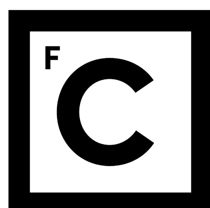
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade de Lisboa: Repositório.UL

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA



Ciências
ULisboa

CONSISTENT AND FAULT-TOLERANT SDN CONTROLLER

André Alexandre Lourenço Mantas

MESTRADO EM SEGURANÇA INFORMÁTICA

Dissertação orientada por:
Prof. Doutor Fernando Manuel Valente Ramos

2016

Agradecimentos

Gostaria de agradecer a várias pessoas que me acompanharam ao longo deste percurso e sem as quais nada disto seria possível.

Primeiramente, aos meus pais por me criarem e educarem naquilo que sou hoje, por me proporcionarem a possibilidade de frequentar a faculdade e de realizar este mestrado (mal eu sabia onde me estava a meter).

À minha namorada Sara por todo o apoio e incentivo que me deu ao longo dos bons e maus momentos desta jornada. Por estar sempre lá para mim e ajudar em tudo aquilo que conseguia, inclusive a decifrar a letra de médico do meu orientador.

Aos professores Paulo Veríssimo e Nuno Neves pelas aulas de Segurança na licenciatura que me despertaram o interesse pelo MSI. Já no mestrado, tenho que agradecer à professora Dulce por toda a ajuda que me deu e por ter disponibilizado uma das suas aulas durante o primeiro semestre do primeiro ano para que o professor Fernando pudesse apresentar a cadeira de Protocolos em Redes de Dados. Não sei se teria escolhido esta cadeira no segundo semestre sem esta apresentação e, sem isso, não teria escolhido este tema de tese.

Ao meu orientador, Fernando Ramos, começando pelas aulas de PRD, passando pelas SDN, e acabando na proposta do tema para a tese. Obrigado pelo apoio, pelas reuniões, ideias e discussões que levaram ao Rama e a este relatório final.

Ao projeto SUPERCLOUD e à FCT pelo financiamento da bolsa para a tese.

Ao Desmarques por tudo desde o primeiro ano de licenciatura até hoje. Pelo estudo para os exames, pelos projetos e discussões das cadeiras que correram melhor ou pior, pelas gargalhadas descontroladas em locais inoportunos e por tudo o resto.

A todos da nossa sala de mestrado no C8 pelos dias que passamos a partir a cabeça com as nossas teses, reuniões, almoços e lanches e muito mais.

Ao bar do C1 pelas torradas e pães com chouriço e queijo, ao refeitório Universo da Faculdade de Direito pelas ervilhas com ovos escalfados, pela feijoada e, claro, pela massa universo.

A todos os outros que não mencionei aqui mas que me acompanharam na faculdade e que de uma forma ou de outra fizeram este percurso mais divertido.

E finalmente à seleção de Portugal por ter ganho o Euro 2016, mostrando-me que tudo é possível.

À vida.

Resumo

O conceito de Software-Defined Networkings (SDNs) acaba com o acoplamento que existe nas redes tradicionais entre o plano de controlo e o plano de dados. Este desacoplamento facilita o desenvolvimento de aplicações inovadoras e flexíveis para gerir, monitorizar e programar as redes. Em SDN, as aplicações usam uma visão da rede logicamente centralizada, fornecida por controladores que programam remotamente os switches na rede. Se esta visão da rede não for coerente com o verdadeiro estado da rede, as aplicações vão executar sobre um estado desatualizado e produzir resultados incorretos. Isto pode degradar significativamente o desempenho da rede e gerar problemas como criação de *loops* ou falhas de segurança. Como em qualquer sistema em produção, falhas em componentes devem ser expectáveis. Desta forma, é importante que o plano de controlo seja capaz de manter uma visão coerente da rede mesmo na presença de falhas nos controladores ou nas ligações. Para isto, a visão da rede tem que estar replicada de forma coerente entre vários controladores para que a falha de um não comprometa a disponibilidade do sistema. Adicionalmente, o estado mantido pelos switches tem que ser tratado de forma coerente, o que é particularmente difícil na presença de falhas.

Este trabalho propõe um plano de controlo SDN resiliente que permita a aplicações de rede *inalteradas* correr num ambiente tolerante a falhas e coerente (tanto a nível do estado do controlador como dos switches). Para conseguir o ambiente tolerante a falhas, os controladores devem replicar (de forma transparente) entre si os eventos recebidos antes que estes sejam entregues às aplicações de rede. Para conseguir um sistema coerente, a ideia principal é fazer com que os controladores processem as mensagens de controlo (incluindo enviar comandos aos switches) transaccionalmente, exatamente uma vez, e pela mesma ordem total. Esta ordem total é decidida por um dos controladores, o *líder*, que também é o controlador responsável por replicar os eventos pelos outros controladores e por enviar os comandos para os switches. Desta forma todos os controladores chegarão ao mesmo estado interno, o que faz com que tenham a mesma visão da rede e possam gerir os switches no caso de o controlador *líder* falhar.

Atualmente existe apenas um controlador para SDN que utiliza estes mecanismos e que oferece as garantias de tolerância a falhas e coerência referidas, o Ravana. No entanto, o Ravana requer que sejam feitas modificações aos switches de rede e ao protocolo

usado entre os controladores e os switches (i.e., o protocolo OpenFlow) como parte da sua implementação. Especificamente, o Ravana requer que:

- Os switches mantenham dois *buffers* locais, um para eventos que enviam aos controladores e outro para comandos que recebem destes;
- O protocolo OpenFlow seja alterado para adicionar duas mensagens novas de confirmação (controlador confirma a receção de eventos e switch confirma a receção de comandos) e duas mensagens novas para a limpeza dos buffers mantidos pelos switches (enviadas pelos controladores).

Estes requisitos tornam a adoção do Ravana em sistemas reais problemática, pois não existem switches com estas características nem é expectável que o protocolo OpenFlow seja alterado no futuro próximo de acordo com as modificações requeridas pelo Ravana. Com estas limitações em mente, desenvolvemos o Rama, um controlador para SDN que oferece as mesmas garantias de tolerância a falhas e coerência que o Ravana mas que *não requer modificações aos switches nem ao protocolo OpenFlow*.

O protocolo do Rama utiliza técnicas inovadoras para cumprir estes objetivos. Enquanto no Ravana cada switch apenas envia os eventos para o líder atual (e guarda o evento num buffer para o caso de ser preciso reenviar para o novo master), no Rama os switches enviam os seus eventos para todos os controladores. Para isto, fazemos uso do protocolo OpenFlow para definir os tipos de mensagens que os switches devem enviar para cada controlador (por exemplo, definindo os controladores como estando no papel *equals*). Isto permite que os controladores se possam coordenar entre si para não processar o mesmo evento mais que uma vez e, mais importante ainda, para que nenhum evento seja perdido sem depender do reenvio do evento pelo switch (como todos os controladores recebem o evento, basta que um não falhe para o evento não se perder). De seguida, o controlador líder replica o evento para os outros controladores (obedecendo a uma ordem total) e entrega-o às suas aplicações de rede. Estas aplicações vão gerar comandos para serem enviados para os switches. Aqui entra outro aspeto diferenciador entre o Rama e o Ravana. Enquanto no Ravana o líder simplesmente envia o comando a contar que, em caso de falha, o switch seja capaz de filtrar comandos repetidos, o Rama usa um novo mecanismo introduzido no OpenFlows (OFs) 1.4 – os Bundles (ou grupos) – para garantir que os switches apenas processam cada comando uma única vez. Os bundles permitem aos controladores enviar vários comandos para um grupo mantido pelo switch e de seguida dizer-lhe para processar todos os comandos nesse grupo de forma atómica. Ao fazer isto, o switch, de acordo com o protocolo OFs, tem que enviar uma mensagem de volta ao controlador a confirmar que aquele grupo de comandos foi completamente processado. Esta confirmação diz ao controlador líder que mais nenhum controlador pode voltar a enviar os comandos para aquele evento. Porém, como o líder pode falhar sem avisar os outros controladores que recebeu esta confirmação, não é trivial atingir este

propósito. Para resolver este problema, propomos um mecanismo adicional (já que não é possível programar o switch para enviar esta confirmação para todos os controladores): adicionamos uma mensagem do tipo PacketOut ao grupo mantido pelo switch que tenha como destino todos os controladores. Desta forma, quando o switch processa o grupo, o líder recebe a confirmação normal, indicativa de o grupo estar processado, e os outros controladores recebem uma mensagem com o conteúdo definido pelo líder (por exemplo, o identificador do evento).

Estes dois mecanismos, quando usados em conjunto – enviar eventos para todos os controladores e enviar comandos usando bundles – permitem ao Rama oferecer as mesmas garantias de coerência que o Ravana mas sem alterar os switches nem o protocolo OFs.

O Rama foi desenvolvido tendo como base o Floodlight, um controlador SDN modular, open source, escrito em Java e com uma comunidade ativa. No Floodlight, aplicações de rede (e.g., balanceador de carga, firewall) podem ser integradas como módulos num núcleo bem definido que implementa abstrações de rede comuns (e.g., topologia da rede, descoberta e gestão de caminhos na rede) e oferece uma interface que torna fácil enviar e receber mensagens aos switches na rede. Para tratar as mensagens vindas dos switches de rede, o Floodlight usa um processamento em forma de encadeamento (pipeline), onde cada módulo processa a mensagem (i.e., atualizar o seu estado e possivelmente enviar comandos para programar o switch em conformidade com a situação) um após o outro. Cada módulo pode ainda definir relações de ordem com outros módulos (i.e., se este módulo deve receber mensagens antes ou depois de outros módulos) e, quando recebe a mensagem, decidir se o processamento da mesma pelos outros módulos deve continuar ou parar.

No desenvolvimento do Rama decidimos usar o ZooKeepers (ZKs) para fazer a coordenação entre os controladores pela sua confiabilidade, simplicidade e ampla utilização. O ZK da Apache pode ser visto como um serviço de coordenação centralizado que expõe um conjunto de primitivas (e.g., gestão de nomes, gestão de configurações, sincronização, serviços de grupo) a ser utilizado por aplicações distribuídas de forma a que estas não tenham que reimplementar as primitivas de cada vez que são necessárias. As características principais do ZK são: alto desempenho, alta disponibilidade, acesso estritamente ordenado e o uso de um modelo de dados hierárquico semelhante a um sistema de ficheiros. O Rama usa o ZooKeeper para:

- Gestão de membros: saber que controladores estão ativos, detetar falhas de controladores e realizar eleição de líder no caso de falha do líder atual. Para isto usamos o mecanismo de nós temporários que são criados por cada controlador e apagados pelo ZK quando a ligação deste com algum controlador é terminada (e.g., por timeout).

- Replicação de eventos: o líder armazena um ou mais eventos em nós numa *pasta* observada pelas outras réplicas de forma a que estas sigam a ordem definida pelo

líder e saibam em que estado ele se encontra. Adicionalmente, também é mantida a informação de que eventos já estão processados pelos switches (i.e., o líder já enviou os comandos para o evento).

Para a fase de avaliação, usamos uma versão modificada do Cbench – a ferramenta usada para medir o desempenho de controladores OpenFlow – de forma a suportar bundles. Com as nossas experiências demonstramos que, embora o Rama atinja níveis de desempenho aceitáveis (cerca de 28 mil respostas por segundo) ainda fica um pouco atrás do Ravana (40 mil respostas por segundo). Esta diferença é devida ao custo adicional que o Rama incorre de modo a não modificar os switches nem o protocolo OpenFlow. Nomeadamente, o uso de bundles leva ao envio de mensagens adicionais para os switches (envio de quatro mensagens contra uma do Ravana para responder a um evento), o que em conjunto com a replicação e ordenação dos eventos reduz um pouco o desempenho do sistema em geral.

Desta nossa proposta resultam várias contribuições, incluindo um poster aceite na conferência NSDI'16, contribuições para o deliverable 4.2 do projecto H2020 SUPERCLOUD, e ainda contribuições para projetos externos:

- Floodlight: contribuições para o controlador Floodlight relacionadas com o tratamento de bundles e outros melhoramentos.
- OpenvSwitch: contribuição para aceitar e tratar mensagens do tipo PacketOut dentro de bundles. Previamente estas mensagens eram rejeitadas.
- Cbench: modificação da ferramenta Cbench que avalia o desempenho dos controladores OpenFlow para aceitar e tratar mensagens relacionadas com bundles.

Palavras-chave: Redes definidas por software, Tolerância a falhas, Coerência forte, OpenFlow

Abstract

The concept of Software-Defined Networking (SDN) breaks the coupling in traditional networks between the control and data planes. This decoupling allows the development of innovative and flexible applications to manage, monitor, and program the network. In SDN, applications use a logically centralized network view, provided by the controllers, to remotely program switches in the network. If this network view is not consistent with the actual state of the network, applications will operate on a stale state and produce incorrect outputs. This can significantly degrade the network performance (e.g., packets can be forwarded through a failed link) and create problems such as network loops or security failures. As in any system in production, component failures must be the rule and not the exception. Thus, it is important that the control plane is able to maintain a consistent network view in the presence of failures in both controllers and links. Therefore, the network view must be consistently replicated across several controller replicas so that the failure of one controller does not compromise the entire control plane. Additionally, to ensure a correct system, the state maintained by switches must be handled in a consistent way, which is particularly difficult in the presence of failures.

This work proposes a resilient SDN control plane that allows modification-free applications to run in a fault-tolerant and consistent environment (in both controllers and switches state). To achieve the fault-tolerant environment, controllers must replicate (transparently) the received events among themselves before these are delivered to the network applications. For the consistent environment, the main idea is that controllers process control messages transactionally, exactly once, achieving a correct and consistent operation of both controllers and switches, even if some of them fail. The two main techniques used are instructing switches to send events to all controllers, which coordinate to ensure exactly once event processing, and using OpenFlow Bundles as the acknowledgement mechanism used by controllers to process commands on switches exactly once.

The differentiating factor and novelty over existing works is the fact that the consistency guarantees our proposal assures, in a fault-tolerant SDN environment, do not require modifications to neither switches nor to the OpenFlow protocol.

Keywords: Software-Defined Networking, Fault-Tolerance, Strong consistency, Control plane, OpenFlow.

Contents

List of figures xi

List of tables xiii

1 Introduction 1

1.1 Control plane in SDN . . . 2
1.2 Motivation . . . 3
1.3 Goals and Contributions . . . 5
1.4 Additional Outcomes . . . 6
1.5 Document structure . . . 6

2 Context & Related Work 7

2.1 Context . . . 7
2.2 Scaling SDN in the control plane . . . 11
2.2.1 Improving control plane performance . . . 11
2.2.2 Distributing the control plane . . . 11
2.3 Scaling SDN in the data plane . . . 13
2.4 Resilience in SDN . . . 14
2.4.1 Consistent network updates . . . 15
2.4.2 Fault-tolerance . . . 15

3 Rama Design 19

3.1 Architecture . . . 19
3.2 Why Consistency Matters . . . 20
3.2.1 Inconsistent event ordering . . . 21
3.2.2 Unreliable event delivery . . . 21
3.2.3 Repetition of commands . . . 22
3.3 Consistent and Fault-tolerant protocol . . . 22
3.3.1 Fault cases . . . 25
3.4 Consistency Properties . . . 28

4 Implementation 31
4.1 ZooKeeper 31
4.2 Floodlight architecture 33
4.3 Rama architecture 34
4.4 Event Replication and ZK Manager 35
4.4.1 Fault Detection and Leader Election 37
4.4.2 Event batching 37
4.5 Bundle Manager 38

5 Evaluation 39
5.1 Rama Performance 40
5.2 Latency 43
5.3 Failover Time 43

6 Conclusion & Future Work 45

Glossary 47

References 53

List of Figures

1.1 Control and data plane coupling in traditional network vs. SDN 2
1.2 SDN flow execution 2

2.1 SDN architecture in layers 8
2.2 Main components of an OpenFlow enabled switch 9
2.3 OpenFlow table entry structure 10
2.4 Ravana Protocol 17

3.1 High level architecture of the system 20
3.2 Control loop 23
3.3 Fault-free case of the protocol 24
3.4 OpenFlow Bundles 25
3.5 Failure case 1 of the protocol 26
3.6 Failure case 2 of the protocol 27

4.1 ZooKeeper components 32
4.2 Floodlight modules architecture 33
4.3 Floodlight thread architecture 34
4.4 Rama thread architecture 34
4.5 ZooKeeper node structure 36

5.1 Experiment setup 39
5.2 Throughput comparison 40
5.3 Rama throughput with different number of switches 42
5.4 Variation of Rama throughput with batch size 42
5.5 Rama failover time 44

List of Tables

3.1 How Rama and Ravana achieve the same consistency properties using different mechanisms 29

4.1 Simplified ZooKeeper API 32

Chapter 1 – Introduction

Traditional networks have had an undeniable success over the years, even with all the complexities that their devices must face. These devices are responsible not only for forwarding packets across the network but also for path computation and running network protocols that implement core network functionalities like tunneling, discovery and access control. As a result, network devices must be equipped with powerful hardware, which will makes expensive and inflexible. On top of that, since these devices are implemented in proprietary software and hardware, modifying or improving them is difficult, which makes it very hard to experiment with new control protocols in existing networks.

