Universidade Federal do Amazonas
Instituto de Computação
Programa de Pós-Graduação em Informática

Paulo César da Rocha Fonseca

Resiliência em Redes Definidas por Software através de Replicação

Manaus
2013

Paulo César da Rocha Fonseca

Resiliência em Redes Definidas por Software através de Replicação

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas, como requisito parcial para a obtenção do título de Mestre em Informática.

Orientador: Edjard Souza Mota

Manaus
2013

Fonseca, P. C. R.
Resiliência em Redes Definidas por Software através de Replicação
43 páginas
Dissertação (Mestrado) - Instituto de Computação da Universidade Federal do Amazonas.

1. Redes Definidas por Software
2. Tolerância a Falhas
3. Replicação

I. Universidade Federal do Amazonas. Instituto de Computação.

Comissão Julgadora:

à Dona Francisca, minha vó branca

Transformai-vos pela renovação da vossa mente.

Romanos 12:2

Agradecimentos

Agradeço, antes de tudo, a Deus. Não é possível definir em palavras o papel dele na sequência de acontecimentos que me trouxe até aqui, não posso fazer nada além de agradecer, dia e noite.

Ao meu orientador, Edjard Mota, por ter me dado a oportunidade que definiria minha vida acadêmica e por ter me acompanhado por todo este percurso. Orientador admirável, sempre presente e nunca alheio aos nossos problemas, sempre disposto a ajudar, muitas vezes fazendo sacrifícios que extrapolavam as responsabilidades de um orientador para nos ajudar, sempre empurrando os nossos limites e nos mostrando na prática que podíamos mais do que nós imaginávamos. As lições que aprendi com ele, carregarei pelo resto da vida e só tenho a agradecer.

Aos meus pais, Paulo José e Sônia Cláudia, por serem pessoas excepcionais, servindo de exemplo profissional e acadêmico durante toda a minha vida e sempre me lembrando, involuntariamente, quão imensamente agradecido eu deveria ser por ter tido a sorte de ser filho de pessoas tão incríveis. Agradeço a toda minha família, por ter sido este porto seguro, sem o qual não conseguiria viver.

À Carol, minha namorada, por renovar as minhas forças diariamente, por me fazer sobreviver e viver. Pela paciência necessária para suportar a distância nas várias de suas formas devido o desenvolvimento deste trabalho. Por ter me ensinado a dar valor às pequenas coisas da vida. Pelas palavras de incentivo nos momentos difíceis e pela multiplicação da alegria nos momentos de sucesso.

Ao LabCIA, em especial ao Alexandre Passito, pela orientação imprescindível por toda a pesquisa, tomando decisões vitais para os sucessos que conquistamos, Ricardo Bennesby, pela amizade que me ajudou em todos os momentos possíveis, Rodrigo Braga, por servir de exemplo de responsabilidade e dedicação. E a todos os membros atuais, em especial ao Yan Brandão, que o seu trabalho exceda o meu em todos os sentidos.

Aos meus amigos, em especial ao Leandro Machado, Bruno Mandell, Fidel Graça, Hugo Cunha, Emory Raphael, Jonathan Byron, Flávio Montenegro, Bruno Dias, Petrônio Mello, Luan Moura, Jonas Caio, Débora Chagas e Ana Paula por fornecer a zuera necessária para completar esta caminhada.

À Monica Nogueira, pesquisador da Universidade da Carolina do Norte/EUA, pela revisão em todos os nossos artigos científicos e desta dissertação, pelo tempo dispendido, pelas observações interessantes e sugestões valiosas.

Aos professores do IComp, por terem tido um papel essencial na minha formação acadêmica e profissional.

Aos professores que formaram a banca da minha qualificação, prof. Dr. Edjair Mota, prof. Dr. Leandro Galvão e prof. Dr. César Melo, pelas observações importantes que orientaram este trabalho até a presente forma.

Aos professores que formaram a banca da minha defesa de dissertação, prof. Dr. Dorgival Guedes (UFMG), prof. Dr. Alexandre Passito e prof. Dr. Leandro Galvão, pelo tempo dispendido e pelos conselhos valiosos.

À CAPES, por ter fornecido o auxílio financeiro que viabilizou esta pesquisa.

Resumo

Redes definidas por software (Software Defined Networking ou SDN) é um novo paradigma para o desenvolvimento de aplicações inovadoras de gestão de redes e uma nova forma de olhar para a resolução de muitos problemas que existem em toda a Internet hoje. A arquitetura mais popular para a implantação deste paradigma é o gerenciamento de rede centralizado, uma vez que o seu uso permite simplificar a tarefa complexa e difícil de controlar os serviços de uma rede. Um dos problemas levantados pela abordagem de controle centralizado e amplamente discutido na literatura é a questão de falha em um único ponto da rede, que pode comprometer negativamente o funcionamento de toda ela. Um método comprovado para alcançar um maior nível de tolerância a falhas é a utilização da técnica de replicação. O objetivo deste trabalho é dividido em três partes: (1) comparar diferentes técnicas de replicação, (2) verificar como cada uma desempenha a tarefa de proporcionar resiliência a uma SDN, e (3) investigar qual técnica é a mais adequada para diferentes cenários. Técnicas de replicação são principalmente classificadas em dois tipos: de replicação passiva e ativa. Este trabalho é um dos primeiros a tratar este problema. Para fins de prova-de-conceito, os mecanismos de replicação implementados funcionam para redes com um switch habilitado para o protocolo SDN. Nossos resultados mostram que a replicação é uma forma adequada para aumentar a resiliência em uma SDN e construir estes serviços para redes utilizando SDN é muito menos complexo e simples.

Palavras-chave: Redes Definidas por Software, Tolerância a falhas, Replicação

Abstract

Software-Defined Network (SDN) is a new paradigm that allows the development of innovative network management applications and provides a new way to look for the resolution of problems which exist throughout the Internet today. In order to simplify the task of managing the network most of SDN architectures uses a centralized network management approach. However, such approach raises, among other problems, the issue of a single point of failure, that can compromise the proper functioning of the network. A proven method to achieve a higher level of network resilience is to use a replication technique. The aim of this work is to investigate: (1) how different replication techniques relate to each other, (2) how each one performs on the task of providing resilience to a SDN, and (3) which technique is the most suitable for different scenarios. Replication techniques are mainly classified in two types: passive and active replication.This work is one of the first to address this issue. For the purpose of proof-of-concept, the replication mechanisms implemented work for networks with one switch enabled for SDN protocol. Our results show that replication is a suitable way to increase resilience in a SDN and to build these services for networks using SDN is straightforward and much less complex.

Keywords: Software Defined Networking, Fault Tolerance, Replication

Lista de Figuras

Sumário

Capítulo 1

Introdução

1.1 Motivação

A crescente complexidade e dificuldade de gerenciar uma rede de computadores faz com que cresça a demanda por mudanças na arquitetura das redes para que esses problemas sejam sanados. Porém, mudanças radicais na arquitetura de uma rede são pouco viáveis, uma vez que serviços como roteamento e autenticação possuem implementações que não possuem código aberto e que estão sujeitas a variações de acordo com características específicas de hardware de cada componente da rede, dependendo do modelo e fabricante.

O novo paradigma proporcionado pelas Redes Definidas em Software (Software-Defined Networks, ou SDN) [1] visa solucionar este problema. A Figura 1.1 representa uma arquitetura SDN. Nela, a rede é organizada em três aspectos: plano de controle, plano de dados e um protocolo de comunicação. O plano de controle é responsável pelo gerenciamento dos recursos da rede e composto por um sistema operacional de redes (ou controlador) e suas aplicações, o primeiro provê uma interface programática centralizada e uniforme que oferece a possibilidade de observação e controle dos dispositivos da rede em alto nível, enquanto as aplicações utilizam esta interface e a visão holística da rede

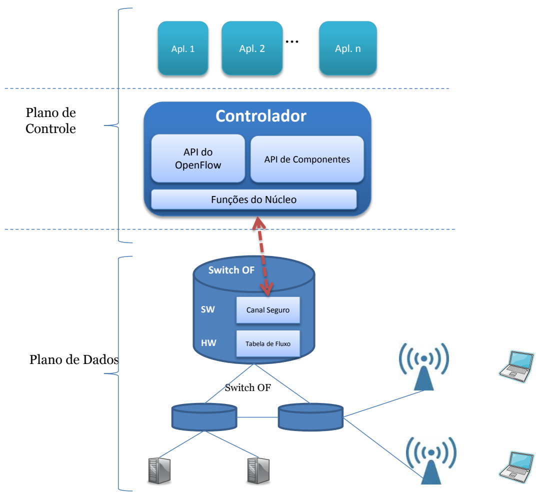

Figura 1.1: Arquitetura SDN

fornecida pelo sistema operacional de redes para gerenciar os recursos e serviços, como roteamento, autenticação e descoberta de *links*. O plano de dados é responsável pelo repasse das informações de acordo com as políticas definidas pelo administrador no sistema operacional de redes. Esta operação é feita através de regras que são instaladas no *switch* e associam ações a cada fluxo de acordo com as políticas do controlador. O protocolo *OpenFlow* [8] define regras e os formatos das mensagens usadas para a comunicação entre os dispositivos responsáveis pelo encaminhamento de tráfego na rede e o sistema operacional de redes. Esta comunicação é realizada através de um canal seguro e exclusivo do *switch* com o controlador.

