Universidad Zaragoza
1542

Trabajo Fin de Grado

Defensa proactiva y reactiva ante ataques DDoS en un entorno simulado de redes definidas por software

Autor

Jorge Paracuellos Cortés

Director

Ricardo J. Rodriguez

Escuela de Ingeniería y Arquitectura / Universidad de Zaragoza

2016

Defensa proactiva y reactiva ante ataques DDoS en un entorno simulado de redes definidas por software

RESUMEN

Las redes definidas por software (Software Defined Networking, SDN) presentan un cambio de paradigma para las redes de comunicaciones debido a la separación del plano de control y de datos, que abstrae el elemento *hardware* del elemento software y dispone de un elemento central (controlador) que gestiona la red de manera centralizada. Es una arquitectura de red flexible, gestionable, adaptativa y económica, siendo ideal para soportar cualquier aplicación que se desarrolle hoy en día. Este controlador, de hecho, proporciona al sistema una capa de abstracción que facilita la creación de nuevos servicios de red y aplicaciones. En este trabajo se ha seleccionado el controlador OpenDayLight por su popularidad y sus características, tras analizar varios controladores de código abierto.

Paralelamente a este cambio de paradigma, los ataques orientados a Internet, y especialmente los ataques de denegación de servicio (Distributed Denial of Service, DDoS), siguen sucediéndose. Los ataques DDoS tratan de agotar los recursos del sistema consumiendo el ancho de banda. En este Trabajo de Fin de Grado, se han estudiado los diferentes tipos de ataques DDoS, centrándose posteriormente en uno de los más comunes, *flooding* sobre el protocolo HTTP.

Tomando en consideración estos aspectos, en este TFG se ha desarrollado un mecanismo de defensa proactiva, que rejuvenece las replicas periódicamente, independientemente del estado en que se encuentren, y reactiva, que actúa cuando se produce la detección de una amenaza, ante ataques DDOS sobre un controlador de SDN en un entorno de red simulado (concretamente, por Mininet). El escenario de trabajo propuesto supone un servidor web que se encuentra distribuido en distintos nodos (gracias al uso de SDN), de modo que ante un ataque DDoS tolera la indisponibilidad de ciertos nodos. De este modo, se pretende mostrar una idea del funcionamiento de redes SDN en un entorno real y su potencial para contrarrestar ataques DDoS asegurando la calidad de servicio. Por último, se han realizado pruebas experimentales para demostrar su funcionamiento ante diferentes escenarios de ataque. Los resultados muestran que la defensa propuesta proporciona una capa de seguridad adicional al sistema que es capaz de mitigar los ataques DDoS. El código desarrollado se ha liberado para su utilización y para garantizar la reproducibilidad de los resultados obtenidos.

Proactive and reactive defense against DDoS attacks in a simulated SDN environment

ABSTRACT

Software Defined Networks (SDN) have emerged as a new paradigm for communication networks. SDN decouple the control plane from data plane and separate hardware layer from software layer. SDN provides a flexible, manageable, adaptative, and economic network architecture, becoming an excellent choice for supporting complex network applications currently deployed by numerous telco companies. SDN features a central device (named controller) that manages the network in a central form. The controller provides an abstraction layer to facilitate the creation of new network services and applications. In this project, we selected OpenDayLight among other open-source SDN controllers OpenDayLight controller because of its popularity and its capabilities.

SDN is gaining in popularity during recent years, being adopted by well-known web-scale providers as Google, Amazon, Facebook and Microsoft or communication service providers as AT&T, CenturyLink, NTT, among others. Similarly, Distributed Denial-of-Service (DDoS) attacks show also an increase trend. DDoS attacks attempt to drain the system's resources to disrupt the normal operation of the system, thus leading to unavailability of services. In this work, various types of DDoS attacks were studied and HTTP flooding attacks were chosen as the most representative DDoS attack.

In this work, we propose a defense mechanism against a DDoS attack integrated in a SDN controller. Our mechanism performs a proactive and reactive defense in different time intervals. We consider a system that provides a web service using a SDN composed of different nodes (servers) that are replicated to guarantee a quality of service. A server can be up or down. Proactive defense rejuvenates nodes periodically, regardless of their state. Reactive defense acts only when a threat is detected, leading the node under attack to down state and its replica to up state. We have conducted experimentation on different scenarios to prove how the proposed defense mechanism ensures the quality of service while mitigating a DDoS attack. The source code of the defense mechanism, developed within OpenDayLight framework, is released for general use and to allow others to reproduce experiments.

Índice

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

En este primer capítulo se presentan los conceptos fundamentales que se trabajan a lo largo de la memoria. En primer lugar, se ponen en contexto las redes definidas por software (Software Defined Networking, SDN) así como los ataques de denegación de servicio distribuidos (*Distributed Denial of Service, DDoS*). Una vez mostrada el área de aplicación, se presentan las motivaciones de este proyecto y los objetivos a alcanzar. Finalmente, se describe brevemente la organización y contenido de la memoria para ayudar al lector a ubicar con claridad los capítulos.

1.1. Contexto

Este proyecto está relacionado con la nueva arquitectura de red en desarrollo desde hace tres años denominada *Software Defined Networks* (SDN) [FRZ13]. Esta arquitectura separa la capa de control y la capa de datos por medio de un elemento que actúa como núcleo de la red, denominado controlador. La separación entre las capas es necesaria ante el gran crecimiento de las redes de comunicaciones, ya que se debe operar de una manera dinámica ante los eventos que ocurran y esto no es posible con redes tradicionales donde el comportamiento de los dispositivos depende de su configuración previa.

En segundo lugar, se debe tener en cuenta el constante aumento de ataques DDoS [ZJT13] que se están produciendo cada vez con mayor frecuencia. Por ejemplo,se observó un aumento del 60 % entre 2009 y 2010 [Arb10]. La seguridad de todo sistema garantiza cuatro requisitos: autenticación, integridad, confidencialidad y disponibilidad del sistema. Un ataque de denegación de servicio distribuido es un intento de provocar la saturación o fallo de un servicio *online* enviando tráfico inservible desde múltiples orígenes. Los ataques DDoS presentan una gran amenaza para la disponibilidad de servicios críticos [Gar00], que se ven totalmente degradados a causa de estos ataques y además pueden ocasionar pérdidas de negocio u otras catástrofes [SSKS10].

1.2. Motivación

Se ha decidido utilizar SDN como arquitectura para este trabajo en primer lugar por la gran apuesta que está realizando la industria para adoptar esta tecnología en sus nuevas redes de comunicaciones [Cis11]. En segundo lugar, presenta beneficios teóricos y técnicos ya que requiere poca inversión inicial, proporcionando un ahorro de costes y una mayor agilidad de red.

Por otro lado, el creciente aumento de ataques DDoS para intentar desestabilizar e incluso minar la capacidad de respuesta de los servicios ofrecidos por las redes de comunicaciones llevan a las grandes corporaciones del sector a buscar nuevas técnicas o mecanismos para asegurar la demanda ante estos escenarios de ataque.

Estas necesidades presentan un contexto idóneo para pensar en la evolución de las redes de comunicaciones actuales aplicando un mecanismo de defensa sobre las mismas, siendo SDN una de las soluciones con mayor relevancia a día de hoy. SDN ya ha tenido casos de éxito real que prometen las previsiones teóricas [Goo12] o casos más concretos para grandes redes multimedia [Ver12].

1.3. Objetivos del proyecto

El objetivo de este proyecto es estudiar los ataques de denegación de servicio y diseñar un mecanismo de defensa contra ellos en una arquitectura de red sobre SDN. El proceso que se ha seguido es el siguiente:

- Estudiar qué es una SDN y cómo funciona.
- Estudiar los diferentes tipos de controladores para SDN.
- Estudiar los diferentes tipos de ataques DDoS y sus mecanismos de detección.
- Diseñar con UML e implementar en Java un mecanismo de defensa proactivo y reactivo que evite la amenaza.
- Experimentar la defensa en un entorno controlado.
- Evaluar el alcance de la solución y su viabilidad.

Este proceso ha dado como resultado un modelo teórico para el mecanismo de defensa proactivo y reactivo, implementado como prototipo sobre el controlador de SDN OpenDayLight [Ope16a] y el simulador de redes Mininet [Min15]. Un sistema reactivo mantiene una continua interacción con su entorno de tal forma que responde ante los estímulos externos en función de su estado. Por el contrario, un sistema proactivo es aquel que interacciona periódicamente independientemente de los eventos que sucedan. Con este prototipo se ha podido evaluar la viabilidad de la defensa propuesta en diferentes escenarios de ataque DDoS. Se han obtenido resultados satisfactorios de modo que se ha demostrado que el prototipo de mecanismo de defensa funciona correctamente.

1.4. Estructura del documento

El presente documento está dividido en dos partes: la memoria, donde se explica el desarrollo del trabajo de fin de grado; y los apéndices, donde se amplía la información de ciertos puntos relevantes.

El Capítulo 2 define los conocimientos previos necesarios para comprender este trabajo en su totalidad. El Capítulo 3 muestra los trabajos relacionados con este TFG. El Capítulo 4 explica las diferentes posibilidades sobre las que centrar dicho proyecto respecto a los tipos de ataques DDoS y sus mecanismos de defensa, así como de los controladores de SDN. El Capítulo 5 explica el mecanismo de defensa tanto a nivel conceptual como su implementación. El Capítulo 6 muestra y analiza los resultados obtenidos y finalmente, en el Capítulo 7 se exponen las conclusiones obtenidos con este TFG.

Respecto a los anexos, el Apéndice A es donde se hace balance del esfuerzo invertido en este TFG. Por último, el Anexo B detalla la configuración necesaria del sistema para poner en marcha los escenarios y poder realizar las pruebas. El código correspondiente a la implementación puede encontrarse en Bitbucket.

`https://bitbucket.org/Nessaji/sdnproreactdefense`

Capítulo 2

Conocimientos previos

En este capítulo se describe con mayor detalle el entorno sobre el que se desarrolla el TFG. En concreto: qué es una SDN; el protocolo OpenFlow; así como el software Open vSwitch y Mininet; qué es un ataque DDoS; el lenguaje de modelado de sistemas de software (*Unified Modeling Language*, UML);y finalmente el método de Montecarlo utilizado para realizar las pruebas.

2.1. Software Defined Network

Las redes definidas por software son un concepto que actualmente se encuentra en desarrollo y expansión. Sin embargo, el concepto en sí lleva años evolucionando hasta alcanzar el punto en el que se encuentra hoy en día [FRZ13, KREV+15].

Este tipo de redes trata de separar de una manera independiente el plano de control (*software*) del plano de datos (*hardware* que se encarga de la conmutación de los paquetes) tal y como se observa en la Figura 2.1, consiguiendo con esto redes programables, automatizadas y adaptables a las necesidades y problemas futuros. Se debe diferenciar que las redes tradicionales deterministas en las que el comportamiento de los dispositivos depende de su configuración previa van a evolucionar a una arquitectura de red dinámica con una interfaz de programación en la que un software gobierna su comportamiento.

Una de sus principales características es su gestión centralizada gracias al uso de un controlador que mantiene una visión global tanto de la red como del contenido de la misma, de modo que tiene la capacidad de modificar, eliminar o añadir flujos de datos según necesidades. Una de las ventajas es que se implementa bajo estándares abiertos de modo que no depende de protocolos propietarios o dispositivos de proveedores específicos. Además, dentro de la arquitectura SDN se puede programar directamente sobre la arquitectura de red utilizando módulos software instalados en el controlador, lo que permite agilizar los procesos de administración y configuración.