The slow evolution in the traditional network control plane is caused by this fundamental problem which leads to a lack of both network management abstractions and of common control platform. This makes the design and deployment of control protocols to take many years, and prevents the development of innovative, flexible and reliable applications for controlling networks [1].

To solve this and other issues with traditional networks, the concept of Software-Defined Networking (SDN) was introduced with the goal of decoupling the control plane (which decides how to handle traffic) from the data plane (which forwards traffic according to the control plane decisions) via standardized interfaces. A well-defined programming interface between the switches and the SDN controller enables this decoupling and allows the control logic to evolve independently from the data plane. Additionally, this separation of concerns is very attractive as it opens the doors to new ways of managing, monitoring and programing the network. Figure 1.1 ilustrates the differences between traditional networks and SDN. In traditional networks (1.1a), each network element has its own (coupled) control plane with possibily different protocols and interfaces. In contrast, SDN (1.1b) moves the control plane to a new, logically centralized, network node - the controller (see section 1.1).

The most common and adopted interface in SDN is OpenFlow [2]. SDN and OpenFlow both started as an academic project but quickly gained commercial interest. Examples include B4 [3], a software-defined network developed by Google to interconnect its datacenters world-wide, and VMware's network virtualization platform, NSX [4, 5]. More information about SDN and OpenFlow is given in section 2.1. However, the new possibilities offered by SDN are not simple to put in practice and bring new problems not

present in traditional networks. Indeed, the task of providing a logically centralized network view brings many new challenges that must be addressed. Sezer et al. [6] explore challenges and possible solutions regarding network performance (network nodes must continue to be fast), scalability (controllers may need to manage a wide area network), security (new SDN-related security threats) and interoperability (SDN coexistence with traditional networks).

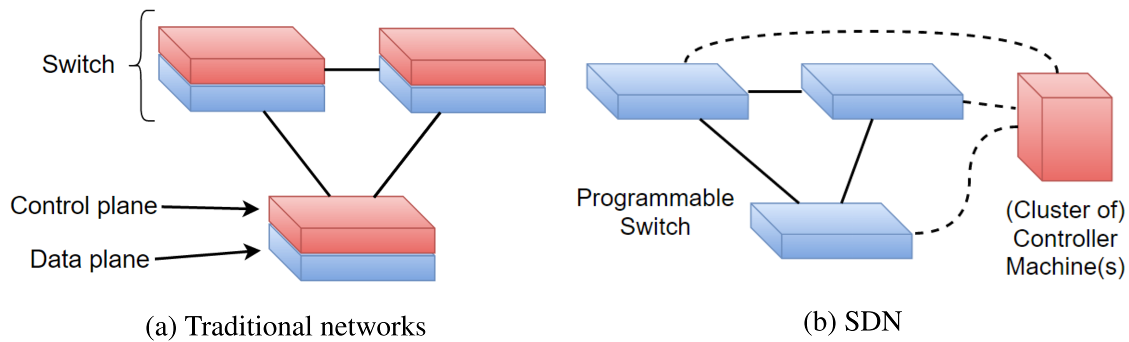


Fig. 1.1: Control and data plane coupling in traditional network vs. SDN

1.1 Control plane in SDN

The controllers are the crucial enabler of the SDN paradigm: they maintain the logically centralized network state to be used by applications and act as a common intermediary with the data plane. Controllers are responsible for installing flow rules on switches that dictate how traffic should be handled. This can be done in a proactive, reactive or hybrid fashion (see section 2.1). Figure 1.2 shows the normal execution in an SDN environment when a switch receives a packet that does not know how to forward. Note that, usually, only the first packet of a specific flow needs controller intervention. Upon receiving this packet, the controller will install flow rules on the switches with a specific timeout depending on the controller needs. These flow rules allow the switches to forward subsequent packets directly without contacting the controller.

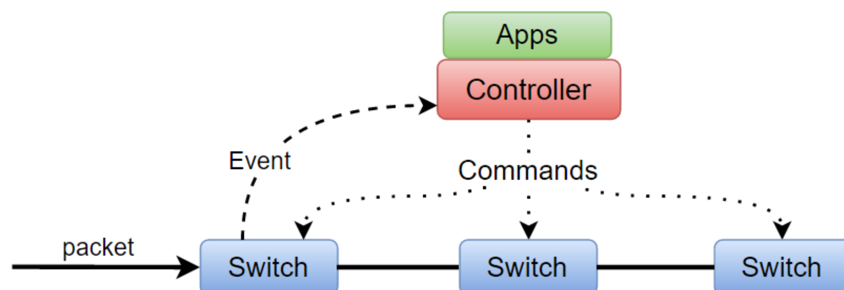


Fig. 1.2: SDN flow execution. Switches send events to the controller as needed and the controller replies with one or more commands that modify the switch's tables.

Martin Casado and Teemu Koponen, two of the first researches advocating SDN, have asked the question of what makes a good controller Application Programming Interfaces (APIs), arguing that controllers should not be yet standardized so that they can evolve without constraints [7]. Additionally, they describe three classes of controller platforms for SDNs: (i) application specific controllers, built with a single goal in mind, (ii) protocol specific controllers, that provide an interface based on the protocol (e.g., OpenFlow) and (iii) general SDN controllers that fully decouple the control applications from the underlying protocols. As noted in [1], classes (ii) and (iii) need to address many challenges such as:

- Generality: the API provided by the controller must be generic enough to allow applications to deliver a wide range of functionality in a variety of contexts;
- Scalability: the controller itself should not be a point of scalability problems;
- Reliability: the control platform must handle failures (of controllers, switches and links) gracefully;
- Simplicity: the control platform should provide common abstractions to ease the development of management applications;
- Performance: the additional latency introduced by the control plane should be reasonable for the provided mechanisms.

For simplicity reasons, the original SDN architecture was designed to use a centralized control plane (i.e., only one controller serving switches at each time) [8]. This design has inherent scalability problems as the number and size of the network increases (i.e., the number of events grows with the number of switches, latencies grow with the size of the network, and flow setup times will grow significantly). It is important to note that this general idea that SDN has scalability problems is not strictly related to SDN itself – traditional control protocols face the same challenges – and the centralized design is part of its historical evolution [9]. The main benefit of SDN over traditional networks is that SDN eases the task of overcoming these common challenges by separating concerns and, as such, facilitating the development of new network applications. Additionally, this centralized control plane with only one controller managing the networks, is a single point of failure that we want to avoid. To overcome these design limitations in terms of reliability, there has been a significant number of proposals, explored in chapter 2.

1.2 Motivation

As stated before, SDN provides a *logically* centralized control plane, which does not necessarily mean that the control plane is composed by only one controller machine. In

fact, multiple controllers should coordinate themselves to provide this logically centralized control plane. Trivially, a single controller would be a single point of failure for the whole system because the switches would lose their *intelligence* (the control plane) and consequently become unable to make new decisions (although backup rules can be pre-installed). By distributing the switches across multiple controllers, we can offload each controller, achieving better performance, and tolerate faults, which is very important if we consider an SDN in operation.

In SDN, the controllers enable the development of simple and modular applications because they provide fundamental abstractions like control plane distribution, fault tolerance, consistent updates and others [10]. For example, if a controller has the ability to tolerate faults in the system, applications will not be aware of faults and will still operate correctly in their presence. The same idea is valid for application state. In a logically centralized, but distributed and fault-tolerant control plane, the controllers need to maintain a consistent and updated global network view. Ideally, applications would not be aware of the distribution and replication of their own state (i.e., centralized applications) and would operate always on an updated and consistent network view. This is particularly important in a fault-tolerant control plane. In case of contoller failure, if the network view of the new controller (that will manage the switches) is not consistent with the one of the crashed controller, applications will operate in a stale network view, which can significantly degrade the performance of the system [11, 12].

To build a consistent global network view across replicated and fault-tolerant controllers, events (packets) need to be processed in a consistent way that guarantees three properties: (i) events are processed in the same (total) order in all controllers, (ii) no events are lost (processed at least once) and (iii) no events are processed repeatedly (at most once). These properties ensure that all controllers will reach the same internal state and thus build a consistent network view.

However, building a consistent network view in the controllers is not enough to offer a consistent logically centralized controller. In SDN, it is necessary to include switch state into the system model and handle it consistently [12]. Since switches are programmed by controllers (and controllers can fail), there must be mechanisms to ensure that controllers actually send commands to the switches (at least once), but never send repeated commands (at most once). In a fault-tolerant scenario, if a controller fails while it is processing an event, it may have sent or not some commands. A naive approach to avoid that no commands are missed would be for the new controller to simply repeat all commands for that event. However, this could lead to the switch receiving duplicate commands which would result in his state becoming inconsistent (with what is expected by the controller and applications), since some commands are not idempotent and can cause unexpected forwarding to in-flight packets. A simple example of a non idempotent command would be a Packet Out message. If the crashed master sent the Packet Out message and the

new master re-sends this command, the switch will simply forward the same packet twice through the same port, which would result in one additional packet in the network (which may be undesirable in certain networks).

Summing up, to achieve a consistent SDN environment, we need to ensure that the following properties are met:

- **Total Event Ordering:** Controller replicas should process events in the same order and subsequently all controller application instances should reach the same internal state
- **Exactly-Once Event Processing:** All the events are processed, and are neither lost nor processed repeatedly.
- **Exactly-Once Execution of Commands:** Any given series of commands are executed once and only once on the switches.

This type of consistency (in both controllers and switches), provided by controllers, offers full transparency to network applications, and makes controllers more reliable and developer friendly.

To the best of our knowledge, the offer of fault-tolerance and consistency for SDN control planes has only been addressed by Naga Katta et al. in [12]. To achieve these properties, however, Ravana (the system proposed in [12]) requires modifications to the OpenFlow protocol and to existing switches. Specifically, switches need to maintain two buffers, one for events and one for commands, and four new messages need to be added to the protocol – this is further explained in 2.4.2. These modifications preclude the adoption of Ravana on existing systems and hinder the possibility of it being used in the near future (as there are no guarantees these additions would be added to OpenFlow, for instance). They are, thus, the main motivation of our work of achieving the same consistency guarantees *without* requiring changes to OpenFlow or modifications to switches.

1.3 Goals and Contributions

The main goal of this work is to develop a transparent control plane that allows unmodified network applications to run in a consistent and fault-tolerant environment. At the same time, no changes should be required to the protocol between the control and data planes (e.g., OpenFlow) nor to the underlying hardware (e.g., switches). This is an important requirement since it allows the system to be used with existing applications, protocols and hardware.

With this in mind, we propose: 1) the design of a protocol that achieves the above requirements and 2) an implementation and evaluation of a controller – Rama – that implements the protocol. Rama is an OpenFlow controller built on top of Floodlight [13]

that achieves the same consistency and fault tolerance guarantees as Ravana [12], but without modifications to the OpenFlow protocol or to switches.

Additionally, as requisites that were result of our implementation and evaluation phases, two additional contributions are to be mentioned here:

1. A new feature that allows the inclusion of Packet-Out messages inside Bundles was added to OpenvSwitch [14] and integrated into the code line of this widely used software switch.

2. An extended version of Cbench [15], a benchmarking tool for OpenFlow controllers, that allows it to emulate OF 1.4 switches (instead of OF 1.0 switches) was implemented.

Furthermore, some contributions were made to the Floodlight project [13], the base controller where Rama was built on, to handle bundle-related messages and small fixes and improvements.

1.4 Additional Outcomes

In February we submitted an abstract [1] for a poster to the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16) conference. The abstract was accepted and the poster [2] was presented in NSDI'16 poster session [3], generating interesting discussion, including with the authors of Ravana [12].

We will submit a long paper with our proposal to SOSR'17 in the end of October.

We have also contributed to deliverable 4.2 of the H2020 Project SUPERCLOUD with this work.

1.5 Document structure

Chapter 2 gives context about Software-Defined Networks and summarizes the related work in the literature with a focus on scalability, consistency and reliability of SDN.

In chapter 3, we present the design of Rama, the control plane architecture we propose, consisting of controller replicas and a coordination service, as well as a novel consistent and fault-tolerant protocol.

The implementation of the proposed protocol is explained in section 4 with the corresponding evaluation in section 5.

Finally, chapter 6 gives some concluding remarks and describes future work.

[1] https://arxiv.org/abs/1602.04211
[2] https://www.dropbox.com/s/da9m4pdfxwfyl7c/rama-poster-nsdi16.pdf
[3] https://www.usenix.org/conference/nsdi16/poster-session

Chapter 2 – Context & Related Work

This chapter presents the general idea of how SDN works so that we get a clearer understanding of the possibilities it brings, its problems and challenges (section 2.1). Then, in the following sections (2.2, 2.3 and 2.4), we dive into different classes of SDN problems and how some of the works in the literature address them.

2.1 Context

To understand the building blocks of SDN, we first need to look at the three planes of traditional networks:

- Data plane: the network nodes that are responsible for forwarding packets.
- Control plane: the network logic that consists of distributed protocols to build and manage the forwarding tables.
- Management plane: software tools used to remotely monitor and configure how the control plane works and to define network policies.

The main problem with the architecture of traditional networks is that the control and data planes are tightly coupled in the same network devices. Despite the proven Internet success, this problem results in networks that are inflexible, and complex to manage and control. Additionally, this architecture hinders the design and deployment of innovative network protocols because a change requires complicated modifications in every network node.

SDN learns from the traditional networks *mistakes* and, as defined by the Open Networking Foundation[1] [16], defines an architecture that is:

- Directly programmable: Network control is directly programmable because it is decoupled from forwarding functions.
- Agile: Abstracting control from forwarding lets administrators dynamically adjust network-wide traffic flow to meet changing needs.

[1] Open Networking Foundation is a user-driven organization dedicated to the promotion and adoption of Software-Defined Networking through open standards development.

- Centrally managed: Network intelligence is (logically) centralized in software-based SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical switch.

- Programmatically configured: SDN lets network managers configure, manage, secure, and optimize network resources very quickly via dynamic, automated SDN programs, which they can write themselves because the programs do not depend on proprietary software.

- Open standards-based and vendor-neutral: When implemented through open standards, SDN simplifies network design and operation because instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

To have a clearer picture of the SDN architecture, lets decompose it in three layers (see figure 2.1). Each layer is equivalent to one plane in traditional networks. The application, control and infrastructure layers correspond to the management, control and data planes, respectively. As explained, in SDN, the control and data planes are physically decoupled, but the management plane can coexist with the control plane in the same device (the controller).