Esta abordagem define um novo paradigma que provê a construção de aplicações de gestão de redes e uma nova forma de olhar para a resolução de muitos problemas que existem em toda a Internet hoje, como roteamento [2], segurança [3] e mobilidade [5] .

A versão mais completa de um sistema operacional de redes é o NOX [7]. O NOX oferece um núcleo que provê a interface programática para uma camada onde as aplicações de gerência são executadas. Esse núcleo, dentre outras coisas, utiliza o protocolo de acesso *OpenFlow* [1] para obtenção das informações dos switches e roteadores, bem como a implantação das ações solicitadas pelas aplicações de gerência.

Redes com NOX possuem a propriedade de centralização, ou seja, um único controlador na rede é responsável pelo registro, autenticação e controle de roteadores, switches e usuários. O gerenciamento de rede centralizado em SDN simplifica a tarefa de gerir os serviços e reduz o nível de dificuldade para realiza-los, no entanto, levanta, entre outras questões, o problema da vulnerabilidade da rede. Isto ocorre pois, no caso de uma falha no sistema operacional da rede todo o funcionamento da rede pode ser comprometido, uma vez que todas as aplicações e serviços dependem dele. O protocolo *OpenFlow* oferece a possibilidade de configurar um ou mais controladores de backup que pode assumir o controle da rede em caso de falha, mas *OpenFlow* não provê qualquer mecanismo de coordenação entre diferentes controladores. Assim, os componentes vão perder todas as informações de configuração previamente recolhida a partir da rede. Assim, qualquer protocolo ou serviço executado em um sistema operacional de rede deve ser protegido por um mecanismo de prestação de resiliência a falhas, capazes de realizar a coordenação entre os controladores, que irá transformar a rede para um estado seguro com sobrecarga mínima a hosts e switches.

OpenFlow é um protocolo que possui objetivo de possibilitar que protocolos experimentais sejam testatdos em redes reais sem interferir com o tráfego de produção, ele é uma característica que pode ser adicionada a switches ethernet que disponham de tal funcionalidade e trabalha à nível de tabela de fluxo, removendo e adicionando novas entradas nela. Os registros contém o padrão do cabeçalho do pacote pertencente ao fluxo, contadores (para fins estatísticos) e a ação associada ao fluxo. A tabela de fluxo é administrada por um controlador remoto que decide qual ação associar a cada novo fluxo e está conectado ao switch através de um canal seguro (criptografado por SSL). Atualmente um dos principais objetivos do grupo responsável pelo *OpenFlow* é aumentar a quantidade de empresas que comercializam switches *OpenFlow* e universidades que o implementam em seu campus.

Paralelo ao desenvolvimento do *OpenFlow* ocorre o do NOX [7], um sistema operacional para redes centralizado baseado no *OpenFlow*. O NOX age como um controlador do *OpenFlow*, adicionando e removendo registros de fluxo da tabela de fluxo, para exercer controle sobre a rede e seus recursos. Ele provê um plataforma simplificada para escrever componentes que irão gerenciar os recursos da rede. O NOX é dividido em dois aspectos: Os componentes externos de alto-nível que lidam com a API e são feitos majoritariamente em Python, também podem ser escritos em C++ ou ambos, e o núcleo do sistema operacional que lida com tarefas de baixo nível e a sua maior parte é escrita em C++, para conservar a desempenho, com alguns trechos de código em Python.

Gude [7] propõe que para aumentar a confiabilidade da rede seja mantida uma conexão com um controlador secundário que irá tomar controle da rede caso o primário venha a falhar. A literatura apresenta várias abordagens para o problema da replicação e de manter as réplicas consistentes. As duas principais abordagens de replicação são a replicação ativa e replicação passiva.

A replicação passiva (ou replicação primary-backup) [9]. Neste método toda vez que o servidor primário recebe uma requisição de um dos clientes ele completa a operação solicitada, muda o seu estado e envia uma mensagem de atualização de estado ao servidor secundário (ou, caso haja mais de um secundário, aos servidores). O servidor secundário, então, atualiza o seu estado e envia uma mensagem de confirmação para o primário, este, por sua vez, envia uma confirmação ao cliente que a operação foi completada. Paralelamente a isso o secundário envia constantemente mensagens de controle para saber se o primário continua ativo, caso o primário entre em estado de falha e não envie a confirmação (mensagem ACK), o secundário assume o controle da rede. Esta arquitetura é demonstrada pela Figura 1.2. Esta técnica deve seguir as seguintes propriedades:

P-1 Há um predicado local $Primary_s$ associado com o estado de cada servidor s. A

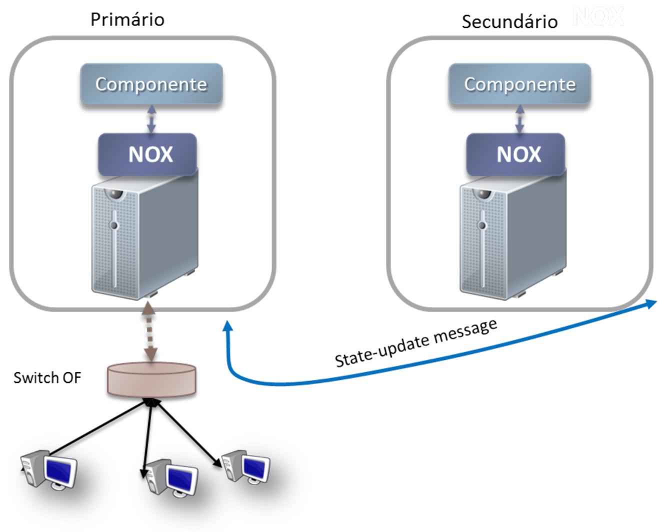

Figura 1.2: Replicação passiva

qualquer momento, existe no máximo um servidor s que satisfaz $Primary_s$.

P-2 Cada cliente i mantém um predicado de destino $dest_i$. Se o cliente precisa de uma solicitação, ele envia uma mensagem para $dest_i$.

P-3 Se o servidor s recebe uma solicitação e não é primário ($Primary_s$ não é satisfeito), ele não enfileirafa a solicitação.

P-4 Existem k e δ valores tais que o servidor se comporta como um único servidor (k, δ) - *bofo (bounded outages, finitely often)*.

A propriedade P-4 garante que o comportamento da rede utilizando primário-backup é indistinguível de o comportamento de uma rede com um servidor-*bofo* em períodos de falha, onde estes períodos podem ser agrupados em k intervalos de tempo, com cada intervalo de duração, no máximo, δ. Esta propriedade garante que qualquer implementação do protocolo precisa limitar o tempo em que o serviço está em falha.

A replicação ativa (também conhecida como abordagem State-Machine) [16] é outra replicação largamente conhecida e usada. Nesta abordagem, o cliente conecta-se com todos os servidores simultaneamente, os servidores processam os pedidos, a resposta é enviada para o cliente. A implementação deste método deve seguir estas propriedades:

A-1 Para evitar a manipulação respostas duplicadas de servidores, o cliente pode: a) processar apenas a primeira resposta ao pedido do cliente, ou b) deve haver coordenação entre os servidores a fim de decidir qual deles irá responder a requisição do cliente.

A-2 Todos os servidores devem ter um comportamento determinístico, portanto, se um servidor está em estado de a e recebe um pedido que deve especificamente ir ao estado b

A-3 O cliente deve ter uma primitiva de comunicação que permite o envio de todos os pedidos para os servidores na mesma ordem.

A propriedade A-1 deve ser mantida para evitar que o cliente inadvertidamente altere o seu estado devido a pedidos duplicados. As propriedades A-2 e A-3 garantem que todos os servidores estão sincronizados no mesmo estado, trabalhando como máquinas de estado. A Figura 3 representa a arquitetura de uma solução de replicação ativa.