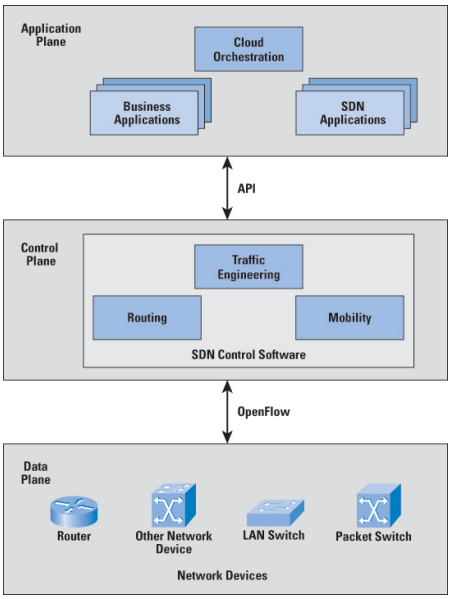


Figura 2.1: Arquitectura de capas en SDN (extraído de [Cis11]).

2.1.1. Arquitectura y elementos de SDN

Toda arquitectura de SDN [SNK12] se basa en un elemento central, el controlador, que se encarga de comunicar las capas de control y aplicación mediante el uso de dos APIs: la NorthBound API y la SouthBound API. *NorthBound* API es la encargada de interpretar las aplicaciones y establecer la comunicación con el controlador, mientras que la *SouthBound* API se encarga de comunicar el *hardware* con el controlador de manera transparente e independiente del elemento *hardware* que se encuentra debajo. Se puede observar en la Figura 2.2 cómo funcionan dichas API entre el plano de control y de aplicación, teniendo al controlador en un punto intermedio.

Así pues, los elementos más importantes son:

- **Controlador**: Como ya se ha descrito, el controlador [sdx13] es el elemento central y más importante de la arquitectura. Además de que se permite replicar el controlador, se pueden utilizar varios controladores para diferentes dominios de modo que hacen las redes SDN mucho más escalables y seguras [KRV13]. Uno de los puntos fuertes de la arquitectura SDN es que se pueden incluir nuevos módulos en el núcleo del controlador según las necesidades del sistema.

- ***SouthBound* API**: Se encarga de la comunicación entre el controlador y los elementos de la red. Permite hacer cambios dinámicos en todos los elementos de la red para adaptarse a las necesidades en tiempo real por parte de los usuarios. Existen varios tipos de soluciones a esta problemática pero el que está tomando mayor relevancia en la actualidad es OpenFlow [Ope15b].

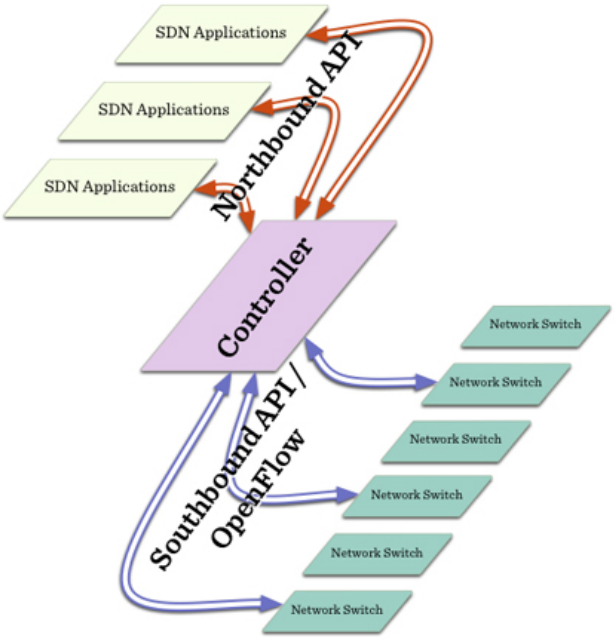


Figura 2.2: Elementos de SDN (extraído de [Net12]).

- ***NorthBound* API**: Se encarga de facilitar la comunicación entre el *core* del controlador con nuevas APIs o aplicaciones. Gracias a la presente arquitectura de SDN las aplicaciones pueden incluir nuevas funcionalidades de red más potentes como es el caso del mecanismo de defensa propuesto en este trabajo.

2.2. Protocolo OpenFlow

Según la definición dada por ONF [Ope15b], OpenFlow es la primera interfaz de comunicación definida entre la capa de transporte y la capa de control en una arquitectura SDN. Es decir, OpenFlow permite la manipulación y monitorización de los elementos del plano de control, como *switches* o *routers*.

La solución de OpenFlow [Ope15c] consiste en separar distintos tipos de tráfico dentro de *switches* y *routers* teniendo en cuenta sólo el transporte de datos. Una de sus características principales es la utilización de tablas de flujo como las que ya implementan los *switches* convencionales. Todos los tipos de *switches*, aún dependiendo del fabricante, suelen contar con tablas *Network Address Translation* (NAT), tablas del *firewall* o tablas QoS. OpenFlow pretende explotar las características comunes entre todas ellas.

El controlador y el *switch* se comunican entre sí a través del protocolo OpenFlow mediante un conjunto de mensajes predefinidos de modo que sea capaz de monitorizar y gestionar los paquetes recibidos, los paquetes enviados, modificar las tablas de

encaminamiento y la identificación de estados.

La ruta de datos de un *switch* OpenFlow está definida según una tabla de flujos que contiene algunos campos de la cabecera de los paquetes y una acción a realizar con dicho paquete (parecido a las reglas de un *firewall*). Si un *switch* recibe un paquete desconocido, para el cual no tiene entradas en su tabla de flujo coincidentes, reenvía el paquete al controlador para que éste tome la decisión sobre cómo gestionar el paquete. Esta funcionalidad es vital en este proyecto porque es una de las características más importantes para el desarrollo del mecanismo de defensa propuesto: el controlador es capaz de interrumpir los flujos de datos tras detectar un ataque de manera reactiva y reiniciar dichos flujos cada cierto tiempo de una manera proactiva.

Por lo tanto, OpenFlow permite desplegar una estrategia de reenvío de paquetes diferente a todo lo conocido e implementar protocolos de comunicación de red de forma centralizada y global. Por ejemplo, OpenFlow se está utilizando para aplicaciones tales como redes de nueva generación y redes de alta seguridad [Ben13].

Componentes de OpenFlow

OpenFlow utiliza principalmente dos componentes:

1. **Switch**: Es el encargado de procesar los paquetes de acuerdo a las reglas definidas previamente por el controlador. Estas reglas son instaladas en la propia tabla de flujo del *switch*.

2. **Controlador**: Elemento principal y más importante de una red SDN y OpenFlow, es capaz de evaluar el estado de toda la red y añadir o eliminar flujos de los *switches* OpenFlow, según las aplicaciones instaladas en el controlador. Está conectado directamente con todos los *switches* que dependen de él mediante una comunicación segura TCP bajo TLS. Un controlador puede ser una aplicación simple que tan sólo añade flujos de forma sistemática o bien una aplicación compleja que tiene instalados varios módulos que reaccionan de forma dinámica ante distintas situaciones de estado de la red, siendo un elemento transparente para el usuario final. Se pueden tener más de un controlador en una red SDN, y aunque esto crea una situación mucho más compleja para gestionar, es menos vulnerable ante un fallo en el controlador al eliminar el problema de punto único de fallo. Además, el controlador puede encontrarse en un dispositivo remoto diferente si es necesario.

Tablas de flujo

Una tabla de flujo está compuesta por un conjunto de entradas [Ope15a], siendo relevantes para este proyecto:

- ***Match Fields***: comprueba que el paquete coincide con esta entrada de flujo, comprobando el puerto de entrada del paquete así como las cabeceras del mismo.

- **Prioridad**: prelación de coincidencia con una entrada anterior.

- **Contador**: Existe un gran número de contadores posibles, los más relevantes son aquellos relacionados con el número de entradas y con los puertos.
- **Instrucciones**: modificación del conjunto de acciones a realizar con dicho paquete cuando el proceso de coincidencia es satisfactorio. Las instrucciones más importantes son aquellas que escriben o borran acciones para cada entrada de flujo.
- ***Timeouts***: existen dos tipos de timeouts definidos para cada entrada de flujo:
  1. ***IdleTimeOut***: tras la ausencia de paquetes coincidentes con dicho flujo en un intervalo de tiempo definido, el flujo es eliminado.
  2. ***HardTimeOut***: siempre que transcurra un intervalo de tiempo, el flujo será eliminado de la tabla a menos que el controlador envíe una actualización del intervalo temporal o modifique su comportamiento.
- ***Cookie***: dato seleccionado por el controlador que sirve para filtrar las estadísticas de flujos, eliminarlos o modificarlos.

2.3. Open vSwitch

Open vSwitch [Ope14] es un software con licencia de código abierto Apache 2.0. Este software fue desarrollado para ser utilizado como un *switch* virtual en entornos de servidores virtualizados, de modo que se encarga de reenviar el tráfico entre diferentes máquinas virtuales en el mismo equipo físico y a su vez entre las propias máquinas virtuales y la red física. Open vSwitch soporta múltiples tecnologías de virtualización basadas en Linux como VirtualBox o VMWare, y también se ha integrado en sistemas de gestión virtual como OpenStack [Ope16b].

Open vSwitch tiene principalmente dos componentes que se pueden diferenciar en la Figura 2.3:

- **Ovs-vswitchd**: Demonio[1] que implementa todas las funcionalidades del *switch*. Está formado por un módulo del *kernel* de Linux para la conmutación basada en flujos.
- **Ovsdb-server**: Base de datos ligera donde se almacenan los parámetros de configuración del *switch* y que éste consulta para obtener su configuración.

Los principales motivos por los se ha optado por utilizar el software Open vSwitch durante la realización de este proyecto son: i) su licencia de código abierto; ii) su compatibilidad completa con el protocolo OpenFlow en su versión 1.0 y posteriores; y iii) su perfecta integración con el software Mininet.

[1]Nomenclatura utilizada en GNU/Unix que hace referencia a un tipo especial de proceso informático no interactivo, es decir, que se ejecuta en segundo plano en vez de ser controlado directamente por el usuario.

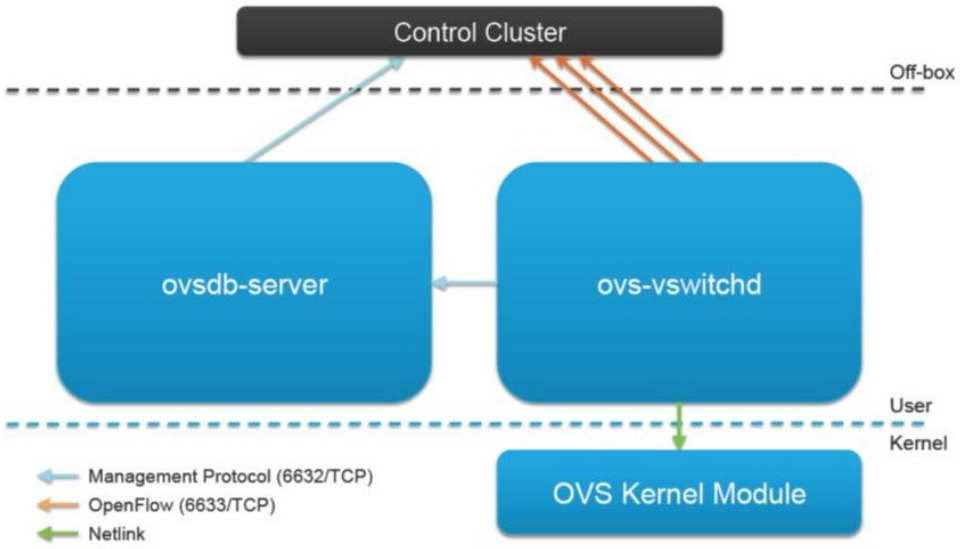


Figura 2.3: Componentes de Open vSwitch (extraído de [CSD14]).