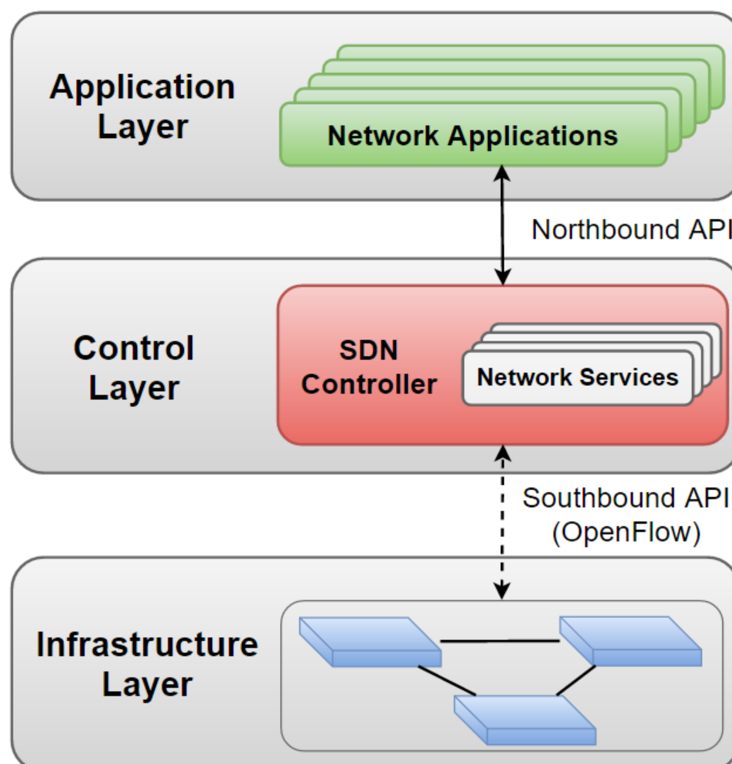


Fig. 2.1: SDN architecture in layers

The SDN concept was born with the OpenFlow protocol [2] that was built with the goal of providing a way for researchers to run experimental protocols in campus networks. OpenFlow is an open standard that defines an interface to allow switches and controllers to communicate (e.g., defines message types and fields). The protocol is implemented as a feature in commercial Ethernet switches.

The main components of an OpenFlow switch are shown in figure 2.2. The switch has a software component that implements the OpenFlow protocol and communicates with the controller using a TCP connection (it can be secure or not). This component can dynamically change the flow tables (hardware) that are used to match and forward packets (as in regular switches).

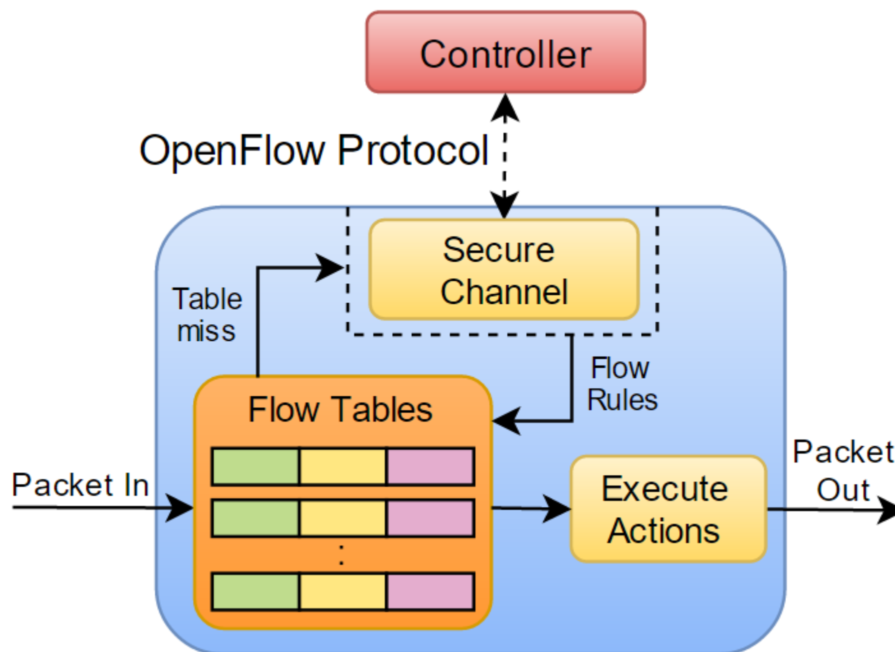


Fig. 2.2: Main components of an OpenFlow enabled switch

A switch can have several flow tables through which incoming packets are processed in pipeline. Each flow table contains multiple entries used to match and forward packets. When the switch receives a packet, it is matched against the table entries from top to bottom. A (simplified) structure of a flow table entry is shown in figure 2.3.

If a packet does not match any table entry, the switch will generate an event with the packet in question and send it to the controller. The controller will decide how to handle the event, which usually includes installing flow rules that match the packet and forward it.

This event, denoted as `PACKET_IN`, is the most common event generated by OpenFlow switches. Other events are generated when, for example, a flow entry is removed, or port status modifications (e.g., link down). In addition to these events generated by switches, the controllers can also send specific messages to query or configure the

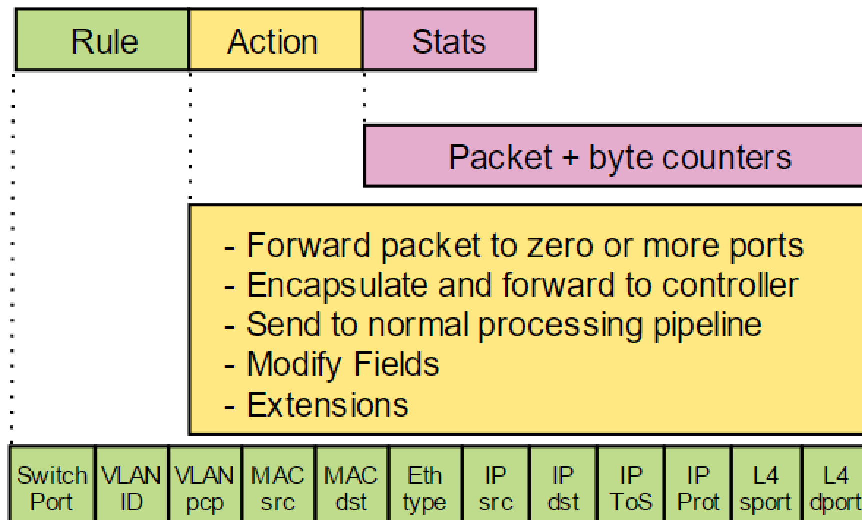


Fig. 2.3: OpenFlow table entry structure

switches.

OpenFlow has evolved over the years and enables new features with each release. Two particular features are of interest to our work. First, OpenFlow 1.3 introduced the master-slave model where switches can be connected to at least two controllers with different roles. For each switch, only one controller has the master role, while one or more controllers may have the slave role. By default, switches send every event to the master controller but only some events (e.g., port down events) are sent to the slave controllers. Secondly, the addition of Bundles in OpenFlow 1.4 allows controllers to have more control on how switches process commands. Controllers can open bundles and add one or more commands to that bundle before closing and committing each bundle. According to the protocol, the commit can be ordered (commands must be processed in the order they were sent by the controller), atomic (all commands must be processed successfully or else no command can be processed), or both, but it is not mandatory that switches implement all combinations. Upon committing a bundle, the switch must send an acknowledgement message to the controller, the *Commit Reply* message, indicating if the bundle commit was successful or not. Section 3.3 explains how we use these two mechanisms introduced by OpenFlow to achieve a consistent and fault-tolerant controller.

The SDN design has some specific concerns with scalability and resilience which we will discuss in the next sections. To overcome the scalability limitations of SDN, different techniques have been proposed. These techniques can be classified into two main categories: (i) improving the control plane with well-known distributed systems techniques, and (ii) moving the intelligence into the switches to offload the controller. In addition, recent works complement these proposals by making Software-Defined Networks more secure, resilient and robust to both applications and component faults.

2.2 Scaling SDN in the control plane

The majority of research work to address the scalability issues in SDN focuses on the control plane, since it does not require modifications to existing network elements.

2.2.1 Improving control plane performance

NOX [8] was the first controller to provide a higher level of abstraction for applications to program the network. However, NOX can only handle 30K new flow installs per second while maintaining a sub-10ms flow install time [17], which is way far from desired: a 1500 server cluster might generate 100K requests per second [18] and a 100 switch data center might generate 10000K requests per second [19].

With this in mind, some works focused on improving the control plane performance by using multithreaded designs to explore the parallelism of multi-core processors [20, 21, 22, 13]. These implementations use common distributed system techniques such as parallelism, I/O batching [20, 21, 22], shared queues [21] and pipelining [20], while also maintaining a simple programming model for developers (i.e., they can write single threaded applications) [20, 21].

Maestro [21] was the first controller exploiting parallelism to achieve near linear performance scalability on multi-core processors. It uses a multithreading design that distributes the work evenly across worker threads and aims to reduce the cross-core overheads.

Beacon [20] not only optimizes the controller performance but it was also designed to help application developers and network administrators. It provides a platform with tools to ease the development of new applications and enables runtime modularity (new and existing applications can start and stop during runtime).

NOX-MT [22], a multi-threaded successor of NOX, was able to outperform NOX by a factor of 33 with just the use of I/O batching, Boost Asynchronous IO (ASIO) and a multi-threaded malloc implementation.

2.2.2 Distributing the control plane

One key technique to scale the control plane is to distribute it. In this approach, physically distributed controllers work together to create a logically centralized global network view. The distribution can be flat (horizontal) or hierarchical and each controller is responsible for a subset of the network switches, forming a controller domain. While distributed architectures have many advantages (e.g., scalability, switch-controller latency, fault-tolerance and load balancing), the coordination between the controllers to achieve a consistent network view brings new challenges.

Levin et al. [11] explore the trade-offs of state distribution in a distributed control plane and motivate the importance of strong consistency in applications' performance.

On the one hand, view staleness affects the correct operation of applications, which may lead to poor network performance. On the other, applications need to be more complex in order to be aware of possible network inconsistencies.

One possible way to optimize the inter-controller coordination (and thus increase performance) is to make use of local algorithms to perform global tasks, as proposed in [23]. The main idea is that each controller instance only needs to reply to events that take place in its local neighbourhood (domain) in order to complete the algorithm, which greatly reduces inter-controller coordination in large scale networks.

The first work to explore and provide a distributed control platform for applications was Onix [1]. It introduced key concepts like the Network Information Base (NIB), partitioning and aggregation. Applications use a set of general APIs that facilitate the access to network state (NIB), which is distributed and replicated over Onix instances to achieve scalability and resilience. Each controller imports/exports its NIB to/from a data store and registers for notifications to keep it updated with other controller instances. Onix also provides two data stores so that applications can choose their trade-offs between consistency and performance.

Proposals like HyperFlow [24] and ONOS [25] take a different approach to maintain a global network view: they use a publish/subscribe system for inter-controller communication (i.e., exchange network state updates and indirectly program switches from other controllers). HyperFlow's publish/subscribe system is tolerant to network partitioning and also serves for controller discovery and failure detection. However, because HyperFlow is implemented as a NOX application, it incurs into unnecessary complexities and requires applications to be modified. On the other hand, ONOS, a distributed and open source SDN controller for wide-area networks, uses ZooKeeper [26] to manage and establish the switch-to-controller mastership. ONOS also provides a simple API for network applications that abstracts the network view without exposing unnecessary implementation details (unlike Onix [1]).

To extend the distribution techniques presented in previous works, ASIC [27] proposes an intra-domain architecture with three layers, where each one has its own scalable solutions and can be adapted independently according to the network demands. First, a load balancing layer distributes the network events to the controllers cluster (second layer) that uses a distributed and persistent storage system (third layer) to achieve a consistent global network view.

DISCO [28], an open and extensible controller built on top of Floodlight [13], uses similar techniques but addresses the challenges of deploying a multi-domain SDN system (e.g., its heterogeneous nature). It leverages on Advanced Message Queuing Protocol (that supports multiple protocols like OSPF, RSVP, BGP) to implement its publish/subscribe system. Network agents use this channel to share aggregated network-wide state to provide end-to-end network services and mobility management. A DISCO

controller manages the switches in one domain and maintains separated intra and inter network state in an extended table. DISCO use cases include inter-domain topology disruption, end-to-end priority service request and virtual machine migration.

While proposals like DIFANE [29] and DevoFlow [30] (expanded in the next section) aim to reduce the number of events reaching the control plane by extending the mechanisms at the switches, Kandoo [31] achieves the same goal but without requiring switches to be modified. Kandoo is based on the idea that some applications are local, and do not depend on the global network state (e.g., learning switch and elephant flow detection), to separate the control plane in two hierarchies. In this scenario, frequent events can be processed by local controllers (which run local applications close to the switches) while the logically centralized root controllers (i.e., Onix [1], HyperFlow [24]) deal with rare events. In addition, a publish/subscribe system is used to exchange information between the controllers (i.e., subscribe to events or ask for network state). Kandoo is able to automatically distribute application state based on a flag that each application must set to indicate if they need local or global state, which makes it not being completely transparent but easy to use nevertheless.

Beehive [32] extends the idea adopted by Kandoo to implement a platform that generates and optimally deploys a distributed version of centralized applications across multiple controllers. The platform uses techniques such as transactions, replication and fault-tolerance to provide applications with concurrent and consistent state access in a distributed fashion. Because finding the optimum placement of applications is NP-hard, Beehive employs a greedy heuristic aiming at processing messages close to their source.

2.3 Scaling SDN in the data plane

Because the data plane is programmed by the control plane that maintains a global view, the OpenFlow design incurs in additional costs when compared to traditional networks – flow setup costs (on both the controller and network) and flow statistics costs. These overheads limit the scalability of the control plane due to the high amount of events that controllers may receive.

With this in mind, and as mentioned before, DIFANE [29] and DevoFlow [30] reduce the number of events received by the control plane by adding new functionalities at the switches. Their main idea is to delegate some work to these extended switches to offload the controllers.

DIFANE [29] proposes that controllers should delegate work to authority switches so that they do not need to be involved in the flow setup. Rules generated by the controllers are partitioned and distributed across the authority switches whose job is to act as controllers for regular switches. DevoFlow [30] attempts to perform a reasonable trade-off between controller load and network visibility with two main mechanisms: 1)

pre-installed wildcard rules are installed to delegate decisions to the switches as much as possible while maintaining a central control and visibility over important flows (e.g., security, QoS); 2) efficient statistics collection techniques implemented in the switches reduce switch-controller network bandwidth.

While partially solved by DevoFlow, this class of proposals choose to trade fine-grained detailed flow-level visibility in the control plane for scalability, which can be reasonable depending on the setting and constraints of the underlying network. In this work, we also try to stay away from anything that involves changing the OpenFlow protocol or the switches.

2.4 Resilience in SDN

Having a strongly consistent network view across the controllers may be critical to the operation of some applications (e.g., load balancing) in terms of correctness and performance [11]. However, as noted in the CAP theorem [33], a system can not provide availability while also achieving strong consistency in the presence of network partitions. Because of this, fault-tolerant and distributed SDN architectures must use techniques to explicitly handle partitions in order to optimize consistency and availability (and thus achieving a tradeoff between them) [34]. Section 2.4.2 focuses on detailing proposals that tolerate faults in SDN.

Part of the strong consistency in the controllers comes from a consistent packet processing (i.e., packets received from switches). OF.CPP [35] explores the consistency and performance problems associated with packet processing at the controller and proposes the use of transactional semantics to solve them. These semantics are achieved by using multi-commit transactions, where each event is a sub transaction, which can commit or abort, of the related packet (the main transaction). However, this transactional semantics in the packet processing are not enough as discussed in the previous sections: controllers should also coordinate to guarantee the same semantics in the switches state. Specifically, the commands sent by the controllers should be processed exactly once by the corresponding switches (to achieve consistent command processing – *a problem our work addresses*).

Furthermore, as each controller manages a set of network switches, a single event can cause applications to install commands on multiple switches. Therefore, it is important to consider how in-flight packets will be processed by switches during an update to the network policy (i.e., the global set of rules installed in the data plane). Ideally, a packet should be forwarded by only one policy and never be forwarded by a combination of an old policy in one switch, and later processed by a new policy in another switch – a property called *consistent network updates*. Because applications should be as simple as possible, the control plane should abstract the need to perform these consistent network

updates [36]. This way, applications do not need to deal with the interactions of old a new network policies when installing forwarding rules in multiple switches. Some proposals for this class of reliability are discussed in section 2.4.1.