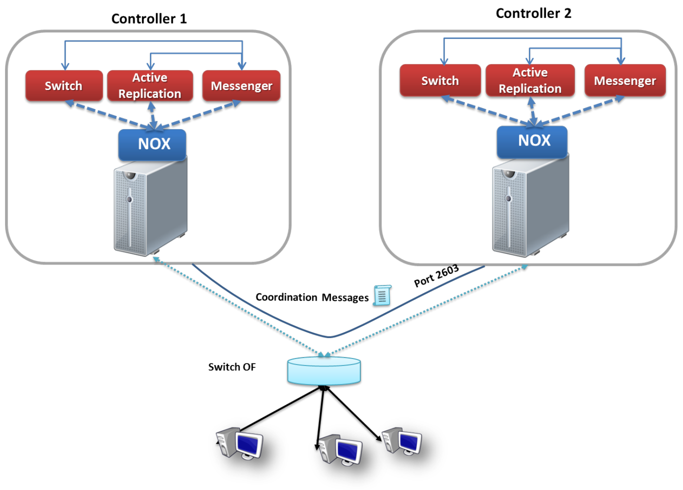

Figura 1.3: Comunicação entre switch e controladores, através da replicação ativa.

Uma abordagem de replicação que usa elementos da replicação ativa e passiva para atingir um nível de resiliência mais flexivel é a abordagem semi-passiva [20]. Nesta abordagem o cliente se conecta e manda a solicitação para todos os servidores, onde cada solicitação é tratada como uma instância do problema do consenso [19]. A solução de cada instância vai corresponder ao controlador que será responsável por processar a solicitação naquele momento. Esta solução provê uma menor *overhead* de comunicação uma vez que não é necessário uma intensa coordenação entre os controladores, menos uso de processamento uma vez que não haverá redundância de processamento de requisições e oferece uma rápida troca entre controladores em caso de falha, dado que ela ocorre de maneira natural pois em cada instância do problema teremos a quantidade de controladores ativos naquele momento e todas as instâncias são independentes entre si.

Em um trabalho anterior relacionado [21] utilizou-se a replicação através da abordagem Primary-Backup [9] para garantir resiliência ao componente switch, responsável pelo repasse de pacotes entre os nós de uma rede. O mecanismo é responsável por manter réplicas consistentes do servidor principal que podem assumir o controle da rede caso ele entre em estado de falha. O objetivo do trabalho era fornecer uma prova-de-conceito de replicação para SDN uma vez que ainda não havia nenhum trabalho que abordasse o tema.

1.2 Objetivo

O objetivo desse trabalho é dotar de resiliência redes definidas por software através de técnicas de replicação. Estas técnicas irão agregar características de auto-recuperação ao NOX, ou seja, a capacidade de detecção, diagnóstico e reparação localizada de problemas resultados de falhas [22], protegendo a camada dos componentes e mantendo a rede em operação, mesmo nos casos onde seja necessária a troca de controladores centrais.

Uma vez que a abordagem visa tratar a tolerância a falhas na camada de aplicação, os mecanismos implementados, por motivos de simplicidade, são limitados a replicação de redes com um switch OpenFlow.

1.2.1 Objetivos específicos

Os objetivos específicos do trabalho são:

1. Identificar as classes de problemas que podem comprometer a camada de aplicação do sistema operacional de redes.
2. Prover um mecanismo de replicação responsável por dotar a rede de resiliência baseado na técnica de replicação passiva.
3. Prover um mecanismo de replicação responsável por dotar a rede de resiliência baseado na técnica de replicação ativa.
4. Investigar como as diferentes técnicas de replicação se relacionam com o protocolo Openflow e seu desempenho na tarefa de manter a rede em funcionamento.
5. Definir os cenários onde cada uma das técnicas é mais adequada.

1.3 Contribuições

O presente trabalho, pelo melhor que pudemos pesquisar, é um dos pioneiros a tratar do assunto de resiliência em redes definidas por software. O gerenciamento de redes centralizado sempre foi visto como algo inviável por grande parte dos pesquisadores devido, principalmente, a preocupações com escalabilidade e segurança em caso de falhas. Enquanto o problema da escalabilidade vem sendo tratado desde a concepção de redes definidas por software com a decisão de projeto de lidar com fluxos ao invés de com cada pacote individualmente, a questão segurança devido à falha em ponto único foi citada apenas pontualmente [7], sem o desenvolvimento de um mecanismo para endereçar o problema.

Nós desenvolvemos dois mecanismos de replicação baseados nas técnicas de replicação passiva e replicação ativa. A limitação em relação ao número de switches, não afeta negativamente a prova-de-conceito. Ambos os mecanismos se mostraram efetivas em tratar o problema da falha em ponto único levantado pelo gerenciamento centralizado, sendo capazes de efetuar a troca de controladores sem interrupção do serviço, continuando do mesmo estado que o controlador anterior de maneira transparente ao cliente. Nenhuma das técnicas exigiu mudanças significativas no protocolo OpenFlow e ambas tiveram suas propriedades mantidas. A replicação passiva se mostrou mais adequada a ambientes onde há a necessidade de uma maior economia de processamento (tanto de CPU, quanto de comunicação) e menor atraso, devido ao fato de não haver necessidade de comunicação ininterrupta entre os controladores e apenas o controlador primário processa as requisições do cliente. A replicação ativa é mais adequada para aplicações de tempo-real, que necessitam que o serviço seja totalmente estável e sem picos de atraso, uma vez que a troca de controladores é mais rápida do que na replicação passiva, pois todos os controladores estão sendo executados concomitantemente com o mesmo nível hierárquico.

Para realizar a coordenação entre os controladores, nós desenvolvemos uma plataforma de comunicação simples responsável por trocar mensagens entre eles. Esta plataforma pode ser utilizada por outros componentes para diversos fins, tais como a extensão da replicação a outros componentes do NOX. Por motivos de simplicidade, assumiu-se que a comunicação é livre de falhas.

Verificou-se que redes definidas por software tornam o desenvolvimento de tais técnicas de replicação simples e dinâmico, a implementação dos mesmos mecanismos na arquitetura tradicional seria de difícil viabilidade devido ao seu alto custo e complexidade.

Em 2012, publicamos um artigo com os resultados preliminares, onde apresentou-se o componente de replicação passiva, chamado CPRecovery [21]. O artigo apresentou o primeiro mecanismo a utilizar replicação para prover um maior grau de resiliência a redes definidas por software. O componente foi testado em vários cenários de falhas como: Interrupção abrupta do processo do controlador; um serviço da camada de aplicação possui algum mau funcionamento que leva o controlador a um estado de falha; e um ataque malicioso faz com que o controlador torne-se inoperante.

Em 2013, foi aceito um artigo que apresenta o estado atual da pesquisa. Nesse artigo, a replicação passiva e ativa são comparadas, utilizando-se métricas para verificar o funcionamento dos componentes em ambientes de larga escala.

1.4 Estrutura do documento

O apêndice B fornece a primeira parte da revisão da literatura, que apresenta o conceito de redes definidas por software, a sua arquitetura geral, o conceito de sistema operacional de redes e o problema levantado pelo gerenciamento centralizado, assim como os principais objetivos do trabalho.

O apêndice C apresenta a segunda parte da revisão da literatura e descreve as propriedades das técnicas de replicação passiva e replicação ativa, assim como uma descrição em alto-nível do seu funcionament, as implementações dos componentes de replicação passiva e replicação ativa, as suas arquiteturas e discorre sobre o funcionamento da arquitetura dos componentes no NOX.

O apêndice D apresenta os experimentos realizados para validar o funcionamento dos componentes de replicação e comparar os seus desempenhos, descreve os cenários utilizados, e analisa os mecanismos de resiliência propostos e os resultados obtidos.

O apêndice E contém os comentários finais sobre os cenários nos quais cada mecanismo é mais adequado, as contribuições do trabalho e trabalhos futuros.

Apêndice A

Introduction

A.1 Objective

The goal of this work is to provide resilience to software defined networks through the use of replication techniques. These techniques will aggregate self-healing characteristics to NOX, i.e., the ability to detect, diagnose and repair problems resulting of failures [22], protecting the layer component and keeping the network operating, even in cases where it is necessary to change the central controllers.

Since the approach we used aims to address fault tolerance at the application layer level, the mechanisms implemented, for reasons of simplicity, are limited to networks with one OpenFlow switch.

A.1.1 Specific Objectives

1. Identify the classes of problems that can compromise the application layer.

2. Provide a replication mechanism responsible for providing resilience to network based on passive replication technique.

3. Provide a replication mechanism responsible for providing resilience to network

based on active replication technique.

4. Investigate how different replication techniques relate to the OpenFlow protocol and its performance in the task of keeping the network running.

5. Define scenarios where each technique is more appropriate.

A.2 Contributions

The present work is one of the pioneers to address the issue of resilience in software defined networks. Centralized network management has always been seen as an anathema by many researchers, mainly due to concerns about scalability and security in case of failures. While the problem of scalability has been addressed from the design of software-defined networks with the design decision to handle flows rather than each individual packet, the security issue due to failure of single point was mentioned only superficially [7], without the development of a mechanism to address the problem.