2.4. Mininet

Mininet es un emulador de redes de código abierto capaz de crear una red de comunicaciones virtual compuesta por equipos, *switches*, controladores y enlaces. Los equipos de Mininet ejecutan el *kernel* de redes básico de Linux y los *switches* son capaces de ejecutar el protocolo OpenFlow consiguiendo un enrutamiento personalizado altamente flexible y escalable. Mininet así permite realizar una topología de red personalizada sin la necesidad de un potente *hardware*.

Su página web [Min15] facilita la familiarización con el programa ya que cuenta con extensos tutoriales y una gran cantidad de información de cómo funciona este *software*.

2.5. Ataques de denegación de servicio

Los ataques de denegación de servicio tienen como objetivo impedir la disponibilidad de un servicio (o activo) mediante el agotamiento de los recursos del sistema. Cuando son originados desde distintos orígenes reciben el nombre de ataques de denegación de servicio distribuidos (en inglés *Distributed Denial of Service attacks*, DDoS). Debido a su mayor capacidad de provocar daño, estos últimos son los más populares en la actualidad y a menudo usados por redes ajenas controladas por ciber-criminales [LJP+08]. Normalmente, los ataques DDoS logran sus objetivos mediante el envío masivo de información, intentando ocupar la mayor parte del ancho de banda de la red en la que se encuentra el objetivo del ataque. Esto limita sustancialmente el acceso a sus recursos provocando graves problemas de disponibilidad.

Estos ataques requieren pocos conocimientos para su ejecución y su éxito depende de los recursos del sistema atacado y de la cantidad de nodos desde donde se origina el ataque. Existen también otros tipos de ataques de denegación de servicio que no necesitan el envío masivo de información, sino que intentan inyectar paquetes de datos capaces de comprometer alguna vulnerabilidad de la víctima, como por ejemplo el popularmente conocido como "ping de la muerte". En [PLR07] se muestra en detalle estos casos y otros ejemplos.

2.6. El Lenguaje Unificado de Moledado (UML)

El Lenguaje Unificado de Modelado UML [RJB04] es un lenguaje estándar gráfico destinado al modelado de sistemas tanto de *hardware* como de software. UML actualmente es promovido por *Object Management Group* y es un estándar ISO. El lenguaje UML permite abstraerse del lenguaje para el que se desarrolla, proporcionando soporte para cada etapa de desarrollo a través del modelado completo de vida del sistema. El modelo es esencial en la definición y diseño del software que se quiere desarrollar, por ello se debe hacer hincapié en lograr un modelo congruente que facilite el posterior desarrollo.

UML define trece tipos distintos de diagramas que sirven para describir diferentes vistas de un modelo que necesita ser caracterizado, enfocado desde el paradigma Orientado a Objetos (OO). Para enumerarlos de una manera ordenada es mejor organizarlos por categorías:

1. **Diagramas de estructura**: clases, componentes, objetos, estructura compuesta, despliegue y paquetes.
2. **Diagramas de comportamiento**: actividades, casos de uso y estados.
3. **Diagramas de interacción**: secuencia, comunicación, tiempos y vista de interacción.

Aquellos que se han empleado en este proyecto y han sido más relevantes son el Diagrama de secuencia y el Diagrama de estados.

Diagrama de secuencia

Un diagrama de secuencia muestra las clases o componentes que forman parte del sistema así como las llamadas que se realizan en cada uno de ellos para llevar a cabo distintas tareas. Los diagramas de secuencia definen las acciones que se pueden realizar en la aplicación. Un diagrama de secuencia contiene líneas de vida, mensajes y objetos. Los mensajes se representan con líneas continuas con una punta de flecha en el extremo, los objetos se representan como rectángulos y el tiempo se representa con una progresión vertical; es decir, aquello que se encuentra más arriba en el diagrama sucede con anterioridad. Los mensajes pueden ser simples, síncronos o asíncronos. Un mensaje simple es aquel que produce una transferencia de control de un objeto a otro. Un mensaje

síncrono es cuando el objeto receptor se encuentra esperando el mensaje para continuar sus acciones. Finalmente, un mensaje asíncrono es aquel donde el objeto no espera la respuesta para continuar.

Diagrama de estados

Un diagrama de estados modela la vida de un objeto mediante una máquina de estados. Cada objeto cuenta con su propio diagrama de estados de modo que se puede referenciar desde un objeto a otro distinto. Este tipo de diagramas muestra el flujo de control entre estados; es decir, en qué estados posibles puede estar el objeto y cómo se producen los cambios entre dichos estados. Un diagrama de estados contiene estados, eventos y transiciones. Un estado es una situación en la vida de un objeto durante la cual satisface una condición, realiza alguna actividad o espera algún evento. Un evento es la representación de un acontecimiento significativo que ocurre en el tiempo. Una transición es una relación entre dos estados e indica que un objeto que esté en el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y se satisfagan unas condiciones impuestas en la transición.

2.7. Método de Montecarlo

El método de Montecarlo [SS77] es un método no determinista que permite resolver problemas mediante la simulación de variables aleatorias. En este proyecto se ha utilizado para realizar evaluaciones de los distintos escenarios configurados.

Capítulo 3

Trabajo relacionado

En este capítulo se presentan los trabajos relacionados con el TFG en categorías según su temática principal.

3.1. Vulnerabilidades SDN

En [EAJ+] y [KRV13] se presentan las vulnerabilidades que puede sufrir una arquitectura SDN así como algunos de los mecanismos que se pueden implementar a nivel de software para resolverlos. En [AL14] también se realiza un estudio de las vulnerabilidades de SDN y se propone un mecanismo de mitigación mediante técnicas de *machine learning*. En [LHK+14] se presenta una aplicación sobre un controlador SDN que permite la detección de ataques DDoS cuando se realizan desde redes de equipos infectados y cómo la arquitectura SDN facilita su mitigación. Por último, [RH15] presenta un *rootkit* que es capaz de instalarse en el sistema operativo de la arquitectura SDN sin ser detectado de tal forma que es capaz de espiar todas la comunicaciones que se realizan en la red. Además, está implementado para el controlador utilizado en este proyecto,OpenDayLight (véase Sección 4.1.2).

3.2. Mecanismos proactivos

En [PDBAA15] se presenta una aplicación para SDN que es capaz de establecer comunicaciones críticas fiables. Utiliza un mecanismo proactivo que sirve para resolver la problemática de un tráfico *multicast* robusto de modo que es capaz de soportar múltiple cantidad de fallos del sistema. Otro ejemplo de mecanismos proactivos es [HBPG15] donde se presenta un mecanismo proactivo aplicado a la gestión masiva de datos centralizada.

3.3. Mecanismos reactivos

Una parte importante de los trabajos previos relacionados con este TFG están estrechamente relacionados con la implementación de mecanismos de defensa reactivos utilizando distintas técnicas, como es el caso de [KKSG] donde se presenta un mecanismo de defensa para SDN en el que se desarrolla un *firewall* a nivel de aplicación. Este es capaz de actuar desde la capa física hasta la capa de aplicación ampliando las características de un *firewall* común. En [LWLP15] se presenta un mecanismo de defensa para evitar ataques DDoS (Distributed Denial of Service) similar a este proyecto. Sin embargo, tan sólo hacen uso de políticas reactivas para mitigar los ataques DDoS —aunque a cambio también se desarrolla un sistema de detección más sofisticado. En [MDC+14] se presenta un mecanismo de defensa reactivo para múltiples tipos de ataques DDoS. Otro caso de uso es [BMP10], donde se presenta un mecanismo de defensa reactivo capaz de detectar ataques *lightweight* DDoS utilizando redes neuronales con una gran tasa de éxito. En el artículo [SPY+13] se presenta un proyecto mucho más complejo que consiste en un *framework* modular utilizado para implementar componentes software de seguridad, añadiendo una nueva capa de abstracción sobre la capa de aplicación. Finalmente en [WZLH15] se presenta un mecanismo de defensa ante ataques DDoS en arquitecturas de *Cloud Computing* y SDN que realiza una implementación de detección de diferentes tipos de ataques y medidas de mitigación de modo que se puede configurar qué se desea utilizar.

3.4. Mecanismos proactivos y reactivos

Apenas se encuentran desarrollados proyectos con mecanismos proactivos y reactivos a la vez, aunque es posible encontrar algunos muy completos como la tesis doctoral [Chu15] donde se muestra cómo la arquitectura SDN facilita la defensa ante vulnerabilidades que sufre *Cloud Computing* aplicando un mecanismo de defensa proactivo y reactivo ante multitud de amenazas. También se debe destacar [LBZ+14], el cual desarrolla una solución llamada DrawBridge que consiste en capacitar a los equipos finales con capacidad para mejorar la ingeniería de tráfico cuando sufren ataques DDoS, de modo que funciona de manera reactiva y proactiva. Finalmente en el artículo [SBC+10] se presenta un sistema proactivo y reactivo basado en un modelo distribuido híbrido para desarrollar un sistema replicado tolerante a intrusiones aunque no aplicado a SDN. En este caso se presenta un mecanismo capaz de garantizar un mínimo de réplicas disponibles asegurando así el correcto funcionamiento del sistema.

El sistema de defensa propuesto en este proyecto aporta un mecanismo proactivo y reactivo para mitigar ataques DDoS siendo capaz de gestionar y modificar la arquitectura de red gracias a SDN en función de las necesidades del servicio.

Capítulo 4

Caracterización del escenario

En este capítulo se muestran los diferentes tipos de controladores de SDN así como el motivo de por qué se ha escogido OpenDayLight, los tipos de ataques DDoS y sus mecanismos de mitigación.

4.1. Controladores SDN

Uno de los elementos principales de la arquitectura SDN es el controlador, núcleo de la arquitectura del cual depende cómo se comporta la red. Actualmente existen varias alternativas diferentes de controladores según si son controladores comerciales, o bien controladores de código abierto. Los más interesantes y relevantes son los controladores de código abierto, ya que sobre ellos se puede desarrollar libremente y cuentan con un gran apoyo de la comunidad. Algunos controladores comerciales también están basados en código abierto, con lo que elegir cuál es el controlador más adecuado para este trabajo ha sido una tarea muy importante y determinante en el resultado final.

Para este proyecto se necesita un controlador de código abierto que permita el desarrollo de aplicaciones sin coste alguno, se encuentre en fases estables de desarrollo y además cuenten con una comunidad que de apoyo en la resolución de problemas. Sin embargo, es necesario realizar un análisis de los diferentes controladores comerciales y de código abierto más importantes para evaluar sus capacidades, fortalezas y debilidades.

4.1.1. Controladores comerciales

Aunque este tipo de controladores se encuentran en desarrollo en la actualidad, al igual que los controladores de código abierto, ya se pueden encontrar algunas soluciones finales ofertadas. Surgen como necesidad de un desarrollo privado frente a los controladores abiertos. Empresas del sector tecnológico como Cisco, NEC o IBM, por ejemplo, son miembros importantes de algunos controladores de código abierto, que también desarrollan sus propias soluciones comerciales. A continuación se describen brevemente algunas de estas distribuciones comerciales.

- **Application Policy Infrastructure Controller** [Cis15]. Diseñado por Cisco, es el controlador para la alternativa de OpenFlow llamada OpFlex. Proporciona una API central, un repositorio de políticas y un repositorio central de datos globales. A diferencia de los controladores SDN OpenFlow, APIC funciona de manera diferente porque no se encarga de encaminar cada flujo de datos, sino que según el estado de la red, aplica distintas políticas en los elementos de red y son estos los que se encargan de tomar las decisiones en función de las políticas establecidas.