2.4.1 Consistent network updates

The concepts of per-packet and per-flow consistency model in SDN were introduced in [36] to provide a useful abstraction for applications: consistent updates. With consistent updates, packets or flows in flight are processed exclusively by the old or by the new network policy (never a mix of both). A network policy can be seen as the set of rules installed on all switches in the network at a given moment. For example, with per-packet consistency, every packet traversing the network is processed by exactly one consistent global network configuration. The authors extend this work in [37] and implement Kinetic, which runs on top of NOX [8], to offer these abstractions in a control plane to be used by applications. The main mechanism used to guarantee consistent network updates is the use of a two-phase protocol to update the rules on the switches. First, the new configuration is installed in an unobservable way (no packets go through these rules yet). After, the switch's ingress ports are updated one-by-one to stamp packets with a new version number (using VLAN tags). Only packets with the new version number are processed by the new rules.

In [38], Canini et al. extend Kinetic to a distributed control plane and formalize the notion of fault-tolerant policy composition. Their algorithm also requires a bounded number of tags, regardless of the number of installed updates, as opposed to the unbounded number of tags in [37].

This class of proposals addresses consistent network updates, which is orthogonal to the work presented here, since this work focuses on consistency issues inside the controller (consistent packet processing) and on the switches (consistent command processing).

2.4.2 Fault-tolerance

As for fault-tolerance, Kim et al. [39] identified three fault domains in SDN: (i) switches and links between them (*data plane*), (ii) inter-controller links and switch-controller links (*control plane*) and (iii) the controllers' machines. In this section we start by exploring domains (i) and (ii) and then move to domain (iii).

To address faults in the data plane, [39] and [40], specific proposals for this subject, take different approaches. CORONET [39], a fault-tolerant SDN architecture, provides fast recovery and is able to recover from multiple link failures. It incurs into minimal control traffic, by changing VLAN configuration after detecting faults in switches or links, using the OpenFlow API. Alternatively, FatTire [40] proposes a new programming lan-

guage that eases the development of fault tolerant applications by compiling regular expressions into OpenFlow rules. This provides an easy way for developers to specify the set of paths that packets can take in the network and the degree of fault-tolerance required.

As a more general approach to handle faults in fault domain (i), most OpenFlow controllers take a reactive approach: port-down events sent by switches are fed to applications that program switches accordingly (i.e., reroute traffic around failures). Nonetheless, this approach incurs into high restoration time and additional load on the controllers. A possible solution is to have pre-installed backup paths in the switches to avoid the overhead of communicating with the control plane.

Covering fault domain (ii), Ros et al. [41] address the problem of dynamically adapting the number and locations of controllers, introduced in Bari et al. [42], but for fault-tolerance. The goal is to determine the optimal number, placement and switch-connections of controllers in order to achieve at least five nines of reliability in the southbound interface (connectivity between controllers and switches).

Distributed control plane architectures can achieve fault tolerance in domains (ii) and (iii) in the sense that if one controller fails, the switches managed by this controller start to be managed by a new one. However, this does not guarantee that the new controller will always have a consistent network view. The most common approach is to use the master-slave model introduced in OpenFlow 1.3 (see section 2.1), which is used in [13, 25, 43, 44, 45, 12]. ASIC [27] tolerates controller faults because the load balancer (assuming it does not fail) will only distribute requests across available controllers. In contrast with these approaches HyperFlow [24] directly reconfigures switches to connect to a new controller. Some proposals only plan to address fault-tolerance in future work [28, 32].

Deserving special attention, the works by Botelho [43] and Katta [12] address fault tolerance in the control plane while achieving strong consistency. SMaRtLight [43] proposes a fault-tolerant controller architecture for small to medium networks and analyses the costs of fault-tolerance in SDN. The architecture uses a hybrid replication approach: passive replication in the controllers (one primary and multiple backups) and active replication in a distributed data store to achieve durability and strong consistency. The controllers are coordinated through the data store and achieve good performance by using caching mechanisms. However, SMaRtLight requires that applications are modified to use the data store directly, contrary to our approach, and it does not consider the consistency of the interactions between the controllers and the switch state.

The closest work to ours is Ravana [12] that provides a transparent fault-free control platform for applications in the face of both controller and switch crashes. To achieve this, Ravana processes control messages transactionally and exactly once (at both the controllers and the switches) using a replicated state machine approach, but without involving the switches in a consensus protocol. The protocol used by Ravana in show in figure 2.4. Switches buffer events (in case they need to re-transmitted) and send them to the master

controller that will replicate them in a shared log with the slaves and reply back to the switch acknowledging the reception of the events. Then, events are delivered to applications that will generate and send one or more commands to the switches. Switches reply back to acknowledge the reception of these commands and buffer them to filter possible duplicates.

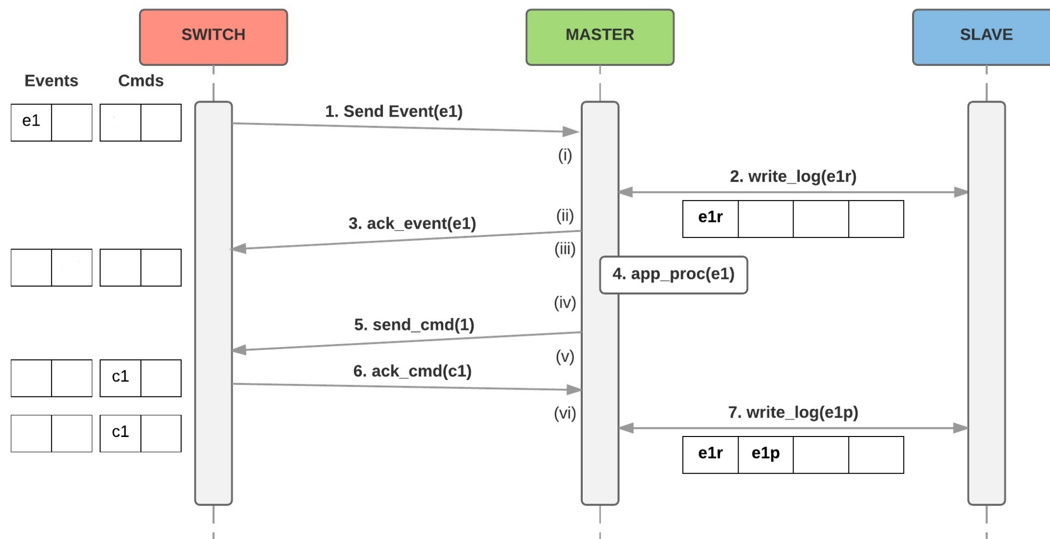


Fig. 2.4: Ravana protocol. In Ravana switches maintain two buffers (displayed on the left) to re-transmit events and filter repeated commands in case of master failure. New acknowledge messages (`ack_event` and `ack_cmd`) are exchanged between the switch and the master to guarantee the consistency requirements.

While Ravana allows unmodified applications to run in a fault-tolerant environment, it requires modifications to the OpenFlow protocol and switches: Ravana leverages on buffers implemented on switches to retransmit events and filter possible repeated commands received from the controllers; explicit acknowledge messages must be added to the OpenFlow protocol so that the switch and the controller acknowledge received messages.

Despite the very clear value of resorting to more advanced features in switches, the fact is that it takes time for the protocol (and subsequently switches) to be changed to include the necessary features. This motivates our work of, starting from Ravana, achieve the same guarantees without the need to modify the protocol or the switches.

This chapter addressed the main concepts of SDN, its problems and challenges and listed some of the works in the literature that were somehow relevant to the problem we want to tackle. In the next chapter we describe the Rama protocol in detail and the

challenges it must surpass to achieve the desired consistency properties without modifying the OpenFlow protocol or the switches.

Chapter 3 – Rama Design

In the previous chapter we summarized how SDN operates, explored some of its challenges and problems, and the relevant proposed solutions. In particular we discussed the challenge that we focus on: maintaining a consistent control and data plane in the presence of faults. The goal of our work is to build a strongly consistent and fault-tolerant control plane for SDN, without modifying switches, to be used transparently by applications. This chapter describes the architecture and protocol for such control plane which is driven by the following requirements:

- Reliability: the system should maintain a correct and consistent state even in the presence of failures (in both the controllers and switches).
- Transparency: the consistency and fault-tolerance properties should be completely transparent to applications and switches.
- Performance: the performance of the system should not degrade as the number of network elements (events and switches) grows.

3.1 Architecture

The proposed architecture for our system, Rama[1] is depicted in figure 3.1. We have decided to name our system Rama since the original idea behind the protocol was inspired by Ravana [12] (see the description of Ravana in section 2.4.2). The main components are: (i) OpenFlow enabled switches (switches that are implemented according the OpenFlow specification), (ii) controllers that manage the switches and (iii) a coordination service that abstracts controllers from complex primitives like fault detection and total order. In our model, we consider only one network domain with one primary controller and one or more backup controllers, depending on the number of faults to tolerate. Each switch connects to one primary controller and multiple (f to be precise) backup controllers. This primary/backup model is supported by OpenFlow in the form of master/slave and allows the system to tolerate controller faults. When the master controller fails, the remaining controllers will elect a new leader to act as the new master for the switches managed by the crashed master. This election is supported by the coordination service.

[1] In the Hindu epic Ramayana, Rama kills the evil demon Ravana, who abducted his wife Sita.

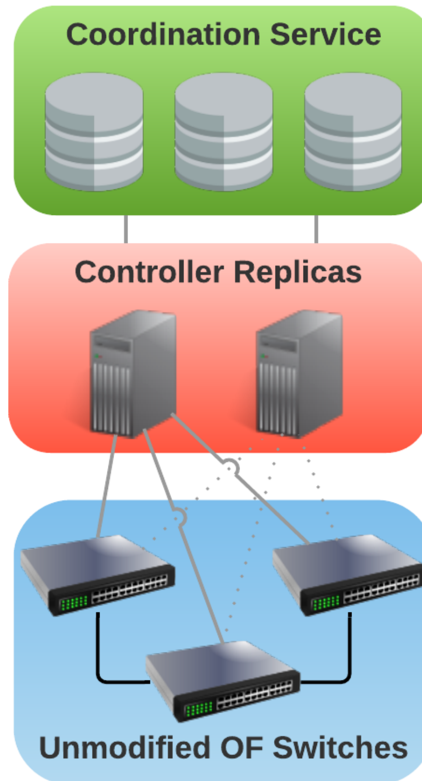


Fig. 3.1: High level architecture of the system. One controller manages the OpenFlow (OF) switches (master) and one or more backup controllers are ready in case of failure (slaves, connected in dashed line). All controllers keep an updated and consistent network view using the coordination service. Network applications run inside controller replicas.

The coordination service offers strong consistency and abstracts the controllers from coordination and synchronization primitives, making them simpler and more robust. Note that here we usually want to have a number of replicas equal to $2f+1$ with f being the number of faults to tolerate. The strong consistency model assures that updates to the coordination service made by the master will only return when they are persistently stored. This means that slaves will always have the fresh modifications available as soon as the master receives confirmation of the update. This results in a consistent network view among all controllers even if some fail. In addition to the controllers' state, the switches also maintain state that must be handled consistently in the presence of faults. This will be discussed in section 3.3.

3.2 Why Consistency Matters

In this section we summarize the importance of maintaining a consistent state in both controllers and switches in a fault-tolerant SDN setting. The first step to have fault-tolerance in the control plane is to have more than one controller available to manage the network switches, which can lead us to some problems.

3.2.1 Inconsistent event ordering

In OpenFlow, each switch maintains a TCP connection with each controller it knows and sends them messages using these channels. If we configure switches to send all their events to all known controller replicas, and let replicas process events as they are received, each one will end up building a different internal state. This is because, although each TCP channel orders events sent by each switch, there is no ordering guarantee between the events sent to controllers by all switches.

Consider a simple scenario with two controllers (c1 and c2) and two switches (s1 and s2) that send two events respectively (e1, e2 and e3, e4). One possible outcome where both controllers receive events in different order (while respecting the TCP FIFO property) is c1 receiving events in the order e1, e3, e2, e4 and c2 receiving in the order e3, e4, e1, e2.

As a result of this consistency problem we derive our first design goal for a fault tolerant and consistent control plane:

> **Total event ordering:** all controllers should process the same (total) order of events and consequently reach the same internal state.

3.2.2 Unreliable event delivery

In order to achieve a total ordering of events between controller replicas two approaches can be used:

1. The master (primary) replica can store controller state (including state from network applications) in an external consistent data-store (as in Onix [1], ONOS [25] and SMaRtLight [43]);

2. The controller replicas can maintain a consistent state using replicated state machine protocols (as in Ravana [12]).

Although both approaches ensure a consistent ordering of events between controller replicas, they are not fault-tolerant in a standard case where only the master controller receives all events.

If we consider – for the first approach – that the master replica can fail between receiving an event and finishing persisting the controller state in the external data-store (which happens after processing the event through controller applications), that event will be lost and the new master (i.e., one of the other controller replicas) will never receive it. The same can happen in the second approach: the master replica can fail right after receiving the event and before replicating it in the shared log (which in this case happens before processing the event through the controller applications). In these cases, since only

the crashed master received the event, the other controller replicas will not have an updated view of the network and, depending on the type of the lost event or on the type of controller applications running on the controller, this can cause serious performance or security problems.

However, a solution used to solve the problem above cannot be careless. Additionally to not losing events, it is also important to not process them repeatedly in each controller replica, since that would also lead to controllers having an inconsistent view of the network. From both these problems comes our second design goal:

> **Exactly-once event processing:** all events sent by switches are processed and are never lost nor processed repeatedly.

3.2.3 Repetition of commands

In SDN, mechanisms to ensure exactly-once event processing are not enough to achieve a consistent system in the presence of faults. This is due to the state maintained by switches taking an important role on how the whole system works.

Consider the case where the master replica is processing an event that generated three commands to be sent to one switch and the slave replica has knowledge of this event. If the master fails while sending these commands, the new elected master will process the event (to reach an updated state) and may send repeated commands. This happens because the old master failed before informing the slave replica of its progress (i.e., which commands have been sent for that specific event) and therefore it cannot decide which commands to send and which commands to filter.

Additionally, to make things worse, one of the differences between traditional client-server models and SDN is that controllers (servers) may reply (i.e., send commands) to multiple switches (clients) as a result of one event sent by a switch. Therefore it is essential to integrate switch state into a consistent and fault-tolerant protocol and handle it carefully, which leads to our third and final design goal:

> **Exactly-once command execution:** any set of commands for a given event sent by controllers is executed only once on the switches.

3.3 Consistent and Fault-tolerant protocol

In an SDN setting, switches generate events (e.g., when they receive packets or when the status of a port changes) that are forwarded to controllers. The controllers run multiple applications that process the received events and may send commands to one or more switches in reply to each event. This cycle repeats itself in multiple switches across the network as needed.

In order to maintain a correct system in the presence of faults, one must handle the state in the controllers (the received events) and the state in the switches (the received commands) consistently. In this work, to ensure this, the entire cycle (figure 3.2) is processed as a transaction and exactly once. This means that (i) the events are processed exactly once at the controllers and in a total order (i.e., all controllers process events in the same order to reach the same state) and (ii) the commands are processed exactly once in the switches. Because the standard operation in OpenFlow switches is to simply process commands as they are received, the controllers must coordinate to send a command exactly once (since we do not want to modify the protocol or the switches). Ravana [12] does not need this coordination because the (modified) switches can simply buffer received commands and discard repeated commands (with the same identifier) sent by the new controller, as explained in section 2.4.2.

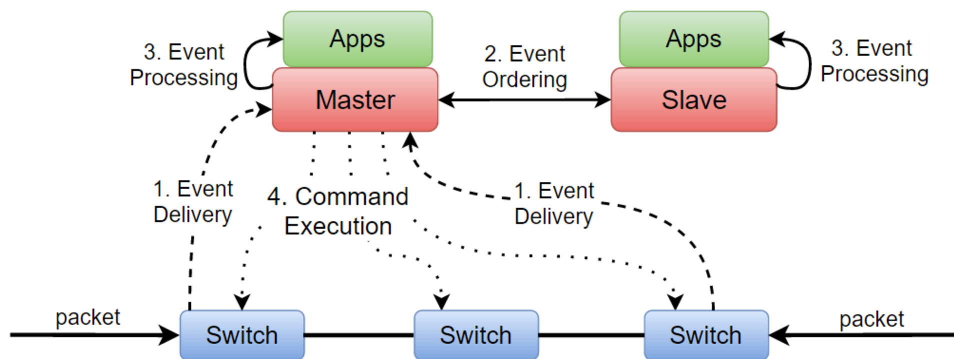


Fig. 3.2: Control loop of (1) event delivery, (2) event ordering, (3) event processing, and (4) command execution. Events are delivered to the master controller, which decides a total order on the received events. The events are processed by applications in the same order in all controllers. Applications issue commands to be executed in the switches.