We developed two replication mechanisms based on passive replication and active replication. Both techniques are effective in handling the problem of single point failure raised by centralized management, being able to make the change of controllers without service interruption, continuing from the same state as the previous controller transparently to the client. None of the techniques required significant changes in the OpenFlow protocol and both had kept the properties cited in their definitions. The passive replication proved to be more appropriate for environments where there is a need for greater economy in processing (CPU processing and communication processing) and lower delay, due to the fact that there is no need for continuous communication between the controllers and the only the primary controller processes client requests. The active replication is more suitable for real-time applications that require the service to be completely stable and without peaks of delay, since the change of controllers is completely transparent to the user because all the controllers are running concurrently with the same hierarchical level.

In order to make the coordination between controllers was developed a communication platform responsible for exchanging messages between them. This platform can be used by other components for various purposes, such as the extension of the replication to other NOX components. For purposes of simplicity, we assumed that the communication is failure-free.

It was verified that software defined networks make the developing of these replication techniques dynamic and simple, the implementation of such mechanisms in traditional architecture would be of difficult viability due to their high cost and complexity.

In 2012, we published a paper with the preliminary results, which showed up the passive replication component, called CPRecovery [21]. The paper presented a first replication mechanism used to provide a greater degree of resilience to software defined networks. The component has been tested in various failure scenarios as: Abrupt interruption of the controller process controller, a service from the application layer has some malfunction that leads the controller to a failure state, and a malicious attack causes the controller enters in a unresponsive state.

In 2013 it was accepted a paper that presents the current state of research. In this paper a comparison is made between active and passive replication, where metrics used to verify operation of components in large scale environments.

Apêndice B

Background and Related Work

One of the approaches that aims to simplify the increasingly complex network management is the Software-Defined Networking (SDN) [1]. SDN provides a new paradigm for building management applications for networks and a new way to look for the resolution to the many problems which exist throughout the Internet today, such as routing [2], security [3], mobility [5], and accountability [6].

The OpenFlow protocol is the main reason for the increasing popularization of the SDN approach. This protocol provides the interface between the control and data planes [23] of the network which enables a seamless communication between these elements. On the control plane, there exists a network operating system responsible for managing the applications that provide the core services of a network (e.g. routing, authentication, discovery). These management applications, hereafter called *components*, use the API of the network OS to communicate with other components and to control the network resources. Any switch with OpenFlow protocol support (also known as "datapath"), and independent from any other hardware-specific characteristics, can be managed by the network operating system as a network resource. The network OS uses a secure channel and the OpenFlow protocol to receive information about the data plane and to apply actions requested by components.

Centralized network management in SDN simplifies the task of managing the services and reduces the level of difficulty to perform complex services; however, it raises, among other issues (e.g. scalability), the problem of vulnerability of the network. This occurs because in the event of a network OS failure the whole network can be compromised, since all the applications and services depend on it. The OpenFlow protocol provides the possibility to configure one or more backup controllers which can assume the network control in case of failure, but OpenFlow does not provide any coordination mechanism between the primary controller and the backup. Hence, the components will lose all configuration and other information previously gathered from the network. Thus, any protocol or service running in a network OS must be protected by a mechanism providing resilience to failures, capable of performing the coordination between controllers to keep the backup consistent with the primary, which will turn the network to a safe state with minimal overhead to hosts and switches.

The goal of this work is to investigate (1) how different replication techniques relate to each other, (2) how each one performs on the task of providing resilience to a SDN, and (3) which technique is the most suitable for different scenarios. We will describe the mechanisms implemented that improves resilience in SDN utilizing component organization, and its implementation and integration with a core component of the network OS.

SDN is event-oriented, i.e. components run independently on top of the network OS, receiving events from the network or from other components. Handling these multiple types of events, we have successfully developed a novel component that increases resilience on the network. Passive replication [9] and active replication [16] are mechanisms that offer resilience against several types of failures in a centralized controlled network and provide a seamless transition (from the host's point-of-view) between controllers.

The SDN component organization approach markedly improves the task of developing and deploying new network management applications. The interface provided by

SDN allows the communication between components, making the development of such type of service much less complex than the traditional hardware-dependent and totally distributed approach. The use of high-level abstractions to represent the resources of the network simplifies the task of maintaining consistence between controllers.

Innovative components that control networks using the SDN approach already exist for mobility control [24], datacenter network management [25] and production network slicing [26], which illustrate the strong potential of the SDN technology.

We have developed and implemented two resilience components for SDN that emulate passive and active replication, respectively. In our initial tests, both components proved to be efficient and adequate in the context of SDN, where the passive replication is more suitable for a less intrusive approach and active replication is suitable for scenarios with no delay tolerance, like real-time applications.

In this work, our main contributions are as follows: (1) developed two functional implementations utilizing the passive replication and active replication enabling resilience; (2) implemented a platform for communication between SDN controllers extending the messenger component of the SDN controller; (3) extended the component responsible for forwarding packets to send state update messages to the server replicas; and (4) ran experiments in the Mininet environment to simulate different scenarios and topologies to investigate how replication mechanisms impact the functioning of the network.

This work is organized as follows: Appendix C describes the passive and active replication techniques, our active and passive replication components, their architectures and implementations. Appendix D analyzes the proposed resilience mechanisms, our experiments and results obtained. Appendix E presents final remarks and future work.

Apêndice C

Replication Mechanisms

To use a centralized approach to provide network services can lead to serious problems. Among these problems we focused, in our work, on the single point of failure, because it can compromise the network performance. There are many situations which can lead the network OS to a state of failure, such as a NOX server shutdown, a management application entering into an infinite loop, two applications competing to access the same network resource, or even an attack from another network. To avoid such failures in SDN, a switch with OpenFlow support can be previously configured to connect to a backup controller that remains in wait state until the switch connection is completed. Once the switch connects, the controller restarts the whole process of network configuration. Meanwhile, some services will be unavailable and resources will be wasted. Thus, a resilience mechanism must be provided to guarantee that, in case of failure, another network OS can smoothly take control of the control plane from a last valid state [27].

One of the approaches to increase resilience for centralized architectures is replication. Passive replication and active replication are two of the main approaches used. The following sections will detail their properties and features.

C.1 Passive Replication

In this technique one or more backup servers (also known as secondaries) are kept consistent with the state of the primary server, and as soon as the primary server enters in a failure state, one of the secondaries is chosen to take control of the network. The passive replication (also known as Primary-Backup Replication) approach used works this way.

To be effective, a primary-backup protocol must hold the following properties:

P-1 There is a local predicate $Primary_s$ associated with the state of each server s. At any moment, there exists at most one server s which satisfies $Primary_s$.

P-2 Each client i keeps a destination predicate $dest_i$. If a client needs a request, it sends a message to $dest_i$.

P-3 If server s receives a request and it is not primary ($Primary_s$ does not hold), it does not queue the request.

P-4 There exist k and Δ values such that the server behaves as a single server (k, Δ) - *bofo* (bounded outages, finitely often).

Property P-1 guarantees that the network behavior using primary-backup is indistinguishable from the behavior of a network with a *bofo* server under failure periods, where these periods can be grouped in k time intervals, with each interval lasting at most Δ. This property ensures that any implementation of the protocol needs to limit the time where service is down. The following sections will demonstrate how all these properties hold in our implementation.

C.2 Active Replication

The State-Machine Approach (also known as Active Replication) is another well-known replication technique. In this approach, the client connects with all servers simultaneously, the servers process the requests, then the reply is sent to the client. An implementation of this method must follow these properties:

A-1 To avoid handling duplicate responses from servers, the client can: a) only process the first reply to its request, or b) the servers must coordinate in order to decide which one will respond to the client.

A-2 All servers must have a deterministic behavior. Thus, if a server is in state a and receives a request it must specifically go to state b.

A-3 The client must have a communication primitive that allows sending all requests to servers in the same order.

Property A-1 must be kept to avoid the client, inadvertently, changes its state due to duplicate requests. Properties A-2 and A-3 guarantee that all servers are synchronized in the same state, working as state-machines.