- **Virtual Application Networks (VAN)** [HP15]. Controlador desarrollado por HP sobre OpenFlow, su funcionamiento interno está orientado a incluir seguridad en las comunicaciones del controlador con el resto de elementos de la red. Además, realiza una administración centralizada, tiene capacidad de automatización de actividades y otras capacidades inherentes de redes SDN. El esquema que presenta VAN no es muy diferente al de otros controladores de código abierto como OpenDayLight.

- **IBM Software Defined Network for Virtual Environments**. Diseñado por IBM a principios del año 2014. Ofertado como la solución de virtualización de red añadiendo flexibilidad y adaptabilidad a la demanda del cliente actual, está orientado a proporcionar seguridad a los datos y servicios ofrecidos dentro de la red. Cuenta con una versión adaptada para OpenFlow pero también funciona independientemente utilizando una tecnología propia denominada *Virtual Environment.*

4.1.2. Controladores de código abierto

Entre los controladores de código abierto existen multitud de alternativas (véase la Tabla 4.1), algunas todavía en fase de desarrollo y otras ya son una referencia en el mercado. En esta sección se describen algunas de ellas según los datos aportados en [KZMB14].

- **POX** [NOX15]. POX es una evolución del controlador NOX, pero desarrollado sobre Pyhton. POX surgió como una introducción a las SDN de modo que aquellos usuarios que quieran comenzar con SDN encuentren un entorno de desarrollo sencillo y manejable. Cuenta con dos métodos de desarrollo, en un primer lugar una API basada en Python y en segundo lugar una API en web que utiliza JSON-RPC. Como muchos otros controladores, cuenta además con una interfaz gráfica en web desde la cual se puede monitorizar la red y controlar otras operaciones. Sin embargo, tiene algunos aspectos negativos como que no cuenta con demasiada información para comenzar desde cero, ni manuales disponibles siendo algo bastante negativo a la hora desarrollar sobre este controlador.

- **Floodlight** [Pro15]. Floodlight fue la evolución del controlador Beacon. Está desarrollado sobre Java y también posee una interfaz web. Soporta el uso de API REST y tiene una comunidad de usuarios bastante grande. La comunidad actúa de manera muy activa en las listas de correo, aspecto muy importante a tener en cuenta.

	Pox	FloodLight	OpenDayLight
Interfaces	SB (OpenFlow)	SB(OpenFlow) NB (Java & REST)	SB(OpenFlow & Others SB Protocolos) NB,(Java & REST)
Virtualización	Mininet & Openv Switch	Mininet & Openv Switch	Mininet & Openv Switch
GUI	Sí	Web UI	Sí
REST API	No	Sí	Sí
Productividad	Medio	Medio	Alto
Código Abierto	Sí	Sí	Sí
Documentación	Escasa	Media	Media
Lenguaje Programación	Python	Java + cualquier lenguaje que utilice REST	Java
Modularidad	Media	Alta	Alta
S.O. Soportado	Linux, Mac Os and Windows	Linux, Mac Os and Windows	Linux
Edad	3 años	4 años	2 años
Soporte OpenFlow	OF v1.0	OF v1.3	OF v1.3
OpenStack Networking	No	Medio	Medio

Tabla 4.1: Tabla comparativa controladores.

Pese a todo esto, presenta algunos inconvenientes como la dependencia de desarrollo de API REST. A lo largo de su tiempo ha ido perdiendo importancia y actualmente apenas es relevante frente a OpenDayLight. En un primer momento se decidió utilizar este controlador para este trabajo, pero surgieron bastantes problemas respecto a la configuración y puesta en marcha porque junto con Mininet hace uso de máquinas virtuales para simular la arquitectura SDN, lo que conllevaba una gran cantidad de carga para el sistema y numerosos conflictos de interconexión entre diferentes máquinas debido a las interfaces virtuales de red.

Para elegir el controlador que se ha utilizado durante la realización de este proyecto se tuvieron en cuenta varios aspectos, desde la comunidad que se encontraba detrás de cada uno, el lenguaje y APIs que utilizaba cada uno, así como su eficiencia y relevancia en el mundo empresarial. Finalmente tras intentar utilizar Floodlight se ha optado por OpenDayLight el cual se explica a continuación.

- **OpenDayLight**.OpenDayLight [Ope16a] es el controlador elegido para este proyecto. A modo de resumen se enumeran a continuación sus principales fortalezas:

  - Apoyo de la industria con más de 50 miembros implicados, algunos de ellos de gran importancia como Cisco, Intel o NEC.
  - Proyecto de código abierto y bajo el amparo de Linux Foundation.
  - Desarrollo completo en Java, con posibilidades de desarrollo de aplicaciones a través de varias tecnologías como son API REST o Document Object Model (DOM).
  - Documentación extensa y detallada con una comunidad amplia y muy activa.
  - Proyecto en continuo desarrollo.

4.1.3. Arquitectura OpenDayLight

En la Figura 4.1 se encuentra la arquitectura del controlador en su segunda versión denominada Helium utilizada en este proyecto. Se pueden distinguir tres capas en esta arquitectura:

- **Aplicaciones de red e instrumentación**: En la capa superior se encuentran las aplicaciones que se encargan del control y monitorización de la red. Además en esta capa se pueden encontrar soluciones que hagan uso de diferentes tecnologías como es el caso de computación en la nube o servicios de virtualización de red. En esta capa se encuentran aquellas aplicaciones que computan la ingeniería de tráficos de la red, así como las aplicaciones de seguridad.

- **Controlador**: Se encuentra en la capa central y es donde se manifiestan las abstracciones que proporciona SDN. El controlador cuenta con una serie de módulos implementados que permiten a las aplicaciones de la capa superior obtener datos e información sobre el estado de la red, tal y como se hace en el mecanismo de

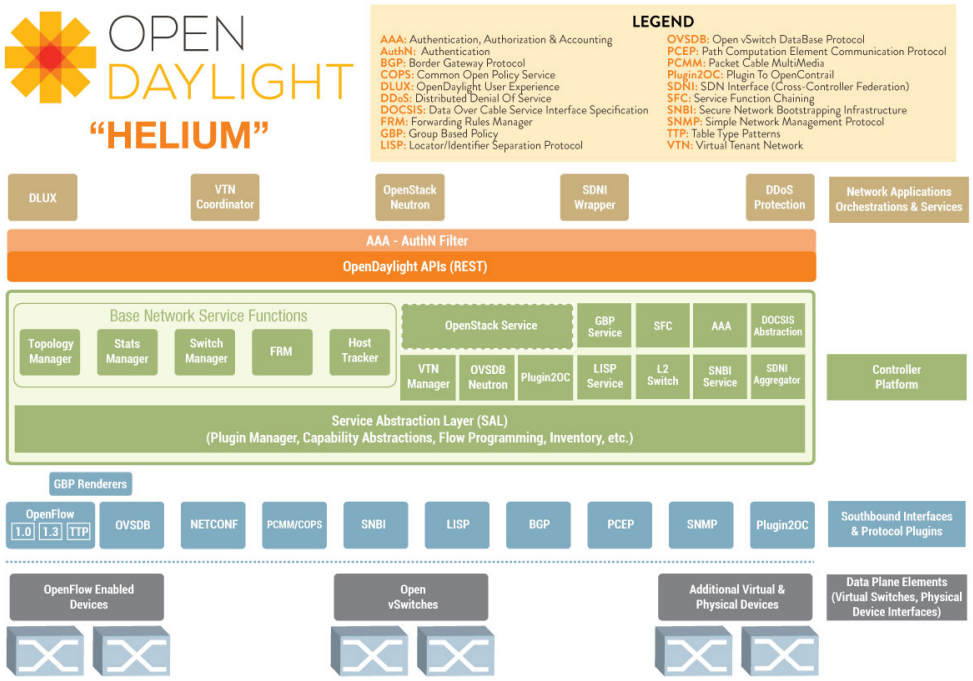


Figura 4.1: Arquitectura OpenDayLight (extraído de [Ope16a]).

defensa propuesto. Además cuenta con una serie de APIs que permiten el desarrollo de las aplicaciones superiores (DOM API, REST API o JAVA), mientras se implementan varios protocolos que permiten la comunicación con los elementos de red inferiores (OpenFlow, Path Computation Element Protocol, Simple Network Management Protocol).

- **Elementos de red**: La capa inferior está compuesta por aquellos elementos de red que son programables mediante los protocolos implementados por el controlador. Gracias a la capa de abstracción del controlador, se pretende que todos los elementos sean compatibles con el controlador OpenDayLight independientemente del fabricante o de si son elementos virtuales o físicos.

El controlador de OpenDayLight está implementado por *software*, haciendo uso de su propia máquina virtual de Java. El desarrollo de aplicaciones para OpenDayLight puede realizarse por diferentes vías gracias a la NorthBound API. Por un lado el controlador soporta el desarrollo mediante el uso del *framework* Open Services Gateway Initiative (OSGi) 14, pero también soporta la comunicación bidireccional usando la REST API. El uso de la API vía web permite ejecutar aplicaciones que no se encuentran en el mismo dominio de direcciones que el controlador, mientras que OSGi se usa para aquellas aplicaciones que sí están en el dominio del controlador como es el caso del mecanismo de defensa propuesto en este trabajo. El controlador es completamente independiente de los

elementos de red y protocolos inferiores gracias al uso de la capa de Service Abstraction Layer (SAL), que se encarga de exponer y trasladar las peticiones de las aplicaciones de capas superiores hacia abajo, transformando estas peticiones según el protocolo de comunicación hacia los dispositivos de red.

4.2. Ataques de denegación de servicio: Clasificación

A continuación se muestra una clasificación de los ataques de denegación de servicio. Únicamente han sido considerados los ataques que, por similitud o por impacto, pueden llegar a ser relevantes en una arquitectura SDN. El resto quedan fuera del alcance de este proyecto. En base a estos criterios se han establecido tres conjuntos de ataque: inundación (que finalmente serán los que se prueben con mayor detenimiento en la fase de evaluación), reflexión y ampliación. El éxito de un ataque puede depender tanto de su capacidad de amplificación, como de inundación. A continuación se describe cada uno de ellos y se motiva por qué se ha elegido estudiar con mayor profundidad el ataque por inundación.

4.2.1. Denegación de servicio basada en inundación

La denegación de servicio basada en inundación trata de alcanzar sus objetivos por inyección de grandes volúmenes de tráfico. Ha sido uno de los mayores temas de interés en la actualidad debido a su sencillez de ejecución y la magnitud de su impacto. En [WCXJ13] se pueden ver las diferentes estrategias para conseguir una inundación eficaz, como inundaciones de tasa alta y baja. Las primeras consisten en la emisión de grandes cantidades de tráfico de manera constante y uniforme, o bien de manera evolutiva que son aquellas en las que se centra este proyecto.

Cuando los ataques de inundación actúan en la capa de red, explotan funcionalidades propias de sus protocolos, siendo TCP, UDP, ICMP y DNS los más explotados. En [DM04] son descritas algunas de sus variantes, siendo la más popular de ellas la inundación SYN, tomada como objeto de estudio para el mecanismo de defensa desarrollado en este proyecto. Este ataque explota el protocolo TCP y su negociación del inicio de sesión en tres pasos. Para lograrlo el atacante envía paquetes SYN con direcciones IP inexistentes o en desuso y cuando el servidor ubica la petición en la memoria, espera a la confirmación del cliente. Mientras espera, dicha petición sigue almacenada en la pila de la memoria. Como estas direcciones IP no son válidas, el servidor nunca recibe la confirmación. De esta manera, el ataque explota el hecho de que cada una de las conexiones abiertas o iniciadas ocupa un espacio de memoria y que se mantiene en ella hasta un cierto intervalo temporal. Con la pila llena, el servidor legítimo no puede tramitar nuevas peticiones, denegando el acceso a nuevos usuarios e interrumpiendo el servicio. Por otro lado, la capa de aplicación ofrece nuevas posibilidades a los atacantes.