By default, in OpenFlow, a master controller receives all asynchronous messages (i.e., `OFPT_PACKET_IN`) and the slaves controllers only receive some messages (i.e., port modifications). This means that only the master controller would receive the generated events from the switches. To change this behavior, the slaves send an `OFPT_SET_ASYNC` message to each switch that modifies the asynchronous configuration in a way that switches will send events also to the slaves. Alternatively, controllers can set their role to `EQUAL`, and the coordination (to decide who processes and sends commands) is done among controllers. In this case, by the OpenFlow protocol specification, switches send all events to every controller in role `EQUAL`.

The fault-free execution of the protocol is represented in figure 3.3. In this case, a switch is connected with one master controller and one slave controller. The main idea is that switches must send messages to *all controllers*, so that they can coordinate themselves even if some fail at any given point. Ravana, because switches simply buffer events (so

that they can be retransmitted to a new master if needed), switches can send events only to the current master, instead of to every controller.

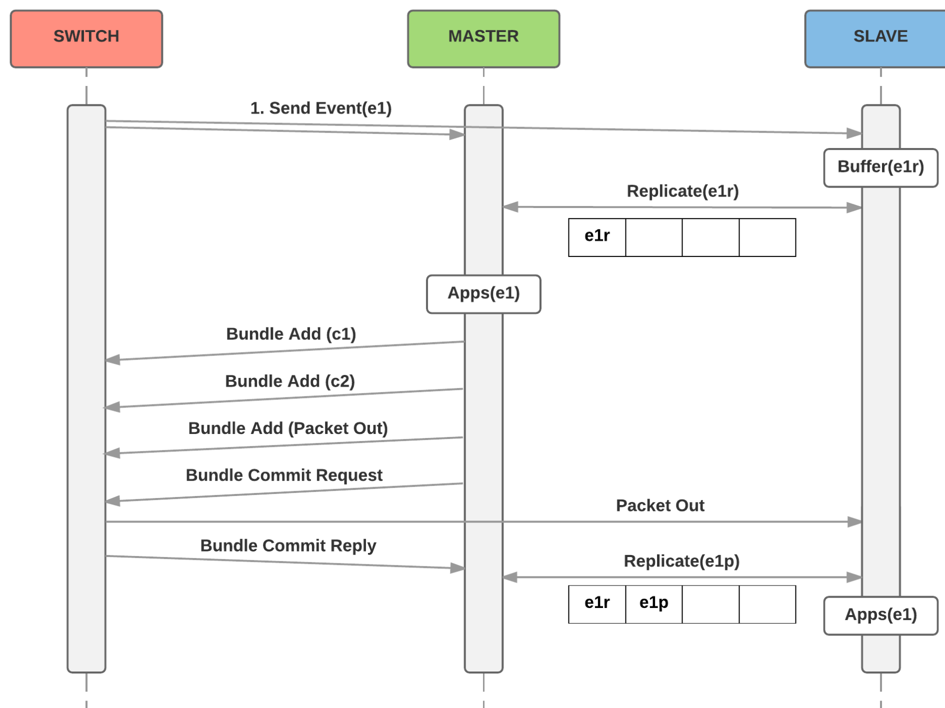


Fig. 3.3: Fault-free case of the protocol. Switches send generated events to all controllers so that no event is lost. The master controller replicates the event in the shared log and then feeds its applications with the events in log order. Commands sent are buffered by the switch until the controller sends a Commit Request. The corresponding Commit Reply message is forwarded to all controllers to make sure that a new master never tries to commit repeated commands.

The master controller replicates the event in a shared log with the other controllers that imposes a total order over the received events. Replicating events in the log can be seen as the master sending an update to the coordination service and the slaves receiving it (to simplify, the coordination service is omitted from the figures).

When the event is replicated across controllers, it is processed by the master controller applications, which will generate zero or more commands. The commands are sent to the switches in bundles (a feature introduced in OpenFlow 1.4, see figure 3.4). A controller can open a bundle, add multiple commands to that bundle and then tell the switch to commit the commands present in the bundle in an atomic and ordered fashion. If for some reason an event does not generate any commands, the master controller will still add one single message as part of the protocol (see below). Note that Ravana does not rely on bundles since switches buffer all received commands so that they can discard possible duplicates from a new master.

When the event is processed by all modules (each sent the commands they acquired),

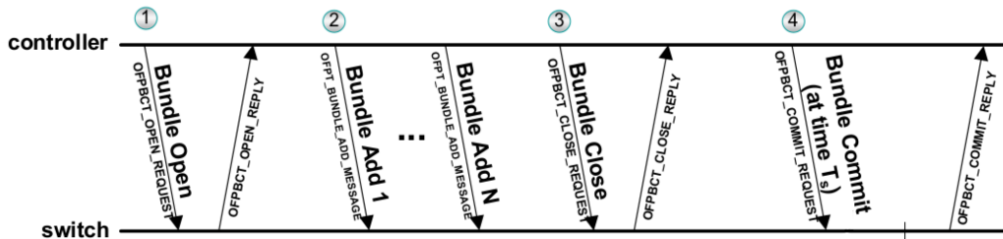


Fig. 3.4: OpenFlow Bundles.

the master controller sends a `OFPBCT_COMMIT_REQUEST` message to each switch affected by the event. The switch processes the request and tries to apply all the messages in the bundle by order. It then sends a reply message indicating if the Commit Request was successful or not. Again, we need to make sure that this reply message is sent to all controllers. This is a challenge, as it is not possible via the OpenFlow protocol. Bundle Replies are Controller-to-Switch messages and hence are only sent to the controller that made the request (in the same TCP connection). To overcome this challenge we inform other controllers if the bundle was committed or not (so that they can decide to resend commands), by including one `OFPT_PACKET_OUT` message in the end of the bundle with action `output=controller`. The result is that the switch will send the data attached in the `OFPT_PACKET_OUT` message to all connected controllers in a `OFPT_PACKET_IN` message. This data is set by the master controller and assures that slave controllers know which events were fully processed by the switch, so that they do not send repeated commands (and thus guaranteeing exactly-once semantics).

The master finishes the transaction by replicating an `event processed message` in the log, which tells the slaves controllers that they can safely feed the corresponding event in the log to their applications. This is done to simply bring the slaves to an updated state equal to the master controller (the resulting commands sent by the applications are discarded).

3.3.1 Fault cases

When the master controller fails, the other controllers will detect the failure (i.e., by timeout) and run a leader election algorithm to elect a new master for the switches. Upon election, the new master must send a Role Request message to each switch, to register as the new master. There are three main cases where the master controller can fail:

1. Before replicating the received event in the distributed log (figure 3.5),
2. After replicating the event but before sending the Commit Request (figure 3.6)
3. After sending the Commit Request message.

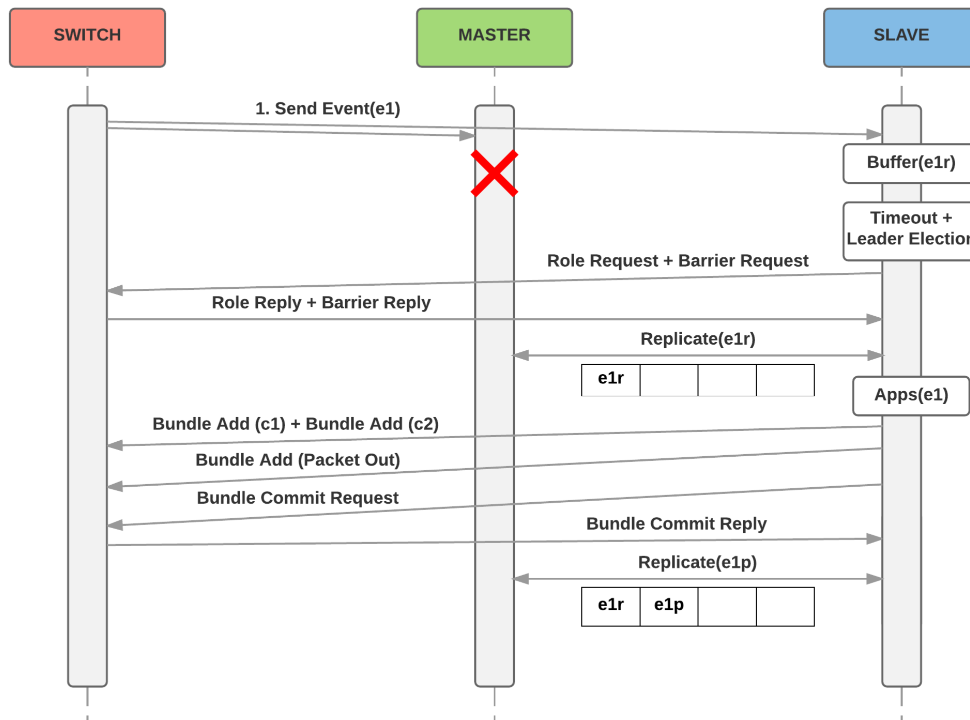


Fig. 3.5: Case of the protocol where the master fails before replicating the received event. Because the slaves buffer all events, the event is not lost and the new master can resume the execution of the failed controller. The new master chooses the order of the events to replicate in the log and can only append new events to the log.

In the first case, the master failed to replicate the received events to the shared log, but, since slave controllers receive and buffer all events, no events are lost. The new elected master appends the buffered events in order to the shared log and continues operation (feed the new events to applications and send commands). Note that before doing this, the new master must finish processing any events logged by the older master, having or not the corresponding `event processed message` in the log) - events marked as processed have their resulting commands filtered. This makes the new master reach the same internal state as the previous one before choosing the new order of events to append to the log (and it is valid for the other fault cases too).

However, if the event was indeed replicated in the log (cases 2 and 3), the crashed master may have already issued the Commit Request message. Therefore, the new master must carefully verify if the switch has processed everything it has received before re-sending the commands and the Commit Request message. To guarantee ordering, OpenFlow provides a Barrier message, to which a switch can only reply after processing everything received before (including generating and sending possible error messages). If a new master receives a Barrier Reply message without receiving a Commit Reply message (in form of `OFPT_PACKET_OUT`), it can safely assume that the switch did not receive

nor execute a Commit Request for that event from the old master (case 2)[1]. Even if the old master sent all commands but did not sent the Commit Request message, the bundle will never be committed and will eventually be discarded. Therefore, the new master can safely resend the commands. In case 3, since the old master sent the Commit Request before crashing, the new master will receive the confirmation that the switch processed respective commands for that event and will not resend them (otherwise we would break the exactly-once commands property).

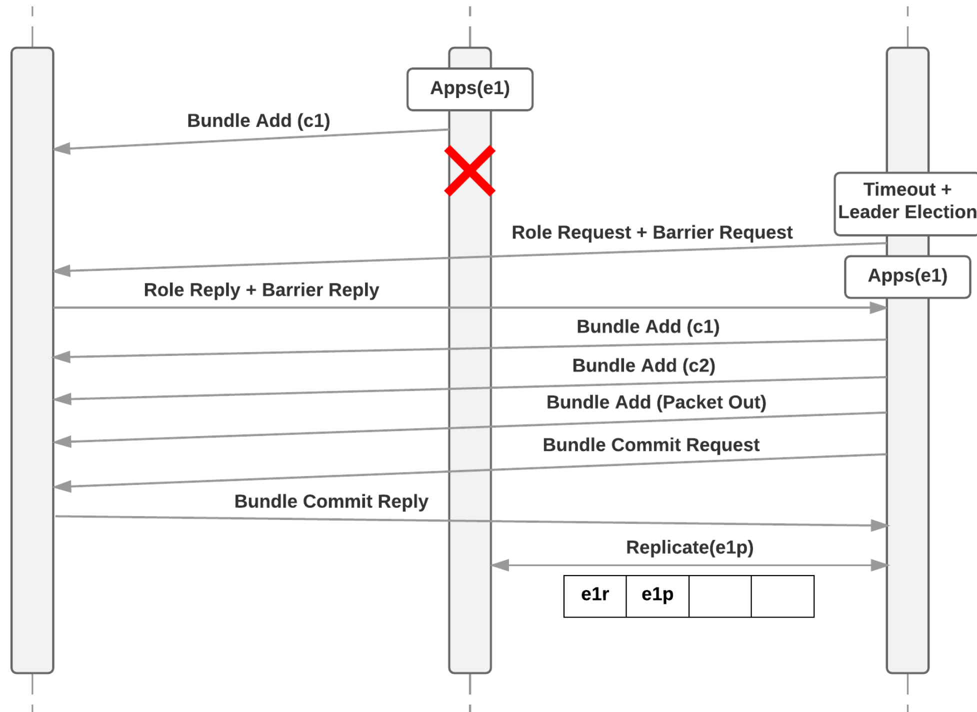


Fig. 3.6: Case of the protocol where the master fails after replicating the event. The first part of the protocol is identical to the fault-free case and is omitted from the figure. In this case, the crashed master may have already sent some commands or even the Commit Request to the switch. If the new master does not receive a Commit Reply before the Barrier Reply, it can safely repeat the commands and order a commit. If the new master received the Commit Reply (the crashed master did send the Commit Request), the new master simply continues operation without sending the old commands.

Note that to ensure that at least one controller will know that the switch completely executed the received commands it is important that the switch sends the Commit Reply message to all controllers (in our case, in the form of `OFPT_PACKET_OUT`). Imagine the other scenario where the switch only sends the Commit Reply message to the master controller. If the master controller fails after issuing the Commit Request to the switch but before receiving or replicating the reply to the other controllers, the new elected master would never know that the switch had committed the commands. As such, the inclusion

[1]This relies on the FIFO properties of the controller-switch TCP connection.

of the OFPT_PACKET_OUT message to the bundles is the key to guarantee the level of consistency required.

3.4 Consistency Properties

The protocol described in the previous sections was designed to achieve the same consistency properties as Ravana but without the need to modify the OpenFlow protocol or the switches. In this section we summarize the desired properties and the mechanisms used to achieve them. For a brief comparison, see table 3.1. Rama aims to achieve three main consistency properties:

Total event ordering: to guarantee that all controller replicas reach the same internal state, they must process a sequence of events in the same order. For this, both Rama and Ravana rely on a shared log across the controller replicas (implemented using the external coordination service) which allows the master to dictate the order of events to be followed by all replicas. Even if the master fails, the new elected master always preserves the order of events in the log and can only append new events to it.

Exactly once event processing: events cannot be lost (processed *at least once*) due to controller faults nor can they be processed repeatedly (they must be processed *at most once*). Contrary to Ravana, Rama does not need switches to buffer events neither that controllers acknowledge each received event to achieve *at-least once event processing* semantics. Instead, Rama relies on switches sending the generated events to *all (f+1)* controllers so that at least one of them will known about the event (even if the other *f* fail – considering a control plane with *f+1* replicas that tolerates *f* faults). Upon receiving these events, the master replicates them in the shared log while the slaves buffer them. If the master fails before replicating the events, the new elected master can append the buffered events to the log. If the master fails after replicating the events, the slaves will filter the buffered events so that they do not append the same events to the log. This ensures *at-most once event processing* since the new master only processes each event in the log one time. Together, sending events to all controllers and filtering buffered events ensure *exactly-once event processing*.