C.3 Component-Based Architecture

In the SDN approach a component is a piece of software that is responsible for a specific task in the network management. Components access the network through an API provided by the network operating system. The main events that occurs in a network (e.g. a switch connects with a controller, the controller receives a packet to be processed) are passed along with their attributes (e.g. a *packet-in event* contains an incoming port, destination IP, etc.). A component can register itself to events that are important to its task, for example, the routing component should register to receive all *packet-in* *events*. Each component has a handler module which is in charge of processing this event message. Components can also generate event messages and send them to other components if those provide some important service, e.g. a switch component generates a *packet-in event* message that is processed by an authenticator component in charge of registering users. Furthermore, new events can be created to represent some aspects of the network not previously defined in the default set of events. NOX components are divided into three classes: (a) *coreapps*, which are responsible for the basic functions of the network OS; (b) *netapps*, which manage aspects related to network services (e.g. routing, authentication); and (c) *webapps*, which provide web services. These components can interact with each other to optimize the accomplishment of their tasks, but, they can also be instantiated independently. A component-based architecture stands on the relationships between management applications. A particular component can provide a service which can be useful to many other components or it can provide a service useful only to its own operation. It is also possible to enforce dependence between components when a service is necessary for the execution of other component. The work described in this work implements a component which provides a backup to the *switch* component. The switch component belongs to the *coreapps* class and is responsible for processing the incoming packets by setting up flows in the datapath.The next sections describe in detail the internal workings of replication components.

C.4 Passive Replication

C.4.1 High-Level Description

This section describes the whole process of replication between the *switch* component running on the primary controller and the secondary controller. In addition, it describes the process of recovery when the primary fails. The former phase is called *replication phase*, the latter is the *recovery phase*. In the replication phase, the passive replication component acts during the normal operation of the network OS. The following actions occur during this period:

1. The network OS receives a *packet-in event* indicating the arrival of a packet whose related actions must be included in the switch's flow table.

2. The network OS verifies if it is the primary server in the network, i.e. the flag *isPrimary* must be true. In this case, it verifies if there is a source table entry related to the datapath which sent the packet.

   (a) If there is an entry in this table, it looks for a destination port, creates a flow associated to this packet, associates this flow to necessary actions (if they exist) and sends the packet.

   (b) If there is no entry, it creates a new entry in the source table associating its MAC address to the datapath and port. Then, it verifies if the secondary network OS is reachable. If the secondary network OS is reachable, the primary server generates a message with the MAC address and datapath identification which must be inserted in the source table. Then, it sends this message to the secondary network OS and waits an *acknowledge* message confirming that the *state update* message was received. Otherwise, it continues its normal operation.

3. If the network OS is a secondary server, i.e. the flag *isPrimary* is false, it ignores the request (as defined in the primary-backup specification).

The switch also sends an *inactivity probe* if the connection with the controller enters into an idle state and waits for a T amount of time. If the controller does not send a reply in the wait time, the switch assumes that the controller is down. In the recovery phase, the passive replication component acts during a failure state of the primary controller, e.g. when the NOX process is aborted, the link between NOX and the

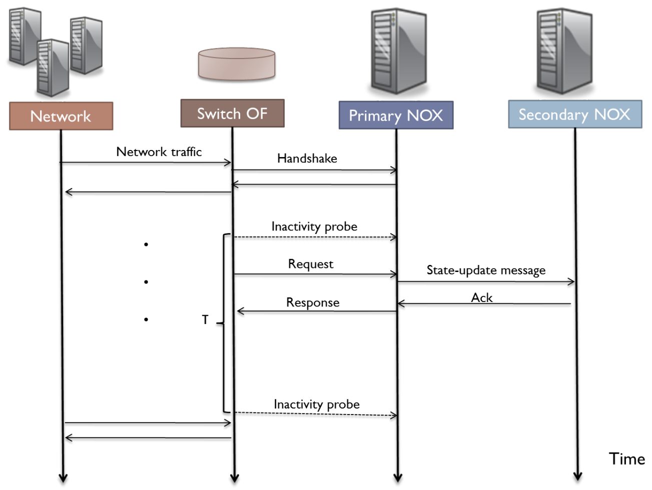

Figura C.1: Passive replication protocol in SDN.

switch is down or, in the same situation described above, the controller does not reply the *inactivity probe*. The following actions occur during this period:

1. The network switch searches for the next network OS in its list (defined through the switch configuration) and starts a connection to it.

2. The secondary network OS receives a connection request from the switch, generates a *datapath join event* and changes its internal state to primary, the flag *isPrimary* must be set to true.

3. The secondary network OS takes control of the switch, becoming the primary. Additionally, the new current primary keeps trying to send the *state update messages* to the former primary controller, which will become a secondary controller if it enters into proper functioning again.

Figure C.1 illustrates the relationship between SDN entities using the primary-backup approach implemented. Handshake is the process of authentication between an OpenFlow switch and the primary network OS. Initially, the switch establishes a connection with the primary controller; eventually, this controller enters into a failure state and does not respond to the *inactivity probe*. Then, the switch successfully tries to connect to the secondary controller, completes the handshake, and continues its normal operation.

C.4.2 Implementation

Both our replication components were implemented in the C++ language in order to increase the performance and efficiency of our solution, and to minimize the blocking time—one of the cost metrics of this technique. The passive replication component was added to the *coreapps* class of components. This component has methods in charge of inter-NOX communication, processing components' messages, and sending them to the network. The passive replication component maintains the data structure, called *source table*, of the *switch* component running on the secondary controller, updated. This *source table* is used to relate a MAC address to a switch port.

The first method, called *sendUpdateMessage*, is in charge of sending *state update messages* to a secondary network OS. It is called by the *switch* component, which parses a string with a datapath_id and MAC address as parameters. This string is parsed and recognized by the network OS. To send the message the component uses the *send* function from the default C socket library. The message is sent to port 2603 of the secondary controller, because *messenger*, the component in charge of message receiving in NOX, listens to this specific port. The method *switchBackup* locally stores the switch's source table, which can be recovered later.

The *changeStateToPrimary* method is called when the network OS takes control of the network. This method identifies these changes through a *Datapath Join Event*. When this event is generated, the switch is connected to the network OS. Once the network OS receives this event, it changes its state to primary (*isPrimary*=*true*). As the network OS can manage more than one switch, it performs this action only for the first switch connection.

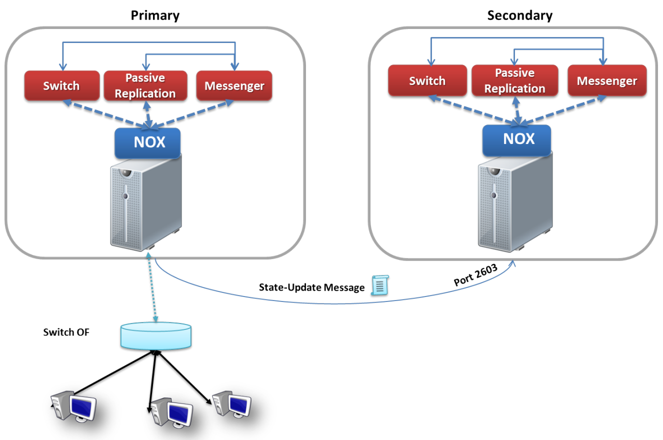

Figura C.2: Communication between Primary and Secondary controllers, through passive replication component.

To enable seamless communication between components, we performed a set of key extensions to the original NOX switch component. A *message handler* method was added to the *switch* component. This method receives messages directed to the controller (specifically, *state update messages*) through the *messenger* component. The handler identifies if a message is of type string, parses and divides it into two substrings representing a MAC address and a datapath_id, and initializes the variables *ethernetaddr* and *datapathid* with these values. Then, the handler inserts the variables into the secondary component's source table, keeping it consistent with the primary network OS. This process only occurs if the network OS is in a secondary state (the flag *isPrimary* must be false). The most important method of the switch component is called *packetInHandler*. It was modified to process packets only if the network OS is not the secondary. If the network OS is the primary, the datapath_id and the MAC address are encapsulated in a message and the *sendUpdateMessage* method is invoked to update the state of the secondary network OS.

One of the goals of our implementation was to hold true the primary-backup properties. A flag called *isPrimary* was created in the *switch* component to represent the local predicate $Primary_s$, cited in property i of Section C. When NOX starts this flag is initialized as *false* (the *primary-backup* specification allows that at a given moment no controller exists that satisfies $Primary_s$). The OpenFlow reference implementation used defines that a switch connects only with one controller. Thus, the component changes its state when the first switch connects. The OpenFlow keeps the server identity $dest_i$ cited in property P-2, only sending its requests to the controller to which it is connected.

Before processing the incoming packets, the *switch* component verifies the *isPrimary* flag. If it is false, the component skips the processing block, ignoring the *packet-in event*, and sends a signal to NOX to continue normally. Hence, property P-3 is kept.

If the network enters into a failure state, eventually, it will connect to a controller in its list, this controller will process the requests, and the network will return to normal operation. These steps ensure that the downtime of the network is limited, since the network manager guarantees that at least one of the controllers in the switches' list is available. Figure C.2 depicts the passive replication component relationship with other network OS components. Different network OSes communicate through the passive replication component that utilizes the methods provided by the *messenger* component. We provide more details about the *messenger* component, which is in charge of message exchange, in the next section.