En [ZJT13] se trata este tema en mayor profundidad y se distinguen tres conjuntos de amenazas: las que se basan en inicios de sesión, envíos de peticiones y en respuesta lenta del servidor. Muy similar a la inundación SYN, el primer grupo trata de colapsar las colas

que permiten el acceso de usuarios a los servicios web. En segundo lugar, la inundación por peticiones consiste en el envío masivo de solicitudes web HTTP (GET/POST) que el servidor tiene que atender. En último lugar, los ataques de respuesta lenta se basan en intentar mantener las conexiones HTTP el mayor tiempo posible enviando datos lentamente, o bien procesando las respuestas con lentitud.

4.2.2. Denegación de servicio basada en reflexión

La inundación basada en reflexión surge de la necesidad de los atacantes de ocultar su origen de la intrusión. Este tipo de ataques, llamados ataques de denegación de servicio distribuida y reflejada (del inglés *Distributed Reflection Denial of Service*, DRDoS), tratan de aprovechar vulnerabilidades en terceras partes para forzarlas a emitir el tráfico malicioso.

Un ejemplo de ataque de reflexión se encuentra en los conocidos ataques de *smurfing* [Kum07]. Los ataques *smurf* son una variante de la inundación SYN que aprovecha elementos intermedios de la red para enmascarar su origen. En su ejecución, el origen de los paquetes es reemplazado por la dirección de la víctima. De esta forma consiguen que todas las máquinas intermedias respondan a la víctima tras recibir su solicitud.

4.2.3. Denegación de servicio basado en amplificación

La inundación basada en amplificación consiste en realizar peticiones a terceras partes con el objetivo de que las respuestas sean de mayor tamaño que el de las propias peticiones. Estas direcciones falsifican su dirección de retorno de manera que las respuestas, en lugar de llegar al atacante, llegan a la víctima.

Uno de los elementos de red más aprovechados para lograr la amplificación son los servidores DNS. Esta vulnerabilidad, denominada amplificación DNS, se discute en detalle en [AKK+13] donde se identifica como principal causante al hecho de que las consultas realizadas al servidor se realizan con datagramas con menor cantidad de información que las respuestas. Se puede amplificar con otros tipos de servidores como es el caso de [Nol13].

4.3. Mecanismos de defensa ante DDoS

Las distintas técnicas de defensa frente a intentos de denegación de servicio son clasificadas según el momento del proceso de intrusión en que actúan. Por lo tanto se pueden agrupar en cuatro tipos de mecanismos: prevención, detección, identificación del origen y mitigación. A continuación se resumen brevemente.

4.3.1. Prevención

Los mecanismos de prevención son aquellos que actúan antes de que el ataque suceda e independientemente de su detección. Su objetivo es minimizar el daño causado

por los atacantes. En este proyecto la parte proactiva actúa como prevención porque periódicamente se reinician los nodos de la red aunque no hayan sufrido un ataque.

4.3.2. Detección

La detección de los ataques es vital para que actúen el resto de los componentes defensivos. Su eficacia se basa en la proporción de ataques reales que son capaces de detectar sin equivocarse. Sin embargo, el hecho de que se presente un alto índice de acierto no implica que sean de buena calidad. También deben afrontar otros problemas, como los falsos positivos, la capacidad para procesar en tiempo real y la distinción de ciertos fenómenos en la red, tales como el conocido "Efecto Menéame". Este último son acumulaciones inesperadas de accesos al sistema de forma legítima por usuarios correctos que habitualmente acarrean errores de detección [ZJW+14].

En la identificación de DDoS son considerados dos paradigmas de los sistemas de detección de intrusiones: reconocimiento de firmas y anomalías. El reconocimiento de firmas se basa en la identificación de patrones de ataques previamente conocidos [TYZM14, SY13, YM05]. Este proyecto no se centra en la detección de patrones, sin embargo, se ha implementado un sistema simple de detección de patrones para realizar las pruebas sobre el mecanismo de defensa, dejando abierto una posible mejora en el módulo de detección. La mayor parte de la comunidad investigadora ha optado por el desarrollo de sistemas basados en el estudio de anomalías porque el sistema de detección de patrones dificulta la detección de nuevas amenazas. El reconocimiento de anomalías implican el modelado del comportamiento habitual y legítimo de un sistema, con el fin de identificar eventos que difieran de las acciones legítimas. De este tipo de detecciones han sido propuestas diferentes técnicas, tales como modelos probabilistas basados en Markov [SNK12], teoría del caos [CMW13], lógica difusa [KS13] o estudio de las variaciones de la entropía [ÖB15].

4.3.3. Identificación del origen

En la etapa de identificación del origen, la víctima trata de desenmascarar la ruta del vector de ataque con el fin de señalar a su autor. Este proceso a menudo es muy complicado, ya que el atacante dispone de diferentes métodos para ocultar su rastro. Estos varían desde sencillos procesos de suplantación de identidad, hasta atravesar tramos de redes anónimas. Por lo tanto, llegar hasta el extremo final es una situación idílica que apenas se consigue. Para realizar las pruebas de este proyecto se ha optado por no ocultar el origen de los equipos que atacan, ya que el mecanismo propuesto actúa sobre los flujos de datos directamente. Se debe tener en cuenta que aproximar la ubicación permite la realización de un despliegue defensivo mucho más eficaz ya que no se permite el tráfico posterior (incluso legítimo) de un atacante [KBP14].

4.3.4. Mitigación

Una vez detectada una amenaza debe procederse a su mitigación. En el caso de la denegación de servicio, la mitigación consiste en el despliegue de una serie de medidas que reduzcan el daño causado y, a ser posible, restauren los servicios del sistema comprometido. Habitualmente consisten en el incremento de la reserva de recursos disponibles para la supresión de cuellos de botella [KVF+12] y la actualización de las listas de acceso y políticas de cifrado.

El mecanismo propuesto en este proyecto es capaz de incrementar la reserva de recursos disponibles en función de las necesidades de la red. La arquitectura SDN permite desactivar aquellos nodos que se encuentran comprometidos y conectar un nuevo nodo (*backup*) en la posición que se encontraba el nodo comprometido. Junto a esto se cuenta con un mecanismo reactivo que es capaz de eliminar los flujos de datos dañinos en tiempo real, así como bloquear la comunicación con los nodos en los que se detecta un proceso de ataque hacia el sistema.

Capítulo 5

Arquitectura del Sistema

En primer lugar, en este capítulo se explica cuál es el despliegue de la arquitectura propuesta donde se implementa el mecanismo de defensa. A continuación, se justifica el modelo implementado en el mecanismo de defensa y se explica cómo funciona tanto conceptualmente (a nivel de diagramas de secuencia y máquinas de estados de los componentes), como a nivel de implementación.

5.1. Despliegue

El despliegue del sistema se muestra en la Figura 5.1. En un equipo se ha instalado el sistema operativo Ubuntu 14.04 en su versión más estable y sobre éste se ha instalado Mininet (véase Sección 2.4), y el software del controlador OpenDayLight (véase Sección 4.1.2) en una versión estable para el desarrollo (versión base 0.2.2). El controlador de OpenDayLight se conecta con Mininet mediante el acceso remoto por la interfaz *localhost* a través de un túnel SSH.

Los ataques DDoS se ejecutan desde los propios equipos emulados por Mininet de modo que en la topología de Mininet cuenta con un cierto número de nodos que actúan como servidores y un número concreto de equipos que actúan como clientes y atacantes según las pruebas que se realizan. Así los equipos clientes actúan como *Traffic generator* y los atacantes como *Attack DDoS*.

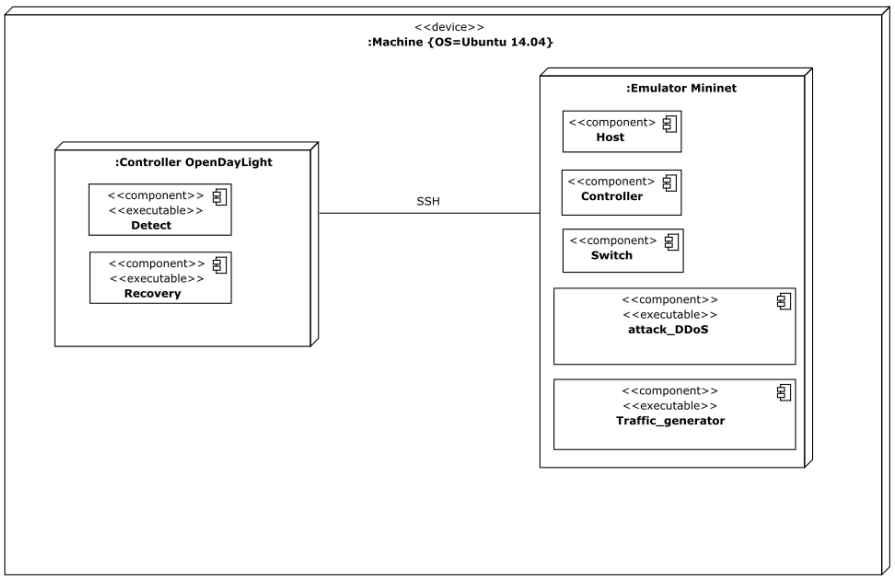


Figura 5.1: Diagrama de despliegue.

5.2. Explicación formal

El modelo del sistema que se puede observar en la Figura 5.2 funciona como un clúster, es decir, se cuenta con un número n de nodos, el cual se puede definir según las necesidades y capacidad del sistema. A su vez, cada nodo se encuentra replicado m veces de modo que estas réplicas actúan como posibles *backups* de cada nodo. Si se produce un ataque sobre uno de los nodos, se colocará su réplica posterior, eliminando la réplica afectada y creando una nueva réplica *backup*.

Dado que el número de nodos lo puede definir el administrador del sistema y éste viene condicionado según los recursos *hardware* del equipo, también se deben definir el número de réplicas asociadas a cada nodo. De este modo, si el numero de nodos es muy pequeño el número de réplicas para cada nodo debería ser mayor para lograr una mayor estabilidad del sistema ante una gran cantidad de ataques. Si por el contrario se cuenta con muchos nodos, se puede permitir que el número de réplicas sea más limitado e inferior. El objetivo de este sistema es garantizar que en todo momento existen el suficiente número de nodos disponibles capaces de asegurar la calidad del servicio ofrecido por el sistema.

De este modo se debe asegurar que la relación entre m y n sea $m \cdot n = i$, siendo i el número total de equipos que actúan como servidor y que el sistema es capaz de simular simultáneamente en ejecución.

Cuando uno de los nodos se ve afectado por un ataque y por lo tanto se debe colocar una nueva réplica en su posición se debe tener en cuenta el tiempo que transcurre realizando dicha recuperación del sistema. De este modo se definen dos intervalos temporales (véase Figura 5.3) de modo que en el intervalo reactivo T_R se divide en z slots temporales en el que en cada uno se realiza la detección y se pueden recuperar k réplicas

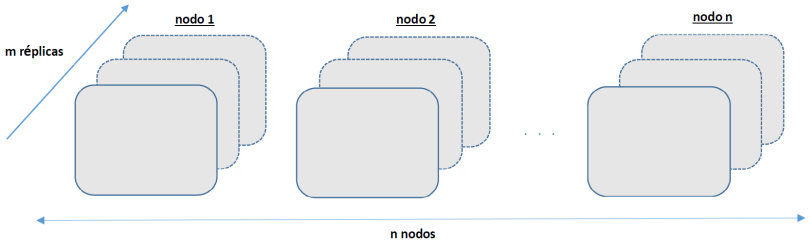


Figura 5.2: Diagrama del sistema.