Exactly once command execution: for any given event received from a switch, the resulting series of commands sent by the controller are processed by the affected switches exactly *once*. Here, Ravana relies on switches acknowledging and buffering the received commands (to filter duplicates) from controllers. As this requires changes to the OpenFlow protocol and to switches, Rama relies on OpenFlow Bundles to guarantee transactional processing of commands. Additionally, the Commit Reply message, which is triggered after the bundle finishes, is sent to *all* controllers and thus acts as an acknowledgement that is independent of controller faults. If the master fails, the new master needs to know if it should resend the commands for the logged events or not. A Packet Out mes-

sage at the end of the bundle acts as a Commit Reply message to the slave controllers. This way, upon becoming the new master, the controller replica has the required information to know if the switch processed the commands inside the bundle or not, without relying on the crashed master. Furthermore, note that the new master sends a Barrier Request message (see section 3.3.1) to the switch. Receiving the corresponding Barrier Reply message guarantees that neither the switch nor the link are slow (because we received a reply) and thus there is no possibility of the Packet Out being delayed. Therefore, the using of Bundles that include a Packet Out at the end plus the use of a Barrier message ensures that commands will be processed by the switches *exactly-once*.

<table>
<tr><th>Property</th><th>Ravana</th><th>Rama</th></tr>
<tr><td>At least once events</td><td>Buffering and retransmission of switch events</td><td>Switches send events to every controller with role EQUAL</td></tr>
<tr><td>At most once events</td><td colspan="2">Event IDs and filtering in the log</td></tr>
<tr><td>Total event order</td><td colspan="2">Master appends events to a shared log</td></tr>
<tr><td>At least once commands</td><td>RPC acknowledgments from switches</td><td rowspan="2">Bundle commit is known by every controller by piggybacking PacketOut in OpenFlow Bundle</td></tr>
<tr><td>At most once commands</td><td>Command IDs and filtering at switches</td></tr>
</table>

Table 3.1: How Rama and Ravana achieve the same consistency properties using different mechanisms

It is important to note that we also consider the case where switches fail. However, this is not a special case of the protocol because it is already treated by the normal operation of the OpenFlow protocol. A switch failure will generate an event in the controller which will be delivered to the applications, for them to act accordingly (e.g., route traffic around the failed switch). A switch may fail before sending the Commit Reply to the master and the slave controllers. However, this does not mean that the transaction fails. Since this is a normal scenario in SDN, controller replicas simply mark pending events for the failed switch as processed and move on.

While we detail our reasoning as to why our protocol meets the described consistency properties, modelling the Rama protocol and giving a formal proof is left as future work and out of the scope of this dissertation.

Overall, the Rama protocol makes use of the existing mechanisms in OpenFlow to ensure that each controllers has all the required information to act accordingly to the situation and leave the system in a correct and consistent way (e.g., knowing if a bundle was committed on the switch so that it does not send repeated commands). In the next chapter we detail the our implementation of the Rama protocol.

Chapter 4 – Implementation

Our proposal, Rama, can, in principle, be implemented in any existing SDN controller (e.g., POX, Beacon, Ryu, ONOS, Opendaylight) that supports OpenFlow 1.4. This chapter gives details about how we implemented Rama.

We have built our controller on top of Floodlight [13], an open source controller for OpenFlow, implemented in Java. The fact that it is written in Java and that it has a very active community were the main factors for Floodlight to be chosen.

As coordination service, we opted for ZooKeeper [26], for its reliability, simplicity and wide use (see section 4.1).

4.1 ZooKeeper

ZooKeeper [26] is a well known coordination service that enables highly reliable distributed coordination. It exposes a set of primitives like naming, synchronization, and group services to be used by distributed applications. For Rama, ZooKeeper abstracts controllers from fault detection, leader election and event transmission and storage (for controller recovery).

High performance, high availability, strictly ordered access and reliability are some of the design goals of ZooKeeper. The reliability aspect means that ZooKeeper itself should be replicated across a set of hosts (called an ensemble). Each ZooKeeper server has an atomic broadcast component and maintains a replicated database that is an in-memory database containing the entire data tree (see figure 4.1). The atomic broadcast is the core of ZooKeeper: it is *an atomic messaging system that keeps all of the servers in sync*[1]. This agreement protocol ensures that every server processes messages in the same (total) order.

To split workload across ZooKeeper servers, clients can connect to any server and send requests to it. Servers can process and reply to read requests locally (using the local database) but write requests (that change the state of the service), are processed by the agreement protocol. In this protocol, servers forward write requests from clients to a single server (the leader). These servers (followers) receive message proposals from the leader and agree on the order of messages to be delivered proposed by the master. In

[1] https://zookeeper.apache.org/doc/r3.4.8/zookeeperInternals.html

conclusion, the agreement protocol implemented by ZooKeeper ensures that local replicas of the database never diverge in each server.

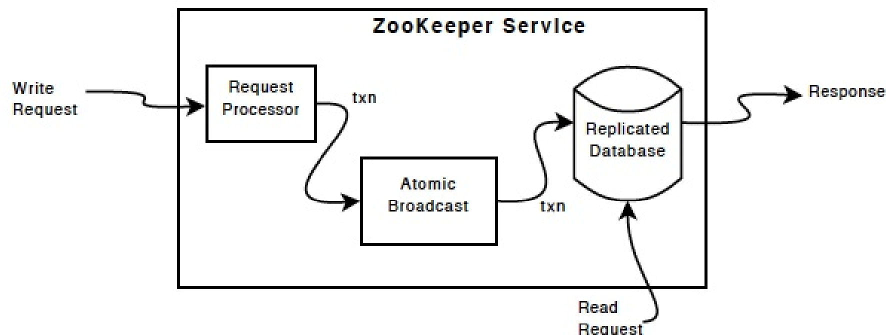


Fig. 4.1: ZooKeeper components in each server[1]

The ZooKeeper data model resembles the traditional directory tree structure of file systems, with each node in the tree (that can be seen as a directory) having data and children nodes associated with it. Two important concepts for Rama provided by ZooKeeper are *ephemeral nodes* and *watches*. The former allows clients (in our case, controllers) to create nodes that will be deleted when their session ends (allowing for fault detection), while the latter enables receipt of nodes modifications (for event replication). A simplified ZooKeeper API is shown in table 4.1.

Method	**Description**
create(path, data, mode)	Creates a node with the given path and mode.
delete(path)	Deletes the node with the given path.
exists(path, watch)	Checks if the node with the given path exists or not
getChildren(path, watch)	Returns the list of the children of the node in the given path.
getData(path, watch)	Return the data of the node with the given path
setData(path, data)	Sets the data for the node of the given path if it exists
multi(operations)	Executes multiple ZooKeeper operations or none of them

Table 4.1: Simplified ZooKeeper API

In the next sections we will discuss how ZooKeeper and its features helped us building Rama using its simple design and API.

[1]Source: https://zookeeper.apache.org/doc/r3.4.8/zookeeperOver.html

4.2 Floodlight architecture

To understand the design decisions made in Rama, it helps to understand how Floodlight is built. Floodlight is a modular controller, which means that multiple modules can be plugged into its core functionality. There are two kinds of modules in Floodlight: controller modules and application modules (see figure 4.2). Controller modules implement core network services (e.g., Link Discovery, Device Manager and Topology Manager) to be used by other modules (either core or application). Application modules, using the core modules, implement the administrator's desired network logic (e.g., hub or learning switch, firewall and other user specific use cases).

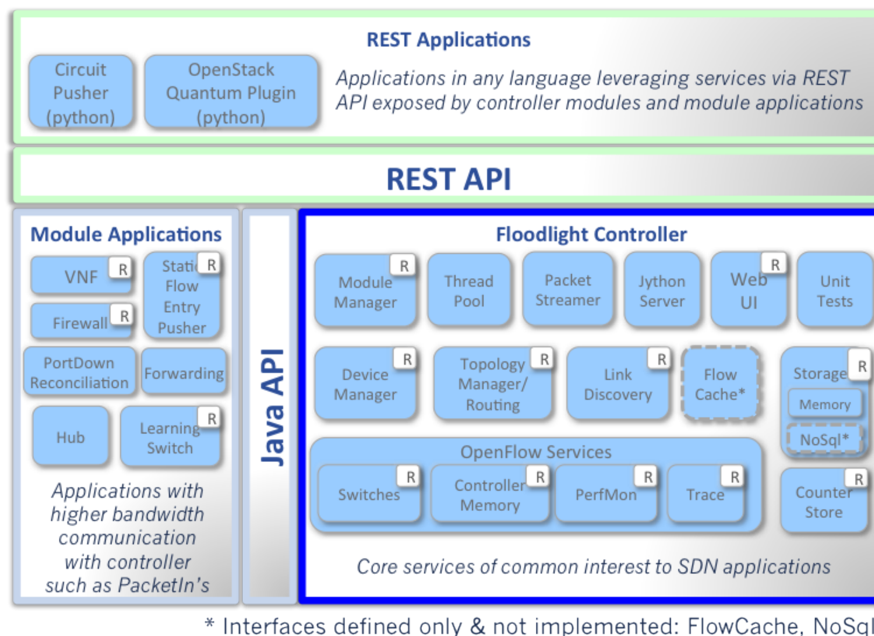


Fig. 4.2: Floodlight modules architecture[1]

Modules can register to receive different type of events (e.g., Packet-In messages, switch modifications) and act accordingly to program the switches. These events are processed in a pipeline that traverses all modules that registered to receive that type of event.

Floodlight uses Netty, an asynchronous event-driven framework, for its I/O operations with switches. A thread pool with a fixed number of threads (worker threads) collects network events, with each worker thread processing one event at a time through the pipeline. Figure 4.3 describes the life cycle of worker threads. Note that each worker thread is only available to pick up a new network event when it finishes processing the pipeline of modules.

[1]Source: https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Architecture

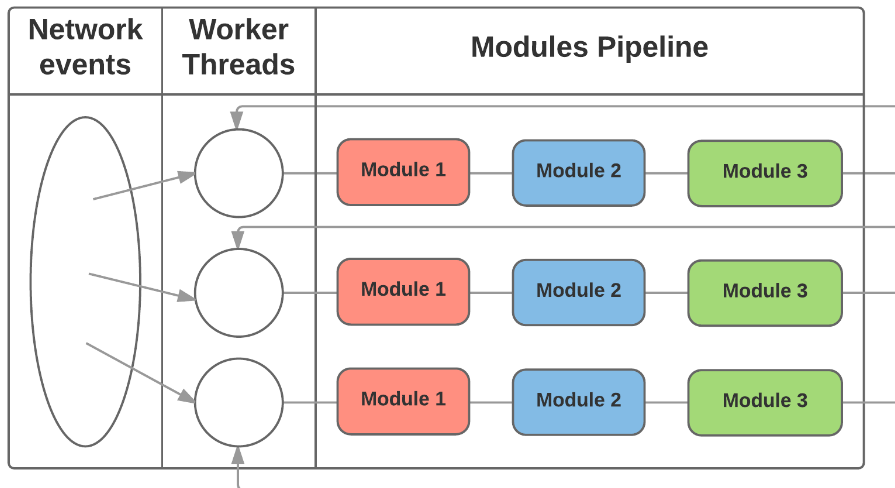


Fig. 4.3: Floodlight thread architecture

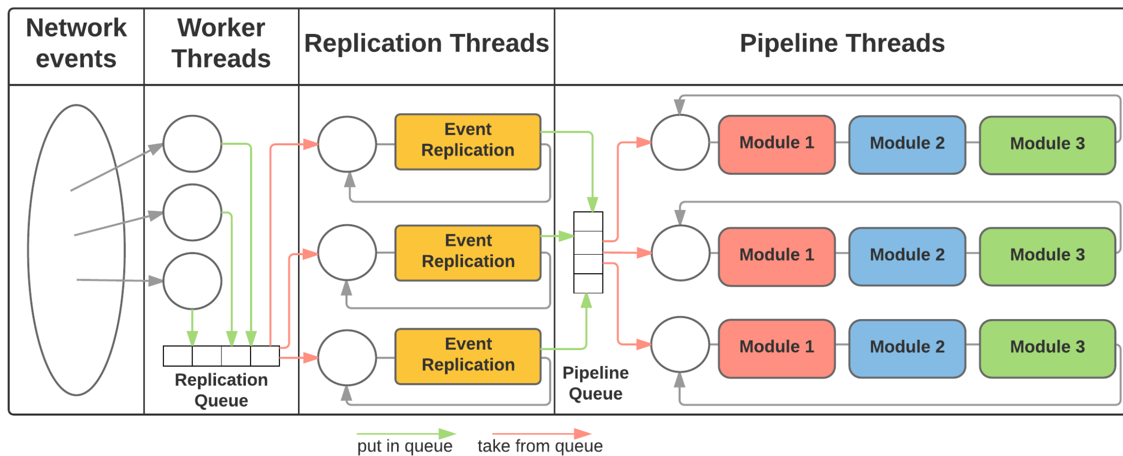


Fig. 4.4: Rama thread architecture

4.3 Rama architecture

Rama introduces two main modules into Floodlight: the *Event Replication* module (section 4.4) and the *Bundle Manager* module (section 4.5). Additionally, the Floodlight architecture was modified for performance reasons (see figure 4.4).

In the original Floodlight, the worker threads would be used to collect network events and to process the modules pipeline. If we kept this design it would be impossible to perform event batching before sending the events to ZooKeeper (this is further explained in section 4.4). Ideally, we want to free the threads that collect network events (netty worker threads) as soon as possible so that they can keep collecting more events. For this purpose, the worker threads' only job is now to put events in a queue (Replication Queue). The Replication threads will take events from this queue and execute the logic in the Event Replication module, which will send the events to ZooKeeper in batches (see section 4.4.2). When ZooKeeper replies to the batched request, events will be added to

the Pipeline Queue to be processed by the Floodlight modules.

One of the requirements for this work is to make the control plane transparent for applications (in Floodlight, to its modules). The Event Replication module is thus made completely transparent to other modules since it acts before the pipeline. The modules will continue to receive events as usual in Floodlight and process them by changing their internal structures and sending commands to switches. Returning to the proposed protocol (see section 3.3), we need to send commands to switches inside bundles, in a transparent way for modules. To achieve this, we only modified a core class of Floodlight, `OFSwitch.java` to interact with our Bundle Manager (see section 4.5). This core class contains the method `write(OFMessage m)` that sends a message to the switch using the established TCP connection and is (already) used by all modules to send commands to switches as part of the Floodlight architecture and design. Therefore, this process of sending messages inside OpenFlow Bundles is completely transparent to the existing and future Floodlight modules.

4.4 Event Replication and ZK Manager

The Event Replication module is the bridge between receiving events from the netty worker threads and putting them in the pipeline queue to be processed by the Floodlight modules. Events are only added to the pipeline queue after being stored in ZooKeeper. To separate tasks, Event Replication leverages on the ZK Manager, an auxiliary class that acts as ZooKeeper client (establishing connection, making requests and processing replies) and keeps state regarding the events (an event log and an event buffer in case of slaves) and switch leadership. We consider that an event is a pair $<$switch, message$>$ and that it is processed as such through the whole Rama architecture. Floodlight also has the notion of *context* that is passed to the modules pipeline, but we can ignore it for now. Event Replication and the ZK Manager work together to attain exactly-once event delivery and total order.

When an event arrives at the Event Replication module, we check whether the controller is in master or slave mode (for that switch). In master mode the event is replicated in ZooKeeper and added to its in-memory log. This log is a collection of `RamaEvent` objects, which apart from the switch and message, contains an unique event identifier (an incremental long number given by the current master), information related to the switches affected by the event, and of which switches already processed the commands sent. The events are replicated in ZooKeeper in batches (see section 4.4.2), so each replication thread simply adds an event to the current batch and becomes free to process a new event. Eventually the batch will be sent to ZooKeeper containing one or more events to be stored. Upon receiving the reply, the events that were stored will be added to the pipeline queue ordered according to the identifier given by the master (i.e., event *i* can only be added to

the queue after event *i-1*).

In slave mode, the event is simply buffered in memory for the case where the master controller fails. A special case is when the event received is the Packet Out that the master controller added to the bundle. In this case, the slave marks that this switch already processed all commands for this event. Slaves also keep an event log as the master, but only events that come from the master are added to it. Events from the master arrive via *watches* set in ZooKeeper nodes. Slaves set watches in the *received-events* node (see figure 4.5) and are notified when the master creates event nodes under that node. After being notified, the slave gets all the children nodes of the *received-events* node and searches for new events, getting the data stored for each one. Note that each node (e.g., e0000000) has a list of multiple events in its data. The node name represents the first id contained in the data of the node. The new events are added to the in memory log (so it is kept up-to-date with the log maintained by the master) and the events are added to the pipeline queue in the same way as in the master controller. An important detail is that event identifiers are set by the master controller, and when slaves deserialize the data obtained from nodes stored in ZooKeeper, they get the same exact `RamaEvent` objects created by the master. Therefore, the events will be queued in the same order as they were in the master controller replica.