C.5 Active Replication

C.5.1 High-level Description

In the active replication protocol the *switch* component runs on all controllers simultaneously. Given that the controllers must act like a state-machine, there is no discrimination between them and they all pass through the same steps. In the replication phase, the active replication component coordinate the process of every request. The following actions occur during this period:

1. The switch sends the request to all controllers simultaneously.

2. The network OS receives a *packet-in event* indicating the arrival of a packet whose related actions must be included in the switch's flow table.

3. The network OS process the request, stores the id of that request and its timestamp.

4. If there are other controllers active, the controller sends the id of the request and its timestamp to all other controllers and waits for the ids and timestamps of other controllers. If there are no controllers running, the network OS sends the response to this request.

5. If one or more timestamps are smaller than the current value, then the network OS does not send its reply. Otherwise, it responds the request.

Unlike the passive replication mechanism, in the recovery phase of active replication there is no change of controllers, if one of the controllers enters in a failure state the others will not be affected because they will automatically detect the failure when the connection with the failed controller becomes unresponsive.

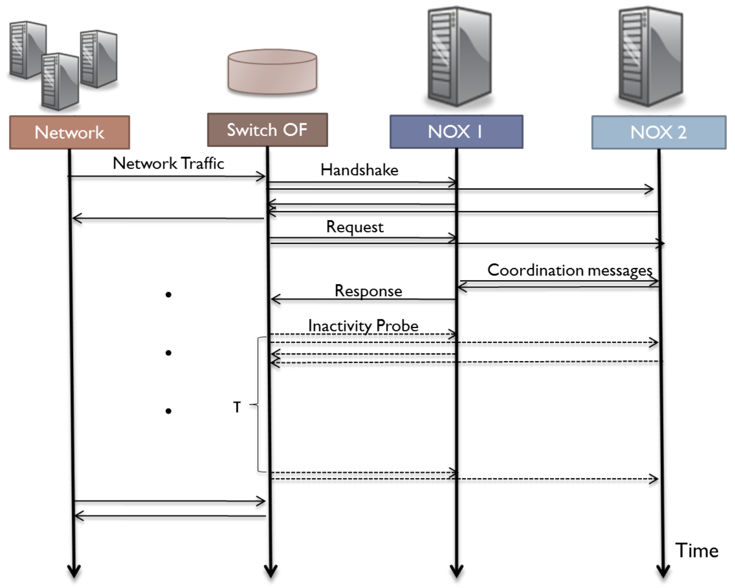

Figura C.3: Active replication protocol in SDN.

Figure C.3 depicts the relationship between SDN entities in our active replication component. Initially, the switch establishes a connection with both controllers and when a request is sent to the switch there is a coordination phase, where the controllers decide which one will respond to the request.

C.5.2 Implementation

The OpenFlow protocol implementation used with this component behaves differently from the implementation used with the passive replication component. In this implementation the switch connects with all controllers simultaneously, sends the requests to all controllers, and processes all responses. To hold the property i cited in Section C.2 the controllers must have coordination to avoid the duplication of responses.

The method responsible for the delivery of messages between controllers is the *sendTimestamp*. The switch component calls this method after processing the packet, sends a message to other controllers to decide whether it will send the reply to the *switch*. This method uses the same socket library and port used by the *sendUpdateMessage* of

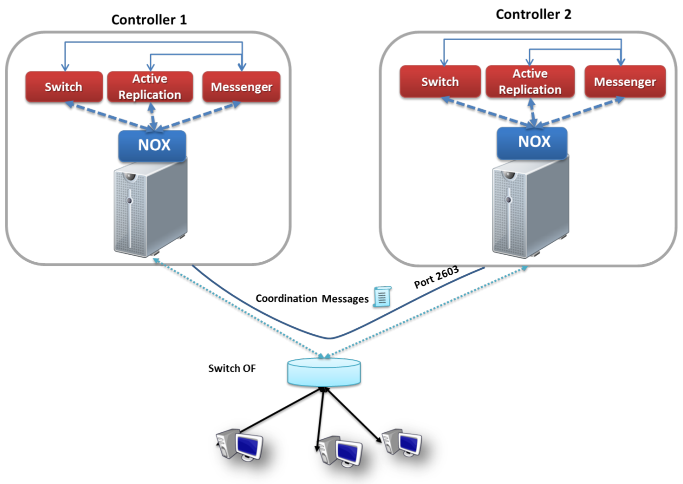

Figura C.4: Communication between switch and controllers, through active replication component.

the passive replication component.

When a switch receives a request from a host it sends a *packet-in event* to all controllers. This request is processed by the *switch* component. The *packetInHandler* was modified to save the time (in microseconds) when the packet was processed and send it to other controllers. If all controllers reply indicating that none of them has a lower timestamp, it sends the reply to the switch. If at least one of the other controllers send a message indicating that its timestamp is lower than the timestamp sent, the boolean variable *processPacket* is set to *false* and the packet is not processed.

To receive the timestamps of other controllers, we added method *messageHandler* to the *switch* component. This method processes and receives the messages, performs the parse of timestamp and *xid*, compares with the timestamp associated with the *xid* received and, then, sends a message to the controller indicating the result of the comparison.

Figure C.4 illustrates the active replication component relationship with other network OS components. This component also uses the *messenger* component. The next section contains more details on this relationship.

Apêndice D

Experimental Analysis

D.1 Earlier Experiments

This section is part of a proof-of-concept implementation of passive replication mechanism. These experiments aimed to evaluate the viability of an replication solution to increase resiliency in SDN. We tested our mechanism in differente scenarios to verify if it will be capable to continue the network functioning with minimal packet loss. We also tested the did experiments with different number of replicas (replication degree) to measure the overhead in the network performance. For purpose of simplicity, our mechanism only supports networks with one OpenFlow switch. Scenarios with more switches may lead to inconsistency between controllers, to aid these problems is necessary to use more complex coordination mechanisms that considers failure in the controllers' communication [10] [11]. This limitation does not compromise the validity of our proof-of-concept because we address the resiliency at application layer level.

D.1.1 Experimentation in Mininet

The passive replication component was deployed and tested using Mininet [12]. Mininet is a new architecture for network emulation using OS-level virtualization features. It

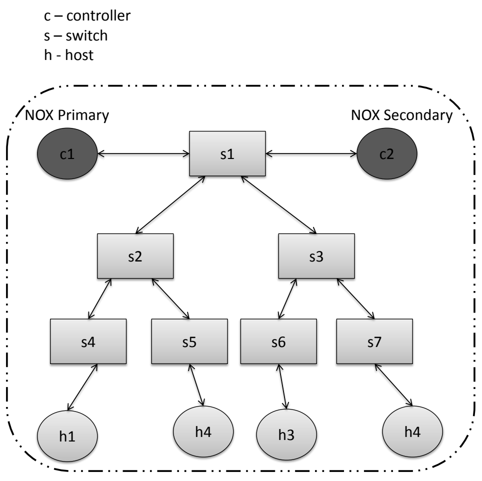

Figura D.1: Mininet topology deployed for scenarios I and II

allows the prototyping of SDN topologies with links, hosts, switches and controllers. This architecture proved to be very flexible, scalable and realistic when prototyping and evaluating our proposed approach.

The passive replication component was tested in different failure scenarios. These scenarios are described in the next sections.

D.1.2 Scenario I: NOX Abruptly Aborts

From the operating system's point-of-view, the NOX is a user process. Thus, the OS can send a terminate signal to this process, making NOX to exit without sending any message to switches or saving any context. It is possible that the OS itself can enter in a failure state, interrupting NOX's operation. This scenario was tested in the topology described in figure D.1. The passive replication component was instantiated on both c1 and c2 controllers: the former acting as a primary network OS and the latter as secondary. The *ofprotocol* implementation of the OpenFlow protocol was used to configure the connection with the backup servers as soon as possible after a failure of a primary network OS.

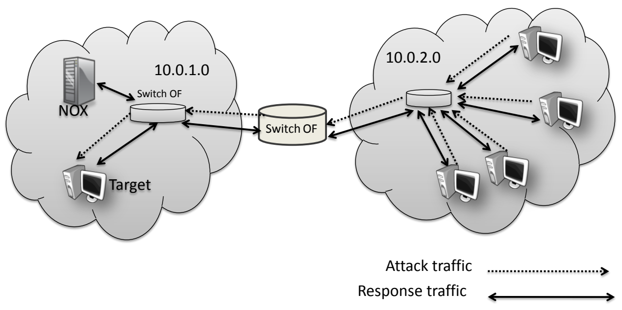

Figura D.2: Topology of the DDoS attack scenario III

D.1.3 Scenario II: An Application Enters in a Failure State

As previously explained, the components run on the top of the network operating system. Thus, an application can leads the network OS into a failure state. This situation will lead NOX to do not reply any other switches' requests. We built an application that creates a socket, connects with the secondary controller and waits for a message. The same topology described in figure D.1 was used. The switch sends a *inactivity probe* to the controller, and as soon as it does not receive any confirmation, it tries to connect with the secondary. The wait time is configured in the *ofprotocol* and must be equal or greater than the *blocking time*.