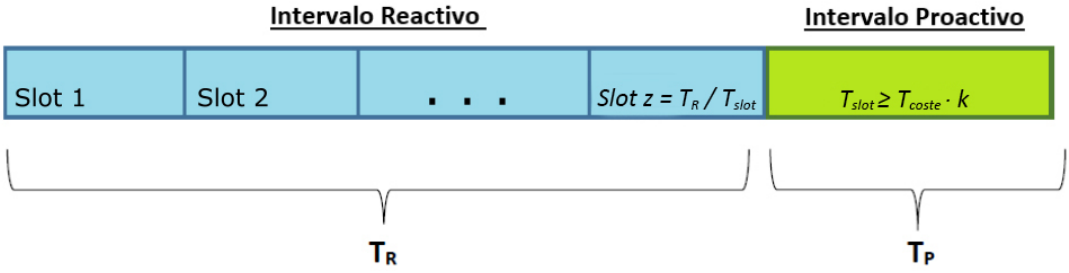


Figura 5.3: Relación entre el intervalo temporal reactivo y proactivo.

simultáneamente. La duración de dichos slots viene definida según T_{coste} y con la relación $T_{slot} \geqslant T_{coste} \cdot k$.

El intervalo reactivo T_R se define en la configuración del sistema y se puede modificar en tiempo real. El número de slots en los que se divide el intervalo reactivo depende de su duración definida en su configuración y el tiempo de slot T_{slot} de modo se cumple la relación $z = T_R/T_{slot}$.

El tiempo de duración del intervalo proactivo T_P viene definido según T_{slot}. Durante el intervalo proactivo se deben realizar rejuvenecimientos sobre las réplicas de un número concreto de nodos k. Aquellos nodos que van a recibir un rejuvenecimiento no dependen de los hechos acontecidos en el intervalo reactivo y se ejecutará de manera secuencial entre la lista de nodos n cada vez que se acceda al intervalo proactivo, garantizando así un rejuvenecimiento periódico de todos nodos al cabo de cierto número de intervalos completos.

5.3. Funcionamiento

En este mecanismo de defensa se destacan tres elementos en su arquitectura: el controlador, los *switch* y los equipos. Para cada uno de ellos se ha modelado una máquina

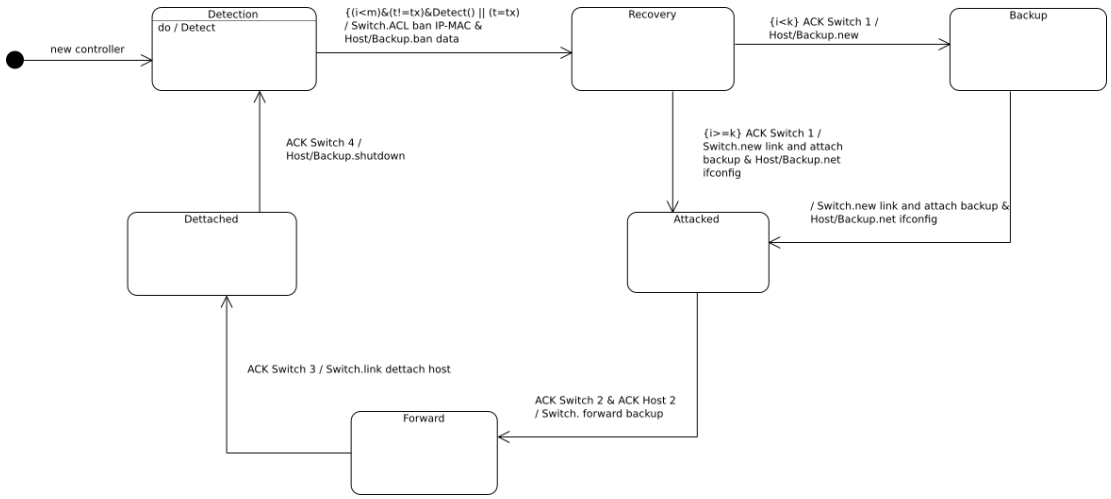


Figura 5.4: Diagrama de máquina de estados del Controlador.

de estados, para observar su comportamiento y todas las etapas de su funcionamiento.

Como se puede observar en la Figura 5.4, en primer lugar el controlador se inicia al comienzo de la ejecución de OpenDayLight, desde el primer momento el controlador se encuentra en el estado *Detection* en el cual se encuentra ejecutando constantemente la función *Detect*. Para realizar la transición al estado *Recovery* debe suceder una de las situaciones indicadas en el arco de la transición, que pase del estado reactivo al estado proactivo ($t = tx$), o bien si se encuentra en el intervalo temporal donde actúa de manera reactiva, se produce una detección de ataque y si dispone de suficientes nodos para continuar con el servicio ($i < m$)&($t! = tx$)&$Detect()$), pasa al estado *Recovery*.

En el estado *Recovery* lo primero que se hace es bloquear el flujo de datos del equipo atacante, así como borrar todos los flujos relacionados con dicho equipo en todos los nodos. A continuación si no se dispone de un número suficiente de réplicas ($i < k$) se emula un nuevo equipo que se añadirá a la reserva de réplicas pasando al estado *Backup*; sino se pasa al estado *Attacked* donde se realiza una nueva conexión y se configura la réplica que ahora actuará como nodo. Posteriormente se balancea el tráfico que estaba sirviendo el nodo atacado en el estado *Forward*, de modo que se reparte equitativamente entre el resto de nodos servidores para evitar pérdidas de tráfico y que el usuario legítimo no se vea afectado por una interrupción del servicio. A continuación se desconecta el nodo afectado por el ataque DDoS pasando al estado *Dettached* para finalmente desconectarlo y volver al estado de *Detection*.

En el caso de los equipos su funcionamiento se puede observar en la Figura 5.6. En primer lugar se produce el evento de creación de un nuevo equipo el cual puede ser invocado por el controlador cuando necesita un mayor número de réplicas o bien por el propio sistema al iniciarse. Para pasar al estado de *Connected* es necesario que se produzca el evento *net ifconfig backup*, que se produce cada vez que se quiere conectar una nueva réplica como nodo de trabajo. En el estado *Connected* el equipo realiza constantemente

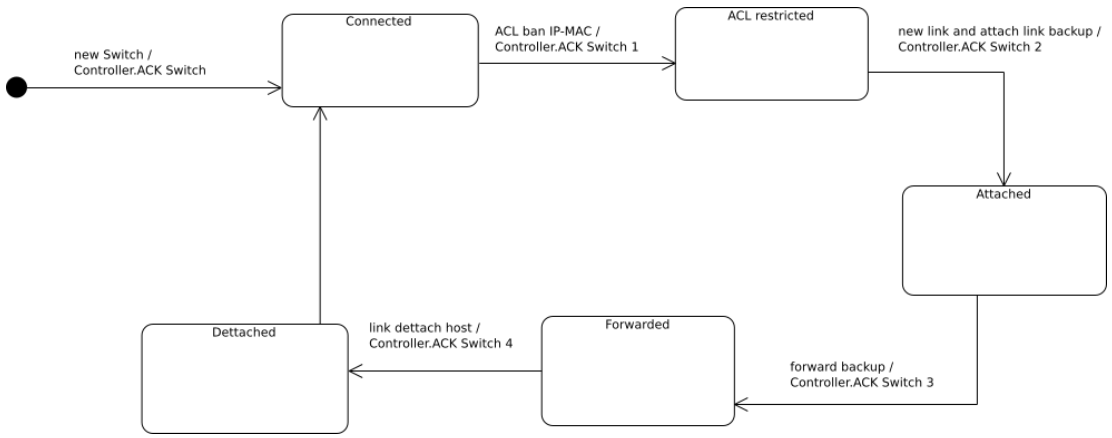


Figura 5.5: Diagrama de máquina de estados del Switch.

la tarea *Work* en la que procesa los flujos de paquetes entrantes. El equipo se mantiene trabajando hasta que el controlador avise de un supuesto ataque DDoS que es entonces cuando el equipo pasa al estado *Standby* y se produce la eliminación de sus flujos de datos(ban data). Finalmente el equipo se apaga cuando el controlador lo indica.

En la Figura 5.5 se observa la máquina de estados correspondiente al *switch*. Los *switch* se encienden todos al comienzo de la ejecución de Mininet pasando al estado de *Connected*. En dicho estado se mantendrá hasta que reciba un mensaje de prohibir una dirección IP y MAC concreta(ACL ban IP-MAC), en ese instante pasa al estado ACL *restricted*. A continuación realiza la creación de un nuevo enlace y su unión con la réplica(new link and attach link backup) pasando al estado *Attached*. Posteriormente pasa al estado *Forwarded* cuando el controlador le notifica que debe comenzar a redirigir tráfico a la nueva réplica(forward backup). Finalmente pasa por el estado Dettached cuando el controlador le indica que debe eliminar el enlace asociado con el nodo atacado(link dettach host) y vuelve al estado inicial *Connected*.

Gracias a los diagramas de secuencia se puede explicar el funcionamiento del sistema desde un punto más abstracto, de modo que se puede observar cómo funciona el sistema y el mecanismo de recuperación además, de cómo se realizan los pasos de mensajes entre los diferentes componentes del sistema, de modo que se relaciona directamente con los diagramas de máquina de estados de cada componente.

En el diagrama del sistema (véase la Figura 5.7), se puede observar como se inicia la ejecución del sistema con la creación de los distintos componentes (*switch*, nodos y controlador). Justo en la parte inferior se puede ver la diferencia entre la parte del sistema que actúa de manera reactiva, que sería la primera parte con el *loop* de modo que se encuentra en una perspectiva reactiva durante un intervalo de tiempo. Alternativamente, debajo se tiene la sección *alt* que se produce cuando el contador de tiempo llega a la marca temporal, en este momento el sistema actúa de manera proactiva aplicando el mecanismo de *Recovery* para los nodos seleccionados.

En el diagrama de la función Recovery de la Figura 5.8 se puede observar cómo se

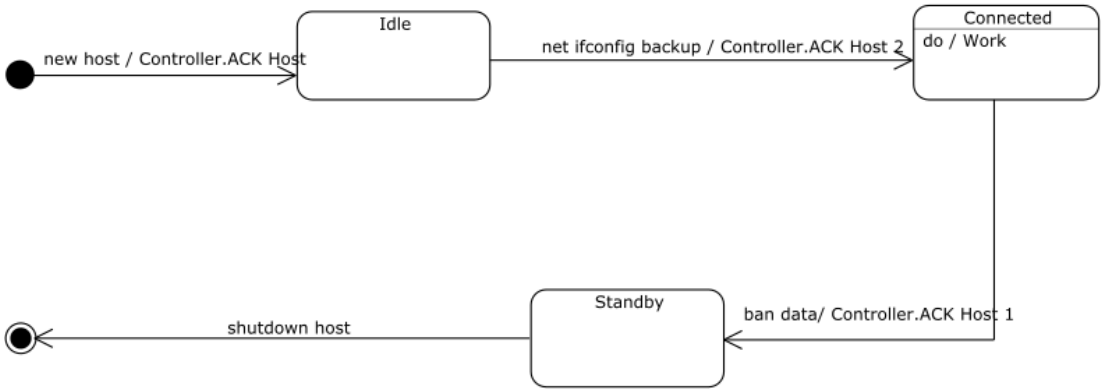


Figura 5.6: Diagrama de máquina de estados de los equipos.

produce el paso de mensajes entre las instancias de los componentes del sistema. Las transiciones se encuentran numeradas en orden del 1 al 9 y la sección *alt* es una excepción que tan solo se produce cuando no se encuentran disponibles suficientes réplicas activas.