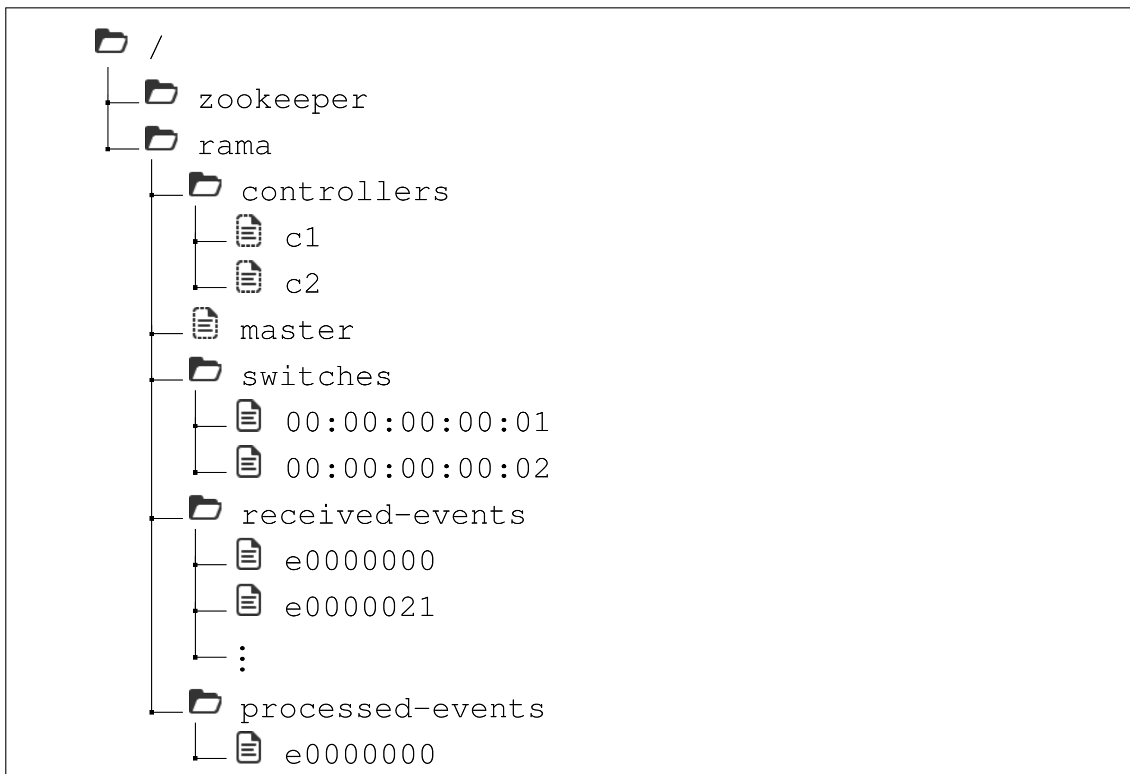


Fig. 4.5: ZooKeeper node structure. In this case, the node e0000000 under received-events contains a list with events from event 0000000 to e0000020

4.4.1 Fault Detection and Leader Election

The ZK Manager module is also responsible for detecting and reacting to controller faults. At start up, each controller will try to create an *ephemeral node* called *master* under the main *rama* folder, with the data set for its identifier within the group. Note that creating a node that already exists results in error. This means that only one controller (the master) will be able to create the node (and succeed), while the other controllers will get an error saying that the node already exists. In this case, (slave) controllers will leave a watch in the *master* node. When the master controller (the one that was successful in creating the *master* node) fails, the *ephemeral node* will be deleted by ZooKeeper and it will send a notification to every controller that has a watch in the *master* node. Upon receiving this notification, controllers will retry the procedure mentioned above and only one of them will be the new master, while the others reset the watch to be notified again in the future.

4.4.2 Event batching

Floodlight thread architecture was modified to allow event batching, which is done for performance reasons. Considering that ZooKeeper is running on a separated machine from the master controller replica, sending one event at a time to ZooKeeper would significantly degrade performance. Therefore, the ZKManager groups events before sending them to ZooKeeper in batches. Batches are sent to ZooKeeper using a special request called `multi`, which contains a list of operations to execute (e.g., create, delete, set data). For event replication, the multi request will have a list with multiple create operations as parameter. This request is sent after reaching the maximum configured amount of events (e.g., 1000) or some time after receiving the first event in the batch (e.g., 50ms). This means that each event has a maximum delay time (regarding event batching). Furthermore, to minimize the number of nodes created in ZooKeeper, each create operation will have data representing a list of events. This list size is bounded by the maximum allowable size of the data in each node, which is 1MB (1,048,576 bytes).

Let's suppose a scenario where a master controller receives 2050 events in less than 50ms, the batch size is 1000 events and we want each node in ZooKeeper to hold at maximum of 100 events. Three multi requests will be made: two consisting of a list with 10 *create node* operations (each node with 100 events) and another with only one create node (with the remaining 50 events). All this in three requests to ZooKeeper and 21 nodes saved. Without any kind of batching, we would have 2050 requests to ZooKeeper and 2050 nodes.

Summing up, we have two goals when batching events: (i) we want to send as few requests as possible to ZooKeeper and (ii) we want to create as few nodes as possible in ZooKeeper. For (i) we use the multi request which can group multiple operations in one

single request and for (ii) we group multiple events in a list and serialize it to use as the data for each create node operation (as opposed to having one *create node* operation for each event inside the multi request).

4.5 Bundle Manager

The Bundle Manager module keeps state related to all the bundles opened for each switch (as result of an event) and is responsible for adding messages, closing and committing them. We modified the write method in `OFSwitch.java` (class that is used by all modules to send commands to switches) to call the Bundle Manager that will wrap the message sent by modules in a `OFPT_BUNDLE_ADD_MESSAGE` and send it to the switch. Upon receiving this message, the switch will add the inner message to the (previously) opened bundle. Therefore, this process is transparent to all modules because they do not need to be modified to know about the presence of the Bundle Manager module. In the end of the pipeline, the Bundle Manager module is called to close and commit the bundles containing the messages added by the modules for this event. Note that one event may cause modules to send commands to multiple switches, so in this step the Bundle Manager may send `OFPBCT_COMMIT_REQUEST` to one or more switches. Before committing the bundle, the Bundle Manager also adds a `OFPT_PACKET_OUT` message to it, so that slave controllers will know if the commands for an event were committed or not in the switch (as explained in section 3.3). This message will be received by the slave controllers as a `OFPT_PACKET_IN` message with data set by the master controller. This data contains the identifiers of the event, of the switch and of the bundle.

The Bundle Manager and the ZK Manager work jointly to attain exactly-once command execution on switches.

Chapter 5 – Evaluation

In the previous chapters we explained the design and some implementation details of Rama, our consistent, fault-tolerant SDN controller. In this chapter we evaluate Rama to understand its viability, the costs associated with the mechanisms used to achieve the desired consistency properties (without modifying the OpenFlow protocol or switches), and how it compares with the alternatives (Ravana [12]).

For our tests we used 3 machines connected to the same switch via 1Gbps links as depicted in figure 5.1. Each machine has an Intel Xeon E5-2407 2.2GHz CPU and 32 GB (4x8GB) of memory. Machine 1 runs one or more Rama instances, machine 2 runs ZooKeeper 3.4.8[1] and machine 3 runs Cbench to evaluate the controller performance. This setup tries to emulate a scenario similar to a real one with ZooKeeper on a different machine for fault-tolerance purposes and Cbench on a different machine to add some network latency.

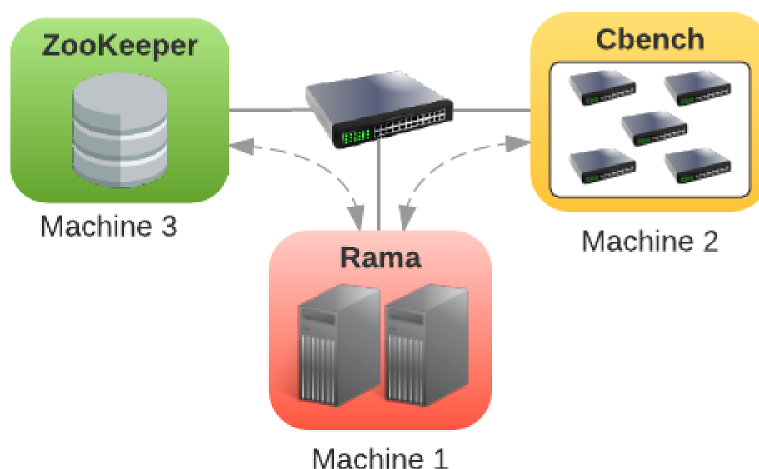


Fig. 5.1: Experiment setup

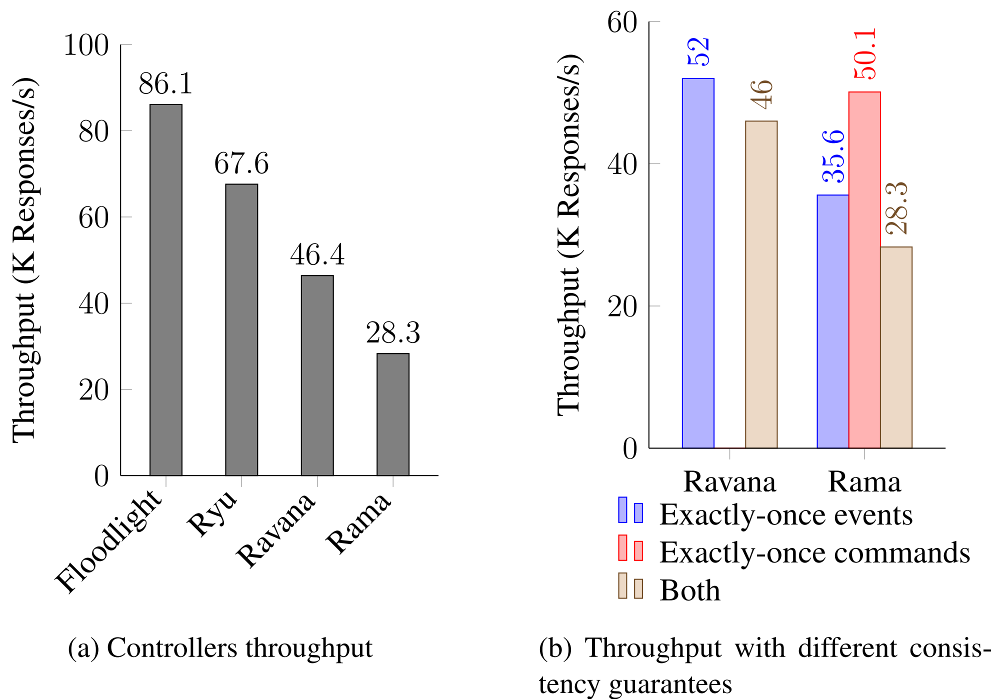
Cbench is a tool that simulates a configurable number of OpenFlow switches that connect to a SDN controller and tests its performance by sending Packet In messages and measuring reported times. Cbench can run in two distinct modes: throughput and latency. In throughput mode Cbench always keeps its network buffer full of Packet In

[1] http://zookeeper.apache.org/doc/r3.4.8/

messages for each switch-controller connection and counts replies (Flow Mod or Packet Out messages) as they arrive. In latency mode Cbench sends one Packet In message and waits for a reply before sending the next one. Note that we modified Cbench to handle OF Bundle messages

5.1 Rama Performance

We have compared the performance of Rama against Floodlight [13], Ravana [12] and Ryu [46] (the base controller for Ravana). Figure 5.2a shows the throughput for each controller (for Ravana and Ryu we use the results reported in [12], as they considered a similar setup). For Floodlight and Rama measurements we run Cbench emulating 16 switches.



(a) Controllers throughput

(b) Throughput with different consistency guarantees

Fig. 5.2: Throughput comparison

Floodlight is optimized for performance achieving around 85K responses per second, with a configuration where only some core modules and a Hub application module is running. Note that we run Floodlight with a single worker thread so it comes down to Ryu's level (which has around 67K responses per second). Rama achieves close to 30K responses per second. We can see that this throughput is not quite at Ravana's level, as our solution incurs in higher costs compared to Ravana for the consistency guarantees provided.

In figure 5.2b we show, separately, throughput results considering the different levels of consistency provided by both Rama and Ravana. The exactly-once events consistency

level () ensures that no events are lost and that controllers do not process repeated events. Additionally, controllers must agree on a total order of events to be delivered to applications. For the latter, both Rama and Ravana rely on ZooKeeper to build a shared log across controllers. In our case, the master controller batches events in multiple requests to ZooKeeper, waits for replies, and orders the events before adding them to the Pipeline Queue. In Ravana the processing is equivalent.

The Exactly-once commands semantics () ensures that commands sent by controllers are not lost and that switches do not receive duplicate commands. Ravana relies on switches to explicitly acknowledge each command and filter repeated ones. For Rama, this includes maintaining state of all opened bundles for switches, and sending additional messages to the switches. Instead of replying only with a Packet Out as in Floodlight, Rama must send messages to open the bundle, add the Packet Out to it, close the bundle and commit it. To evaluate this case, we had to modify Cbench. In our modified version of Cbench, a switch only counts a reply when it receives a Commit Request message from the controller (not when it receives a Bundle Add message with the Packet Out). This allows a faithful emulation of the performance of Rama in a real system – indeed, in Rama a packet will only be forwarded after committing the bundle on the switch to guarantee consistent process.

Note that neither Rama nor Ravana wait for ZooKeeper to persistently store requests on disk (they both use ZooKeeper in-memory). In our case, the multi request is sent asynchronously (i.e., threads are freed to continue operation) and a callback function is registered. This function will be activated when ZooKeeper replies to our multi request and enqueues the logged events (in order) in the Pipeline Queue to be processed by the modules.

As show in Figure 5.2b, some guarantees are costlier to ensure than others. For instance, the cost of providing Exactly-once events semantics is higher than Exactly-once commands semantics. Note that we do not include the results from Exactly-once commands in Ravana as these are not available in [12].

Figure 5.3 shows how maintaining multiple switch connections affects Rama throughput. As switches send events at the highest possible rate, the throughput of the system saturates with around 16 switches. Importantly, the throughput does not decrease with a higher number of switches, which is similar to Ravana.

Rama batches events to reduce the communication overhead of contacting ZooKeeper. In practice, events are sent to ZooKeeper after reaching a configurable number of events in the batch (batching size) or after a configurable timeout (batching time).

To evaluate batching we conducted a series of tests with different configurations to understand how the batching size and time affects Rama performance (figure 5.4). Note that throughput is only affected by batching size and never by batching time. This happens because in throughput mode Cbench sends events at the highest rate possible and therefore

the batching time is never reached.