D.1.4 Scenario III: A DDoS Attack

Distributed Denial-of-Service attacks (DDoS) are one of the main security problems in the Internet. An OpenFlow network is susceptible to this kind attack, since NOX processing capacity can be overwhelmed by a massive number of flow initiations. The topology used to simulate this kind of attack is described in figure D.2. We create two subnets in Mininet, where the subnet 10.0.1.0 is the target and subnet 10.0.2.0 is the attacker. The hosts of the attacker subnet execute a script that continuously creates and sends IP packets with random headers (i.e. with random IP source) to the target subnet. Each packet generates a *packet-in event* in NOX, requiring a creation of a new flow entry in the switch's flow table. The large traffic leads the primary NOX into a non-responsive state as soon NOX loses communication with the switch. Once, the switch does not receive any reply to the *inactivity probe*, it connects with the secondary controller. To increase resilience, the passive replication component must be used in conjunction with a DDoS detection mechanism (e.g. the method proposed in [13]), given the fact that the secondary controller also is susceptible to the DDoS attack.

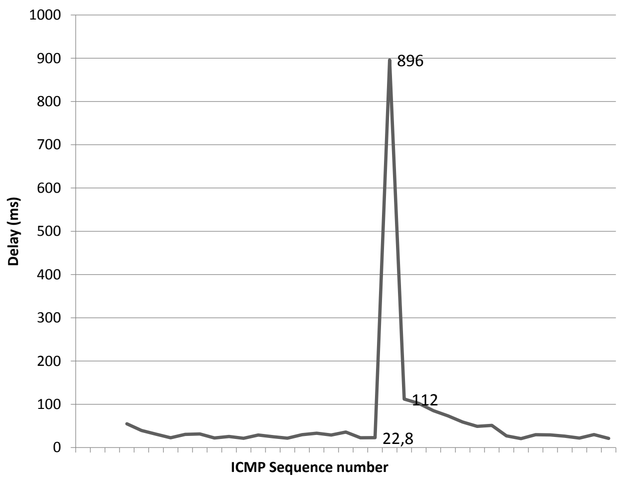

Figura D.3: Graph representing the delay when the primary NOX fails and the secondary NOX takes control of the network.

D.1.5 Evaluation

To analyze the component we evaluated the behavior of the network when the primary enters in a failure state and the switch connects with the secondary, so the latter can assume the control of the network. NOX only processes the first packet of a flow and sets up a flow entry in the switch. Thus, the delay of the following packets is independent from the NOX functioning. To measure the impact of changing the controllers, we configured NOX to process every packet that the switch receives. The graph in figure D.3 represents the increase of delay during the change of the controllers. Despite the delay increase (usually the average value of delay is 20 ms and it arises to around 900 ms), it can be noted that there is no packet loss and after the secondary controller takes control of the network the delay quickly returns to its normal state.

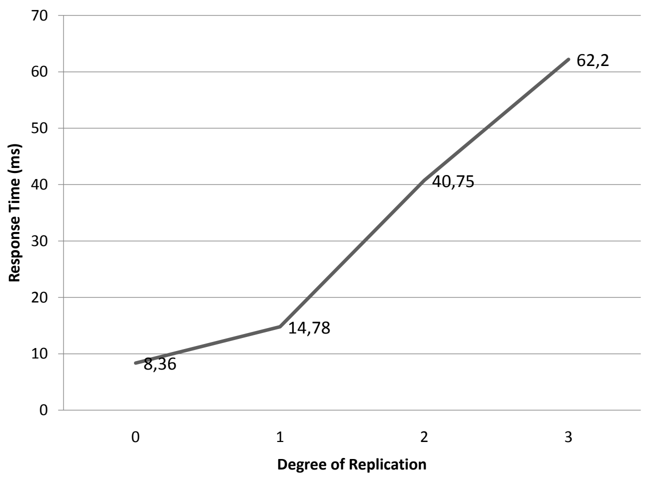

Figura D.4: Response time for different degrees of replication.

Another measurement that we deployed was a variation in the response time to different replication degrees. The response time is the time that NOX takes to process a request from the switch, sends a *state update message* to the backup controllers, receives a confirmation and sends a confirmation to the switch. The OpenFlow protocol implementation used in this experiments allows connection with up to four controllers. Since one must be the primary, our experiments used up to three backup controller ($replicationdegree = 3$). The bandwidth of each link between the primary and the backup controllers was set to 800 $Mbps$.

Figure D.4 represents the increasing of response time for different replication degrees. The average response time without any secondary controller is around 8 ms, thus this can be defined as the time needed to the controller processes the switch's request, updates its state and sends a confirmation to the switch. When the replication degree is 1, the average response time becomes 14 ms, an increase of approximately 75 %. This overhead is caused by the time needed to send and receive a confirmation from the secondary controller. As the replication degree is increased, the response time also increases because the primary needs to wait a confirmation from each secondary controller.

D.2 Active and Passive Mechanisms

Our active replication component (as well as the passive replication component) was deployed and tested using Mininet [12]. We tested both components under different scenarios of high load in order to compare their behaviors and how the overhead caused by the communication between controllers impact the network performance. For purpose of simplicity, our active mechanism only supports networks with one OpenFlow switch. A higher number of switches may lead to inconsistency between controllers, because different switches can send requests in a different order to controllers, causing different controllers in different states. These problems can be addresssed using more comprehensive coordination mechanisms that considers failure in the controllers' communication [10] [11]. This limitation does not compromise the validity of our proof-of-concept because we address the resiliency at application layer level.

In our experiments we used a implementation of OpenFlow version 1.0.0-rev1 and NOX version 0.9.0. First, we observed how the component influenced the delay between hosts as the number of hosts increased. The number of hosts is directly proportional to the number of requests that must be processed by the replication component, since the *source table* will be constantly updated. In our tests, we used topologies with 100, 250, 500, 750, and 1000 hosts, in which each host connects to a central switch that forwards every packet. To generate traffic, each host sends a ICMP request to its neighbor simultaneously, thus, generating a different request for the replication component.

Our experiments showed that the passive replication mechanism achieves a lower latency (time necessary from client to send the request and receive a reply) than the active replication component, as depicted in Figure D.5, and grows slowly as the number of hosts increases. This behavior is due to the fact that the passive mechanism exchanges fewer messages than the active replication component and, since only the primary controller processes all requests, a higher rate of requests will lead to a higher latency. The active mechanism consumes more CPU power because every replica processes each request and the coordination between them is more complex than the communication between the *primary* and *secondary* controllers. We can also observe that latency in the scenario with active replication mechanism is more affected by the overhead caused by the controllers coordination than by the increase of number of hosts. This happens because the traffic is balanced between the controllers and hence the impact of the number of hosts is less significant.

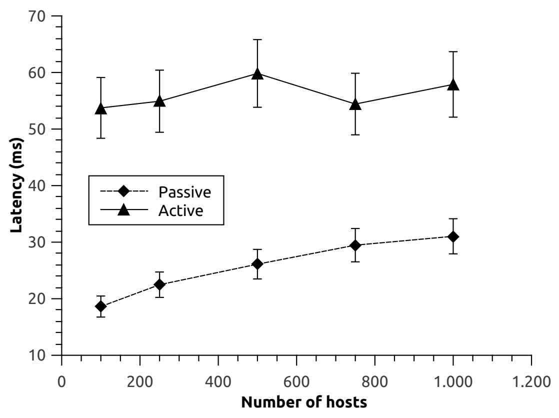

Figura D.5: Latency between hosts in active and passive mechanisms.

Other experiment that we performed, with results depicted in Figure D.6, was to simulate a controller failure to observe how the mechanisms react to keep the network well-functioning. The topology used in this case was minimal, i.e. two nodes, in order to isolate the impact of the failure. In both scenarios the controller enters in failure state at 15 milliseconds. In the passive replication component, the latency suffers an increase caused by the period of change between controllers, where the secondary takes control of the network. In the active replication, there is no need for such a change since, as previously explained, all controllers are connected at the same time and the failure is organically detected by other controllers.