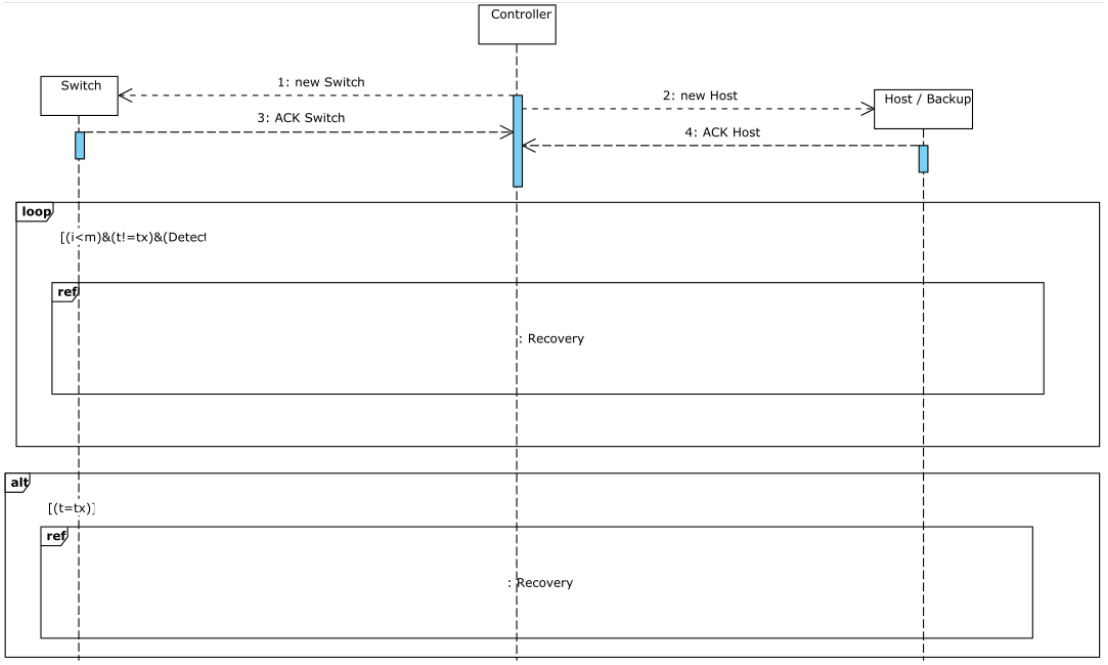


Figura 5.7: Diagrama de secuencia del sistema.

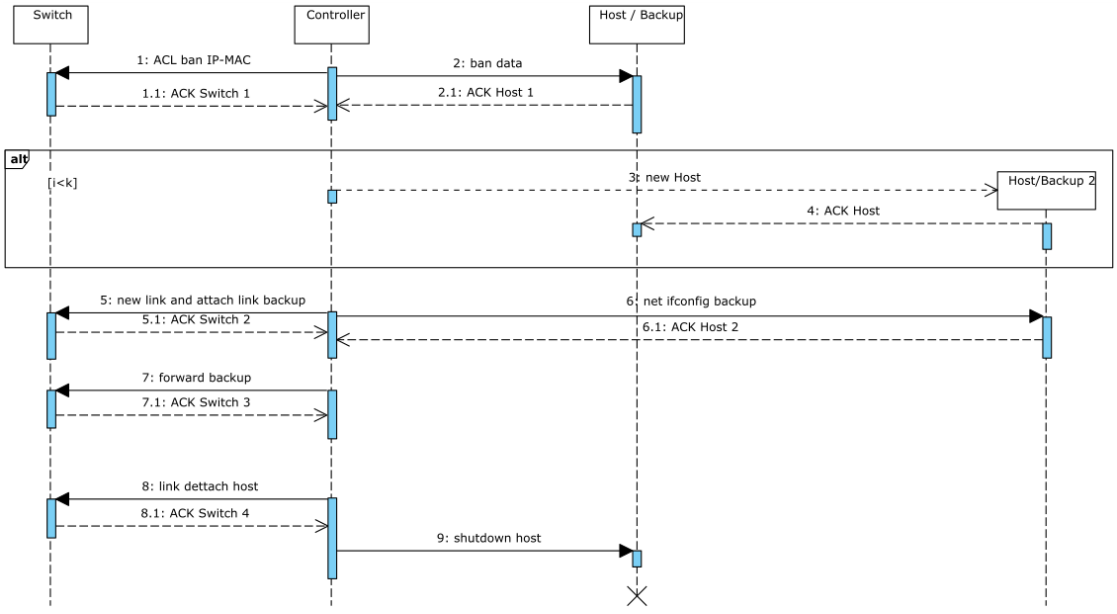


Figura 5.8: Diagrama de secuencia del proceso de recuperación.

5.4. Implementación del mecanismo de defensa y limitaciones

El mecanismo de defensa proactivo y reactivo propuesto se ha implementado en Java como módulo de OpenDayLight, de este modo el controlador es capaz de analizar los paquetes entrantes en el sistema, añadir los flujos necesarios en el *switch* para que éste conmute y encamine los paquetes de un mismo flujo y además realizar una monitorización y control sobre la cantidad de paquetes que entran en el sistema siendo capaz de detectar patrones de ataques DDoS. Dado que no se contaba con una arquitectura SDN real y se ha tenido que utilizar el emulador Mininet, el sistema implementado tiene ciertas limitaciones que el software impone.

En primer lugar no se puede actuar sobre la arquitectura de la red modificando el estado de los nodos o la topología de la red en tiempo real. Para poder evaluar las prestaciones del sistema propuesto, se ha decidido no utilizar ninguna réplica m como *backup* del sistema, de modo que todos los nodos de la topología actúan como nodos n porque Mininet no nos permite realizar una modificación en los enlaces de la red ni en la distribución de los nodos directamente desde el código de Java, sino que es necesario ejecutar *scripts* de Python a través de su consola.

Para solucionar este problema se ha decidido evaluar los rejuvenecimientos del sistema como periodos temporales en los que los nodos se encuentran ocupados y no sirven ningún tipo de servicio, al transcurrir el tiempo de recuperación teórico, el nodo vuelve a funcionar y dar servicio correctamente. Como T_{coste} se ha seleccionado un intervalo temporal comprendido entre 10 y 14 segundos que es lo que le cuesta en promedio a Amazon EC2 realizar un despliegue de un equipo pequeño [OIY+09].

5.4.1. Diagrama de clases

Para describir el funcionamiento de la implementación en primer lugar se debe hacer referencia al Diagrama de clases (véase la Figura 5.9) que muestra una perspectiva a grandes rasgos de cómo se relacionan las diferentes clases del código y cómo se organiza.

- **Activator:** Es la clase encargada de conectar el módulo desarrollado con el software de OpenDaylight y activar los módulos que necesitamos.
- **Flow_management:** es la estructura de datos para almacenar toda la información relacionada con cada flujo.
- **Server:** es la estructura de datos que almacena toda la información de cada servidor.
- **Client:** es la estructura de datos que almacena la información relacionada con cada cliente.
- **PacketHandler:** es la clase que implementa el *scheduler* de paquetes y flujos en el sistema. Se encarga de añadir los nuevos flujos, modificarlos, eliminarlos y obtener la informacion de dichos flujos cuando el sistema de defensa necesita obtener datos.

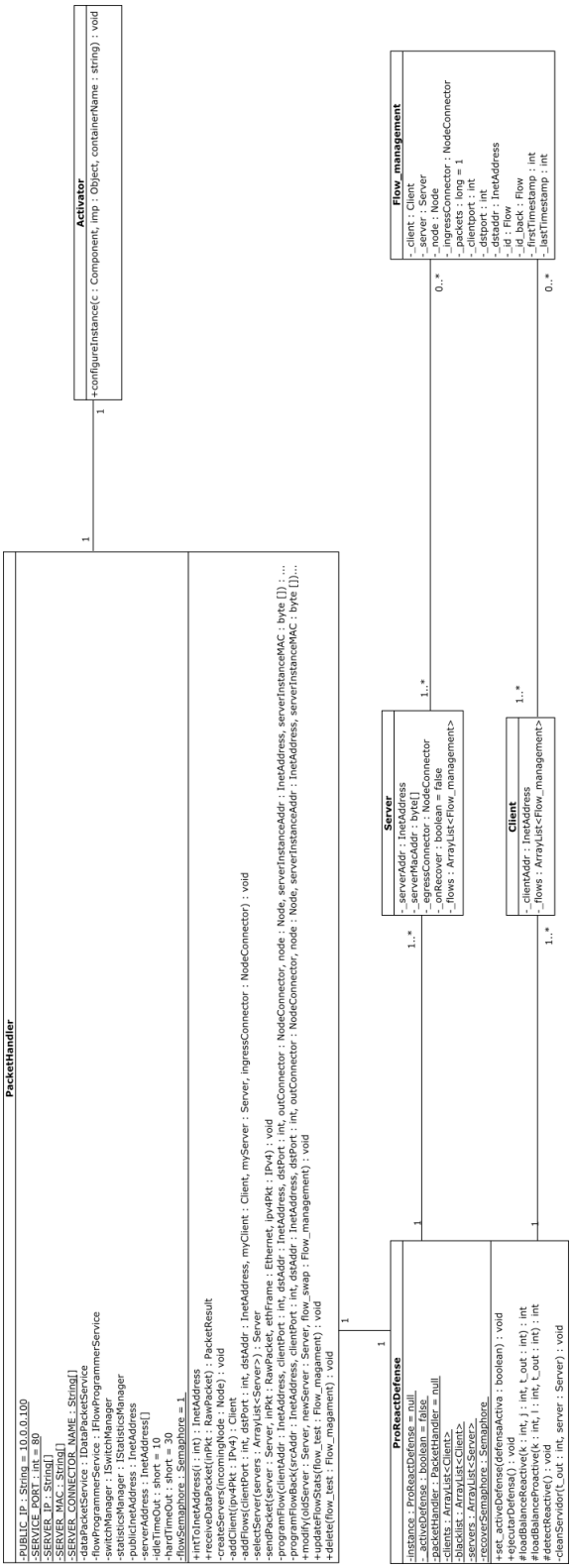


Figura 5.9: Diagrama de clases de la implementación.

- **ProReactDefense:** es la clase que implementa el sistema de defensa, además es la que realiza los rejuvenecimientos del sistema y gestiona los intervalos temporales reactivo y proactivo.

5.4.2. Diagramas de secuencia

Para explicar el funcionamiento del sistema y el mecanismo de defensa proactivo y reactivo en mayor detalle se han realizado dos diagramas de secuencia que explican en profundidad cómo funciona el sistema (véase la Figura 5.10) y otro de cómo funciona la función encargada del mecanismo de defensa (véase la Figura 5.11). Además se ha añadido un pseudocódigo que explica en detalle cómo funciona el *scheduler* (véase el Algoritmo 1).

