon Kim et al. “CORONET: Fault Tolerance for Software Defined Networks”. In: *Proceedings of the 2012 20th IEEE International Conference on Network Protocols (ICNP)*. ICNP ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–2 (cit. on pp. 6, 39, 40).

[25] AlanD. Fekete and Krithi Ramamritham. “Consistency Models for Replicated Data”. In: *Replication*. Ed. by Bernadette Charron-Bost, Fernando Pedone, and André Schiper. Vol. 5959. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 1–17 (cit. on pp. 6, 30).

[26] Eric Brewer. *Towards Robust Distributed Systems [berkeley.edu]*. July 2000 (cit. on p. 7).
URL: `http://goo.gl/u7uhTS` (visited on 12/13/2013).

[27] Seth Gilbert and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services”. In: *SIGACT News* 33.2 (June 2002), pp. 51–59 (cit. on pp. 7, 30).

[28] Dan 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*. HotSDN ’12. Helsinki, Finland: ACM, 2012, pp. 1–6 (cit. on p. 7).

[29] Paulo Verissimo and Luis Rodrigues. *Distributed systems for systems architects*. Vol. 1. Springer, 2001 (cit. on p. 7).

[30] Peter Bailis and Ali Ghodsi. “Eventual Consistency Today: Limitations, Extensions, and Beyond”. In: *Queue* 11.3 (Mar. 2013), 20:20–20:32 (cit. on pp. 7, 30).

[31] Mark Reitblatt et al. “Abstractions for Network Update”. In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’12. Helsinki, Finland: ACM, 2012, pp. 323–334 (cit. on pp. 7, 35).

[32] Nick Feamster, Jennifer Rexford, and Ellen Zegura. *The Road to SDN: An Intellectual History of Programmable Networks [princeton.edu]*. 2013 (cit. on p. 11).
URL: `http://goo.gl/Xxky9s`.

[33] Nick Feamster et al. “The Case for Separating Routing from Routers”. In: *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*. FDNA ’04. Portland, Oregon, USA: ACM, 2004, pp. 5–12 (cit. on p. 12).

[34] Matthew Caesar et al. “Design and Implementation of a Routing Control Platform”. In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 15–28 (cit. on p. 12).

[35] Natasha Gude et al. “NOX: Towards an Operating System for Networks”. In: *SIGCOMM Comput. Commun. Rev.* 38.3 (July 2008), pp. 105–110 (cit. on pp. 16, 22, 25).

[36] Roy T. Fielding and Richard N. Taylor. “Principled Design of the Modern Web Architecture”. In: *Proceedings of the 22Nd International Conference on Software Engineering*. ICSE ’00. Limerick, Ireland: ACM, 2000, pp. 407–416 (cit. on p. 17).

[37] Zen Cai, Alan L Cox, and T S Eugene Ng. *Maestro: A System for Scalable OpenFlow Control*. Tech. rep. TR10-08. Rice University, 2012, pp. 1–10 (cit. on p. 22).

[38] David Erickson. “The Beacon Openflow Controller”. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN ’13. Hong Kong, China: ACM, 2013, pp. 13–18 (cit. on p. 23).

[39] Jeremy Stribling et al. “Flexible, Wide-area Storage for Distributed Systems with WheelFS”. In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI’09. Boston, Massachusetts: USENIX Association, 2009, pp. 43–58 (cit. on pp. 26, 29).

[40] *SDN Controller Ecosystems Critical to Market Success [wordpress.com]*. June 2012 (cit. on p. 27).
URL: `http://goo.gl/06dB4P` (visited on 12/13/2013).

[41] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects”. In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492 (cit. on pp. 29, 32, 42).

[42] Werner Vogels. “Eventually Consistent”. In: *Queue* 6.6 (Oct. 2008), pp. 14–19 (cit. on pp. 29, 30).

[43] Leslie Lamport. “The Part-Time Parliament”. In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169 (cit. on pp. 29, 42).

[44] Barbara Liskov. “From Viewstamped Replication to Byzantine Fault Tolerance”. In: *Replication*. Ed. by Bernadette Charron-Bost, Fernando Pedone, and André Schiper. Vol. 5959. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 121–149 (cit. on pp. 29, 32, 33, 41).

[45] Jun Rao, Eugene J. Shekita, and Sandeep Tata. “Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore”. In: *Proc. VLDB Endow.* 4.4 (Jan. 2011), pp. 243–254 (cit. on p. 34).

[46] William J. Bolosky et al. “Paxos Replicated State Machines As the Basis of a High-performance Data Store”. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI’11. Boston, MA: USENIX Association, 2011, pp. 11–11 (cit. on p. 34).

[47] Alysson Bessani et al. “On the Efficiency of Durable State Machine Replication”. In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC’13. San Jose, CA: USENIX Association, 2013, pp. 169–180 (cit. on p. 34).

[48] Marco Canini et al. “Software Transactional Networking: Concurrent and Consistent Policy Composition”. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN ’13. Hong Kong, China: ACM, 2013, pp. 1–6 (cit. on p. 35).

[49] Mark Reitblatt et al. “FatTire: Declarative Fault Tolerance for Software-defined Networks”. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN ’13. Hong Kong, China: ACM, 2013, pp. 109–114 (cit. on p. 40).

[50] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. “Mencius: Building Efficient Replicated State Machines for WANs”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, 2008, pp. 369–384 (cit. on p. 40).

[51] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. English. 2nd Edition. Springer, Feb. 2011 (cit. on p. 42).

[52] H. T. Kung and John T. Robinson. “On Optimistic Methods for Concurrency Control”. In: *ACM Trans. Database Syst.* 6.2 (June 1981), pp. 213–226 (cit. on p. 46).

[53] Joshua Bloch. *Effective java*. Addison-Wesley Professional, 2008 (cit. on p. 47).

[54] Nikhil Handigol et al. “Reproducible Network Experiments Using Container-based Emulation”. In: *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. CoNEXT ’12. Nice, France: ACM, 2012, pp. 253–264 (cit. on p. 53).

[55] Fábio Botelho. *Appendices* (cit. on p. 55).
URL: `http://goo.gl/q3Cvhz`.

[56] *Java Object Serialization Specification (version 6.0). Online at :`http://docs.oracle.com/javase/6/docs/platform/serialization/spec/serialTOC.html`* (cit. on p. 57).