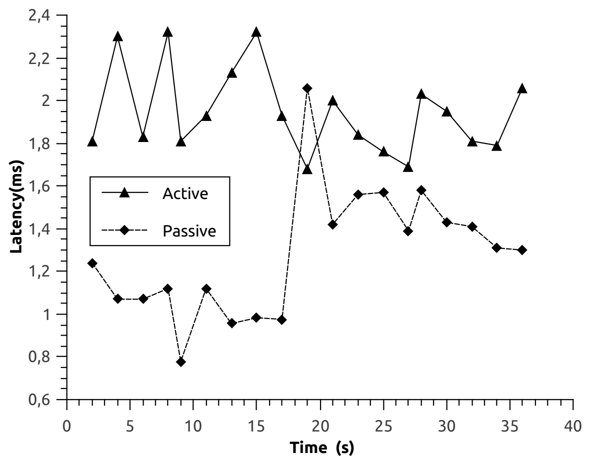

Figura D.6: Latency between hosts in case of controller failure.

In order to determine the number of samples used in our experiments, we performed pilot tests for each scenario to determine the number of experiment replications larger enough to provide result reliability [4]. All experiments were made with a confidence level of 90%. Each scenario needed a different number of samples oscillating from 20 to 60 replications.

Apêndice E

Discussion and Related Work

Even though SDN is a recently developed technology, it is being recognized as one of the most promising architectures to solve many important problems that plague the Internet today. Although research in this area is growing very rapidly, to the best of our knowledge there has been only a very small number of published papers reporting work in our line of research.

Research on failure recovery in SDN is in its infancy. Short discussions covering this topic can be found in [7] and [14]. The concepts of a network OS and software-defined management applications are very distinct from the current Internet architecture. Recent work which studies distributed recovery and healing mechanisms is presented in [15].

A benefit of passive and active replication is that they guarantee that the state of *switch* component are consistent, increasing the system reliability. The general approach employed in the implementation of the recovery mechanisms also enable it to offer services for other components in the network OS, such as authentication, access-control, and load-balance.

Our experiments showed that both components proved to be efficient and adequate to the task of providing resiliency to SDN, where, due to its lower overhead and no need of controller coordination, the passive replication is more suitable for a less intrusive approach, but active replication is suitable for scenarios with no delay tolerance (e.g. real-time applications) because the change of controllers in a failure scenario is organic. The number of switches limitation does not affect negatively our results because we aimed a proof-of-concept of replication in SDN and only addressed the resiliency at application layer level.

E.1 Final Remarks and Future Work

SDN is a promising technology because it offers flexibility in the development of network management applications. In this work, we presented a novel mechanism for resilience to failures in SDN. This mechanism is implemented as a component, i.e. a software application running in the network OS.

In this work, we developed a component for resilience improvement in NOX utilizing its component organization. This kind of organization has demonstrated itself as a much easier and simpler approach to implement management network applications without efficiency losses. In addition, we used the Primary-Backup method to improve resilience in a SDN network. This approach is suitable to this type of network, where a centralized control exists and concentrates the information that needs to be replicated in one point of the network; unlike the distributed approach, where the controller would need to collect the information needed from each switch.

The OpenFlow protocol has also proved itself appropriate to this kind of approach, providing mechanisms that detect if a switch loses connection with the controller, e.g. an inactivity probe message is sent periodically to the controller. The protocol also provides the flexibility to configure an idle connection with a backup controller and the time interval for sending an inactivity probe.

Our component also allows communication between controllers using the *messenger* component, which provides an interface for receiving and sending messages through NOX, but does not perform the solicitation of connection establishment or generation of messages. This inter-NOX communication was used to send and receive *state update messages*, but it can also be used to perform other types of communication, e.g. exchange of policies, route announcements. We extended the *switch* component to utilize this platform to send and receive the *state update messages* and to parse the message and update its state if it is running in the backup controller.

Lastly, we performed experiments to measure the impact of the component in the network performance, its functioning during a transition from a *replication phase* to a *recovery phase*, and its behavior in different failure scenarios. The component proved itself functional in all scenarios (e.g. when NOX aborts, or a component compromises the well functioning of the network, or a DDoS attack occurs), providing an elevated level of versatility. The overhead generated by our component does not compromise the communication between hosts, i.e. the operation of our switch remains transparent to the hosts.

Future work will extend this work by integrating it with a new approach for recovery management, currently in development, which is based on artificial intelligence techniques and previous work reported in [17]. This new approach will be built on top of the passive replication component. We also plan to extend the core implementation of NOX to provide a replication platform API that will allow any component to use methods to seamlessly enable replication of its state.

Future experiments will have more complex scenarios, supporting any number of switches and with a more complete communication platform that considers failure between controllers' communication. To address these issues we plan to use more comprehensive fault tolerance and communication mechanisms [10] [11]. We also plan to implement other replication approaches in order to use different replication techniques for different set of controllers.

Referências Bibliográficas

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[2] M. Suchara, A. Fabrikant, and J. Rexford, "Bgp safety with spurious updates," in *INFOCOM, 2011 Proceedings IEEE*, April 2011, pp. 2966 –2974.

[3] J. Bolot and M. Lelarge, "A new perspective on internet security using insurance," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, April 2008, pp. 1948 –1956.

[4] V.K. Chadha, "Sample size determination in health studies," in *NTI Bulletin*, vol. 38., 2006, pp. 55 –62.

[5] P. Zhang, A. Durresi, and L. Barolli, "A survey of internet mobility," in *Network-Based Information Systems, 2009. NBIS '09. International Conference on*, August 2009, pp. 147 –154.

[6] K. Argyraki, P. Maniatis, O. Irzak, S. Ashish, and S. Shenker, "Loss and delay accountability for the internet," in *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*, October 2007, pp. 194 –205.

[7] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 105–110, July 2008. [Online]. Available: http://doi.acm.org/10.1145/1384609.1384625

[8] "Openflow switch specification. Version 1.1.0," http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf, February 2011.

[9] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, *The primary-backup approach.* New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 199–216. [Online]. Available: http://portal.acm.org/citation.cfm?id=302430.302438

[10] L. Lamport, "Paxos made simple," in *ACM Sigact News 32.4* vol. 32, pp.18–25, 2001.

[11] P. Hunt, M. Konar, F. Junqueira and B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, vol. 8, pp.11–25, 2010.

[12] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets '10. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[13] R. B. Braga, E. M. Mota, and A. P. Passito, "Lightweight ddos flooding attack detection using nox/openflow," *Local Computer Networks, Annual IEEE Conference on*, pp. 408–415, 2010.

[14] S. Das, G. Parulkar, and N. McKeown, "Unifying packet and circuit switched networks," in *GLOBECOM Workshops, 2009 IEEE*, December 2009, pp. 1 –6.

[15] Z.-G. Ying, W. Ren, and J.-P. Li, "Effects of topologies on network recovery," in *Apperceiving Computing and Intelligence Analysis, 2009. ICACIA 2009. International Conference on*, October 2009, pp. 207 –210.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, pp. 299–319, December 1990. [Online]. Available: http://doi.acm.org/10.1145/98163.98167

[17] A. Passito, E. Mota, and R. Braga, "Towards an agent-based NOX/OpenFlow platform for the Internet," in *Workshop on Future Internet Experimental Research*, May 2010.

[18] R. Bennesby, P. Fonseca, E. Mota, and A. Passito. An inter-as routing component for software-defined networks. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pp. 138–145. IEEE, 2012.

[19] T. Deepak Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[20] X. Defago, A. Schiper, and N. Sergent. Semi-passive replication. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pp. 43–50. IEEE, 1998.

[21] P. Fonseca, R. Bennesby, E. Mota, and A. Passito. A replication component for resilient openflow-based networking. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pp. 933–939. IEEE, 2012.

[22] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[23] S. Das, A. Sharafat, G. Parulkar, and N. McKeown, "Mpls with a simple open control plane," in *Optical Fiber Communication Conference and Exposition*

(OFC/NFOEC), 2011 and the National Fiber Optic Engineers Conference, March 2011, pp. 1 –3.

[24] K.-K. Yap, R. Sherwood, M. Kobayashi, T.-Y. Huang, M. Chan, N. Handigol, N. McKeown, and G. Parulkar, "Blueprint for introducing innovation into wireless mobile networks," in *Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures*, ser. VISA '10. New York, NY, USA: ACM, 2010, pp. 25–32. [Online]. Available: http://doi.acm.org/10.1145/1851399.1851404

[25] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying nox to the datacenter," in *Eighth ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[26] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, S. Seetharaman, D. Underhill, T. Yabe, K.-K. Yap, Y. Yiakoumis, H. Zeng, G. Appenzeller, R. Johari, N. McKeown, and G. Parulkar, "Carving research slices out of your production networks with openflow," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 129–130, January 2010. [Online]. Available: http://doi.acm.org/10.1145/1672308.1672333

[27] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07. New York, NY, USA: ACM, 2007, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1282380.1282382