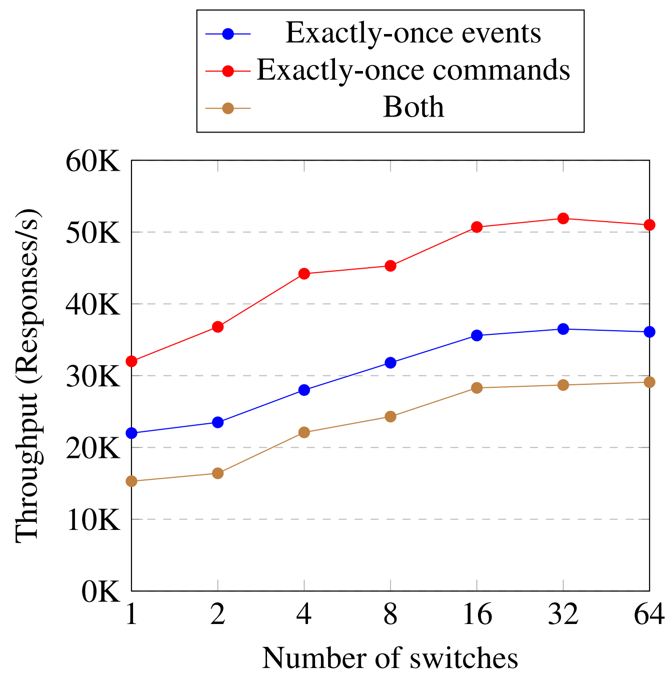


Fig. 5.3: Rama throughput with different number of switches

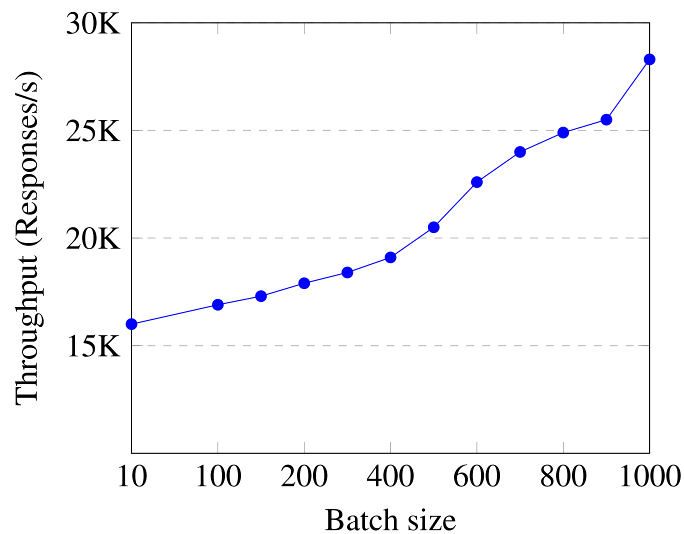


Fig. 5.4: Variation of Rama throughput with batch size

5.2 Latency

For latency measurements, we want to know the average time that one request takes to be processed by Rama.

If we consider that the controller will be batching events until the batch is full or until a timeout is reached, the maximum (worse) possible latency is around the batching time (e.g., 100ms). This corresponds to the case where switches do not send enough events to fill the batch and so Rama waits until the timeout is reached.

On the other hand, if the number of events received fill the batch, Rama will process the events as soon as possible without waiting, resulting in a better latency. To test this, we use the results from running cbench in throughput mode (to trigger the batching limit) with 1 switch (to test the worst case and give a more realistic value) from figure 5.3, which is 15.3K responses per second. To get the seconds it takes for one request to be processed (the latency), we need to calculate the inverse (1/15300) which gives around 65 microseconds.

5.3 Failover Time

To measure the time for Rama to react to failures we use mininet [47], our modified version of OpenvSwitch [14] and iperf [48].

Mininet creates a virtual network with multiple switches and hosts that runs in one machine. This allows us to run command line programs in one host to communicate with other hosts (e.g., ping) using the virtual switches. The virtual switch used by mininet is a version of OpenvSwitch that we modified to handle bundle related messages and only forward packets after a Commit Request message. Iperf is a network bandwidth measurement tool that runs in both client and server mode. The client generates IP or UDP packets and sends them to the server (usually at a constant rate) and the server reports the results in terms of bandwidth, packet loss, and other parameters.

We setup a simple topology in Mininet with one switch and two hosts, one to act as iperf server and another as client. We start the client and sever in UDP mode, where the client generates 1 Mbit/sec during 10 seconds. The switch connects to two Rama instances and sends all events to both. Each Rama instance is connected to ZooKeeper server running on another machine (as before) with a negotiated session timeout of 500ms (the minimun we could set in ZooKeeper). To make sure that no rules are installed on the switch – so that events are sent to controllers each time a packet arrives – we run Rama with a module that only forwards packets (using Packet Out messages) without modifying the switch's tables.

Figure 5.5 shows the reported bandwith from the iperf server and indicates the time taken by Rama to react to failures.

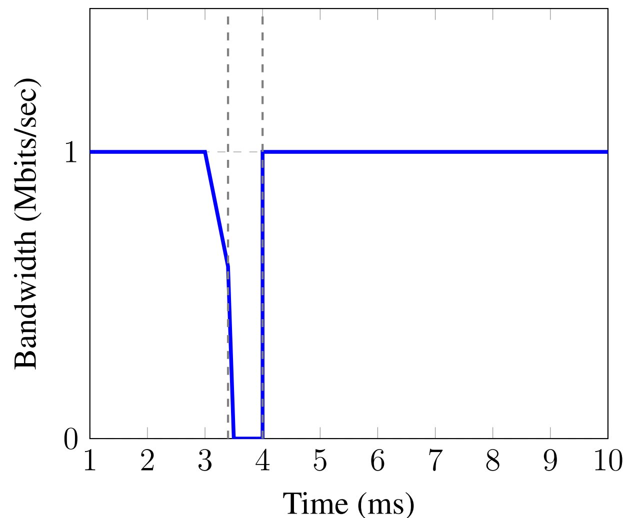


Fig. 5.5: Rama failover time

Namely, the slave replica takes around 550ms to react to faults. This includes the time for:

(a) ZooKeeper to detect the failure and notify the slave replicas (500ms).

(b) Electing a new leader for the switches.

(c) The new leader to transition to master (finish processing logged events from the old master to reach the same internal state).

(d) Append buffered events to the log and start delivering unprocessed events in the log to applications so they start sending commands to the switches.

The major factor is the time ZooKeeper needs to detect the failure of the master controller. As it may be possible to reduce the session timeout, we were not able to achieve it without incurring into problems (either ZooKeeper did not accept client requests when the timeout was too small, or clients were constantly losing their session even when running). For comparison, Ravana reports a failover time of 75ms with 40ms to detect the failure (as opposed to our 500ms to detect the failure).

Summing up, Rama comes close, but does not achieve the performance of Ravana (as expected since the design phase of the dissertation). This is due to the fact that Rama incurs into higher costs (requires more messages to be sent over the network) in order to achieve the same properties as Ravana. Despite the small loss in performance, the value proposition of Rama of guaranteeing consistent command and event processing without requiring modifications to switches or to the openflow protocol makes it an effective enabler for immediate adoption of fault-tolerant SDN solutions.

Chapter 6 – Conclusion & Future Work

In a fault-tolerant Software-Defined Network, maintaining a consistent controller state is not enough to achieve a correct system. Unlike other distributed systems, in SDN it is necessary to consistently handle switch state to avoid loss or repetition of commands under controller failures.

To address these challenges imposed by SDN we propose Rama, a consistent and fault-tolerant SDN controller that handles the entire event processing cycle (i.e., event delivery by switches, event processing by controllers, command delivery by controllers and execution by switches) exactly-once. Rama is built on top of Floodlight – a Java SDN controller – and relies on the widely used ZooKeeper as its coordination service.

Rama differs from the existing alternative, Ravana, by not needing to modify the OpenFlow protocol nor existing OF switches. While Ravana relies on events and commands buffers on switches, and requires new messages to be added to the OpenFlow protocol, Rama leverages on switches sending events to all controllers and on existing OF Bundles to achieve the same consistency guarantees. We believe this is fundamental to ease the adoption of Rama in the near future.

During the evaluation of Rama, we confirmed what was expected: Rama has a slightly worse performance than Ravana because it incurs into higher costs to provide the same consistency guarantees. We believe that the difference in performance is justifiable and worth the fact that Rama does not require modifications to the switches nor to the OF protocol.

Over the course of this dissertation we had the chance to work and contribute in three projects: Floodlight, OpenvSwitch and Cbench. Floodlight and OpenvSwitch are two on-going open source projects with a vibrant community so it was a very interesting and different experience which involved: learning about the system architecture, the existing code, and how to contribute – communicating and discussing plans for contributing, implementing the functionality, testing and submitting patches to the working group.

Looking back at the development of Rama, something that delayed the whole process was not thinking for performance in the first place. At first, we tried to develop Rama for a simple scenario with few switches and not worrying about stressing the system throughput. Later on, while evaluating Rama, we found that its performance was poor and we had to make changes to the overall system architecture as well as optimizations to

our modules.

As for future work, we plan to model Rama and present a formal proof regarding the consistency guarantees Rama provides. Additionally, Rama can still be improved for performance, scalability, robustness and security For instance, Rama can be improved to act as a distributed controller where one controller replica acts as master for a subset of switches and another replicas act as master for the other subsets of switches. Although this would require more synchronization (we would still need to maintain total order of events across all masters), this approach can offload master replicas since under no faults each controller only processes events received by its subset of switches.

Glossary

API Application Programming Interface.

BGP Border Gateway Protocol.

FIFO First in, first out.

IP Internet Protocol.

NIB Network Information Base.

OF OpenFlow.

OSPF Open Shortest Path First.

RSVP Resource Reservation Protocol.

SDN Software-Defined Networking.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

VLAN Virtual Local Area Network.

ZK ZooKeeper.

References

[1] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. "Onix: A Distributed Control Platform for Large-scale Production Networks." In: *OSDI*. Vol. 10. 2010, pp. 1–6.

[2] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[3] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. "B4: Experience with a globally-deployed software defined WAN". In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. ACM. 2013, pp. 3–14.

[4] VMware Inc. *VMware NSX Virtualization Platform*. 2013. URL: `https://www.vmware.com/products/nsx/`.

[5] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. "Network virtualization in multi-tenant datacenters". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 203–216.

[6] Sakir Sezer, Sandra Scott-Hayward, Pushpinder-Kaur Chouhan, Barbara Fraser, David Lake, Jim Finnegan, Niel Viljoen, Mary Miller, and Neeraj Rao. "Are we ready for SDN? Implementation challenges for software-defined networks". In: *Communications Magazine, IEEE* 51.7 (2013), pp. 36–43.

[7] Teemu Koponen and Martin Casado. *What Might an SDN Controller API Look Like? (and should we standardize it?)* 2011. URL: `http://networkheresy.com/2011/08/09/what-might-an-sdn-controller-api-look-like-and-should-we-standardize-it/`.

[8] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. "NOX: towards an operating system for networks". In: *ACM SIGCOMM Computer Communication Review* 38.3 (2008), pp. 105–110.

[9] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. "On scalability of software-defined networking". In: *Communications Magazine, IEEE* 51.2 (2013), pp. 136–141.

[10] Martin Casado, Nate Foster, and Arjun Guha. "Abstractions for software-defined networks". In: *Communications of the ACM* 57.10 (2014), pp. 86–95.

[11] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. "Logically centralized?: state distribution trade-offs in software defined networks". In: *Proceedings of the first workshop on Hot topics in software defined networks*. ACM. 2012, pp. 1–6.

[12] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. "Ravana: Controller fault-tolerance in software-defined networking". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM. 2015, p. 4.

[13] *Floodlight SDN Controller*. 2015. URL: `http://www.projectfloodlight.org/floodlight/`.

[14] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. "The design and implementation of open vswitch". In: *12th USENIX symposium on networked systems design and implementation (NSDI 15)*. 2015, pp. 117–130.

[15] Rob Sherwood and Kok-Kiong Yap. *Cbench: an OpenFlow Controller Benchmarker*. URL: `http://www.openflow.org/wk/index.php/Oflops`.

[16] ONF. *Open Networking Foundation*. 2015. URL: `https://www.opennetworking.org`.

[17] Arsalan Tavakoli, Martin Casado, Teemu Koponen, and Scott Shenker. "Applying NOX to the Datacenter." In: *HotNets*. 2009.

[18] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. "The nature of data center traffic: measurements & analysis". In: *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM. 2009, pp. 202–208.

[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*. ACM. 2010, pp. 267–280.

[20] David Erickson. "The beacon openflow controller". In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, pp. 13–18.

[21] Zheng Cai, Alan L Cox, and TS Eugene Ng. "Maestro: A system for scalable openflow control". In: *Structure* (2010).

[22] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. "On controller performance in software-defined networks". In: *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*. Vol. 54. 2012.

[23] Stefan Schmid and Jukka Suomela. "Exploiting locality in distributed sdn control". In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, pp. 121–126.

[24] 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*. USENIX Association. 2010, pp. 3–3.

[25] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. "ONOS: towards an open, distributed SDN OS". In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, pp. 1–6.

[26] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX Annual Technical Conference*. Vol. 8. 2010, p. 9.

[27] Pingping Lin, Jun Bi, and Hongyu Hu. "Asic: an architecture for scalable intra-domain control in openflow". In: *Proceedings of the 7th International Conference on Future Internet Technologies*. ACM. 2012, pp. 21–26.

[28] Kévin Phemius, Mathieu Bouet, and Jérémie Leguay. "Disco: Distributed multi-domain sdn controllers". In: *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE. 2014, pp. 1–4.

[29] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. "Scalable flow-based networking with DIFANE". In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 351–362.

[30] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. "DevoFlow: Scaling flow management for high-performance networks". In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 254–265.

[31] 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*. ACM. 2012, pp. 19–24.

[32] Soheil Hassas Yeganeh and Yashar Ganjali. "Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking". In: *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM. 2014, p. 13.

[33] Armando Fox, Eric Brewer, et al. "Harvest, yield, and scalable tolerant systems". In: *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*. IEEE. 1999, pp. 174–178.

[34] Eric Brewer. *CAP Twelve Years Later: How the "Rules" Have Changed*. 2012. URL: `http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed`.

[35] Peter Perešíni, Maciej Kuzniar, Nedeljko Vasić, Marco Canini, and Dejan Kostiū. "OF. CPP: Consistent packet processing for OpenFlow". In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, pp. 97–102.

[36] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. "Consistent updates for software-defined networks: Change you can believe in!" In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM. 2011, p. 7.

[37] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. "Abstractions for network update". In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM. 2012, pp. 323–334.

[38] Marco Canini, Petr Kuznetsov, Dan Levin, Stefan Schmid, et al. "The case for reliable software transactional networking". In: *arXiv preprint arXiv:1305.7429* (2013).

[39] Hyojoon Kim, Jose Ronaldo Santos, Y Turner, M Schlansker, Jean Tourrilhes, and Nick Feamster. "Coronet: Fault tolerance for software defined networks". In: *Network Protocols (ICNP), 2012 20th IEEE International Conference on*. IEEE. 2012, pp. 1–2.

[40] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. "Fattire: Declarative fault tolerance for software-defined networks". In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, pp. 109–114.

[41] Francisco Javier Ros and Pedro Miguel Ruiz. "Five nines of southbound reliability in software-defined networks". In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, pp. 31–36.

[42] M Faizul Bari, Arup Raton Roy, Shubhajit Roy Chowdhury, Qi Zhang, Mohamed Faten Zhani, Rizwan Ahmed, and Raouf Boutaba. "Dynamic controller provisioning in software defined networks". In: *Network and Service Management (CNSM), 2013 9th International Conference on*. IEEE. 2013, pp. 18–25.

[43] Fábio Botelho, Alysson Bessani, Fernando Ramos, and Paulo Ferreira. "SMaRt-Light: A Practical Fault-Tolerant SDN Controller". In: *arXiv preprint arXiv:1407.6062* (2014).

[44] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. "ElastiCon: an elastic distributed sdn controller". In: *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM. 2014, pp. 17–28.

[45] Anand Krishnamurthy, Shoban P Chandrabose, and Aaron Gember-Jacobson. "Pratyaastha: an efficient elastic distributed sdn control plane". In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, pp. 133–138.

[46] *Ryu SDN Framework*. 2016. URL: `https://osrg.github.io/ryu/`.

[47] *Mininet: An Instant Virtual Network on your Laptop*. 2016. URL: `http://mininet.org/`.

[48] *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. 2016. URL: `https://iperf.fr/`.

[49] Brandon Heller, Rob Sherwood, and Nick McKeown. "The controller placement problem". In: *Proceedings of the first workshop on Hot topics in software defined networks*. ACM. 2012, pp. 7–12.

[50] Brian M Oki and Barbara H Liskov. "Viewstamped replication: A new primary copy method to support highly-available distributed systems". In: *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. ACM. 1988, pp. 8–17.

[51] An Wang, Yang Guo, Fang Hao, TV Lakshman, and Songqing Chen. “Scotch: Elastically scaling up SDN control-plane using vswitch based overlay”. In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM. 2014, pp. 403–414.