El sistema comienza cuando se recibe el primer paquete de datos en nuestro *scheduler*, el sistema comienza a funcionar ejecutando en paralelo por una parte el mecanismo de defensa mediante *executeDefense()* en el *thread* ProReactDefense y en el *thread* PacketHandler se ejecuta la función *receivePacket()* que es el *scheduler* de paquetes, de modo que si es tráfico TCP se procesa y gestiona el servicio mediante la función *receivePacket()* y sino se ignoran dichos paquetes.

El mecanismo de defensa que se observa en la Figura 5.11 está funcionando constantemente en un bucle infinito de modo que siempre se ejecuta la detección de ataques de manera reactiva (*detectReactive()*) y el balanceo de carga reactivo (*loadBalancerReactive()*) tantas veces como *slots* z contenga el intervalo reactivo T_R, posteriormente se ejecuta el periodo proactivo T_P mediante la función *loadBalancerProactive()*. Esta ejecución sucesiva implementa el mecanismo proactivo y reactivo comentado en la sección anterior, de modo que cuenta con un intervalo reactivo y un intervalo proactivo que se repiten cíclicamente.

El método *detectReactive()* realiza una detección de patrones en función del número de paquetes servidos en un mismo flujo durante un intervalo de tiempo, de modo que si detecta un ataque de *flooding*, se marca dicho flujo y aplica los mecanismos de mitigación correspondientes, que consisten en eliminar dicha flujo y todos flujos asociados con dicho cliente, añadir dicho cliente a la *blacklist* de modo que no pueda volver a solicitar el servicio (sus paquetes directamente pasan al estado DROP en la tabla de OpenFlow del *switch*) y realizar el rejuvenecimiento sobre el servidor afectado. En el intervalo reactivo la función *loadBalanceReact*(k, j, t_out) se encarga de repartir los flujos de los servidores afectados por ataque DDoS entre los nodos disponibles. En el intervalo proactivo la función loadBalanceProact(k, l, t_out) se encarga de seleccionar cuales son aquellos servidores que deben entrar en rejuvenecimiento y repartir los flujos de estos entre los demás nodos disponibles.

En este caso el *scheduler* está configurado para tan solo procesar paquetes TCP, pero sin embargo se podría implementar para cualquier protocolo. Además, se pueden procesar los datos del paquete directamente de modo que se podrían implementar mecanismos de detección más complejos.

En el Algoritmo 1 se explica cómo funciona el *scheduler* de paquetes con mayor detenimiento. De todos aquellos paquetes que se reciben se toman los puertos origen y

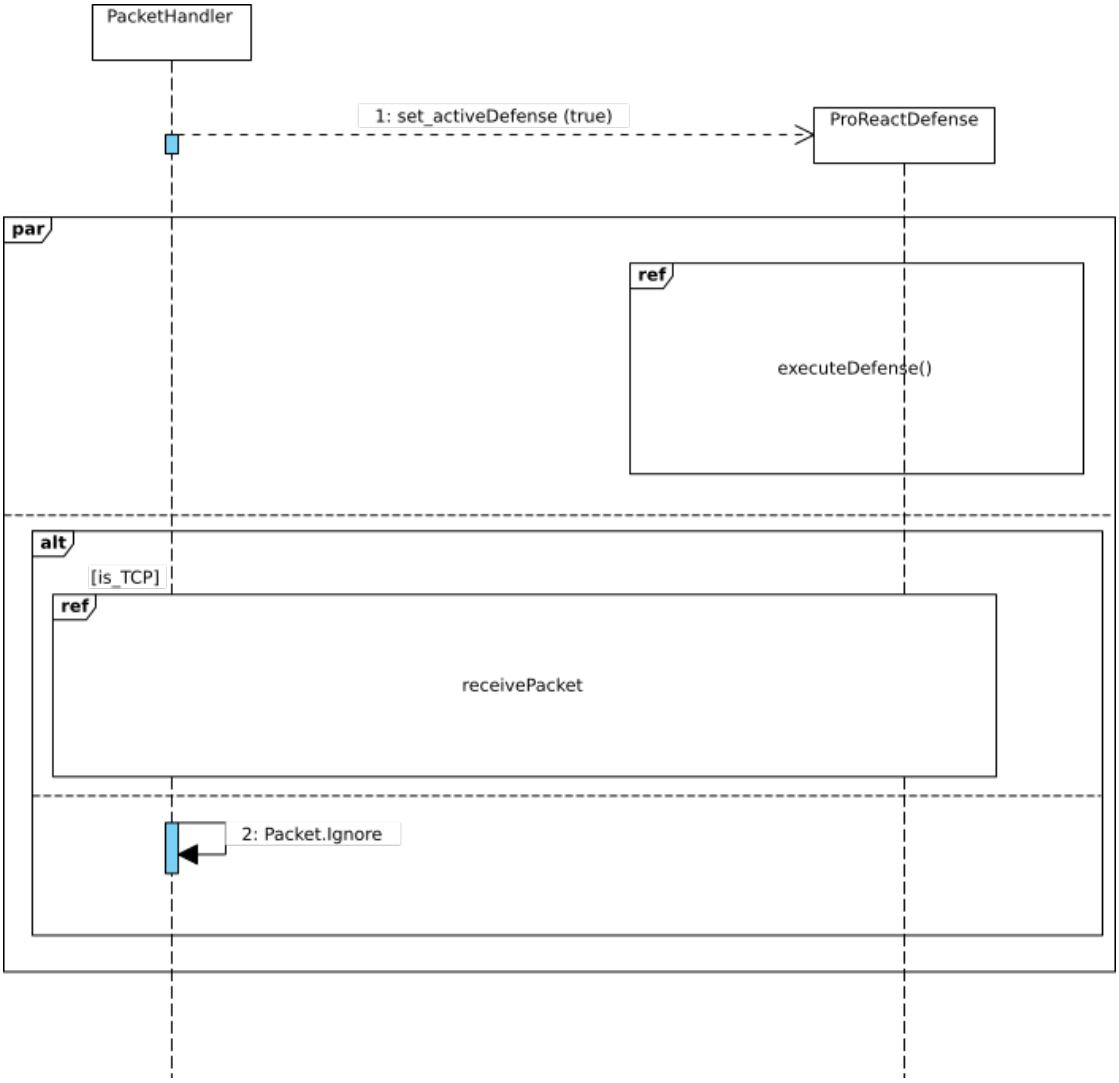


Figura 5.10: Diagrama de secuencia del sistema implementado.

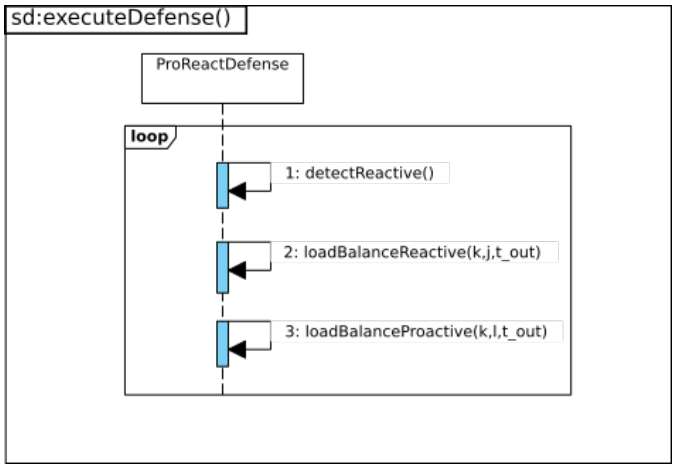


Figura 5.11: Diagrama de secuencia del mecanismo de defensa *executeDefense()*.

destino (líneas 1 y 2), las direcciones IP fuente y destino (línea 3) y el enlace del *switch* por el que se ha encaminado(línea 4). En primer lugar, si el cliente no se encuentra registrado en el sistema se añade a la lista de clientes (líneas 6 y 7), posteriormente se obtiene la lista de flujos de dicho cliente (línea 9). Si el flujo del paquete entrante ya existe, simplemente se actualiza la información de dicho flujo añadiendo un nuevo paquete al contador, se actualiza su marca temporal y se obtiene el servidor que se encarga de servir dicho flujo (líneas 10, 11 y 12). Si por el contrario, no existe dicho flujo se crea, se asigna un servidor a dicho flujo y se añade a la lista de flujos del sistema (líneas 14, 15 y 16). Independientemente del camino hasta llegar al final, se envía el paquete al servidor correspondiente (línea 18) y anteriormente decidido.

Algoritmo 1: *receivePacket*

Input : Packet p
Output: Consume or ignore the packet (PacketResult.Ignore or PacketResult.Consume)

```
1  Get client port clientPort from p
2  Get destination port dstPort from p
3  Get IP address dstAddr & srcAddr from p
4  Get link ingressConnector from p
5  Get list of client C from ProReactDefense
6  if ∄ client c ∈ C, srcAddr = c.srcIP then
7  |  c = createAndAddClient(p)
8  end
9  Get list of flows F from c
10 if ∃ f ∈ F, clientPort = f.clientPort then
11 |  updateFlows(f)
12 |  Get Server s from f
13 else
14 |  Get list of Server S from ProReactDefense
15 |  s = SelectServers(S)
16 |  addFlows (clientPort, dstPort, dstAddr, c, s, ingressConnector)
17 end
18 sendPacket(s,p)
```

5.5. Disponibilidad

Todo el código fuente de la implementación del mecanismo de defensa se encuentra disponible en el repositorio *online* BitBucket.La carpeta doc contiene todo el Javadoc (documentación en formato HTML del código del proyecto).

`https://bitbucket.org/Nessaji/sdnproreactdefense`

con licencia GNU General Public License publicada por la Free Software Foundation en su versión 3.

Capítulo 6

Evaluación y resultados

Para evaluar las prestaciones del mecanismo de defensa proactivo y reactivo propuesto se han considerado diferentes tipos de configuraciones del sistema limitadas por el *hardware* del equipo donde se han realizado. El equipo donde se han realizado las pruebas cuenta con un Intel Core i7-2630QM CPU @ 2.00GHz x 8 y 6GBytes de memoria RAM, con el sistema operativo Ubuntu 14.04 en su versión más estable, Mininet versión 2.2.1 y el software del controlador OpenDayLight versión base 0.2.2.

Para ello se han realizado tres escenarios distintos, la característica común de todos ellos es que cuentan con 2 nodos servidor, 4 nodos que actúan como clientes legítimos y dentro de cada uno 3 configuraciones distintas según su probabilidad de ataque que se ha variado entre 25 %, 50 % y 75 %. En cada configuración se iniciado un *script* que ejecuta cada atacante para realizar un ataque DDoS a la IP pública del servidor cada 60s.

1. **Escenario 1**: 1 atacante.
2. **Escenario 2**: 2 atacantes.
3. **Escenario 3**: 3 atacantes.

Para demostrar la viabilidad del mecanismo de defensa se ha medido el tiempo de servicio de los clientes en los mismos escenarios bajo la misma configuración en un primer lugar (véase la Figura 6.1) sin activar la defensa y posteriormente activando el mecanismo, obteniendo un total de 7200 muestras que se resumen a continuación en la Figura 6.2 y en la Figura 6.3.

Los clientes (rango 10.0.0.3 al 10.0.0.6) solicitan el fichero en paralelo al servidor (nodo 10.0.0.1 y 10.0.0.2) y es el servidor quien balancea la carga del sistema repartiendo los diferentes flujos entre sus nodos de manera transparente para los clientes. Los atacantes (rango 10.0.0.7 al 10.0.0.9, según el número de atacantes) no conocen la arquitectura del servidor y tan sólo pueden atacar a la dirección pública que en este caso es la 10.0.0.100, de modo que el sistema internamente es capaz de gestionar y balancear los flujos según la situación que se encuentre. El *scheduler* tiene implementando un balanceo de carga

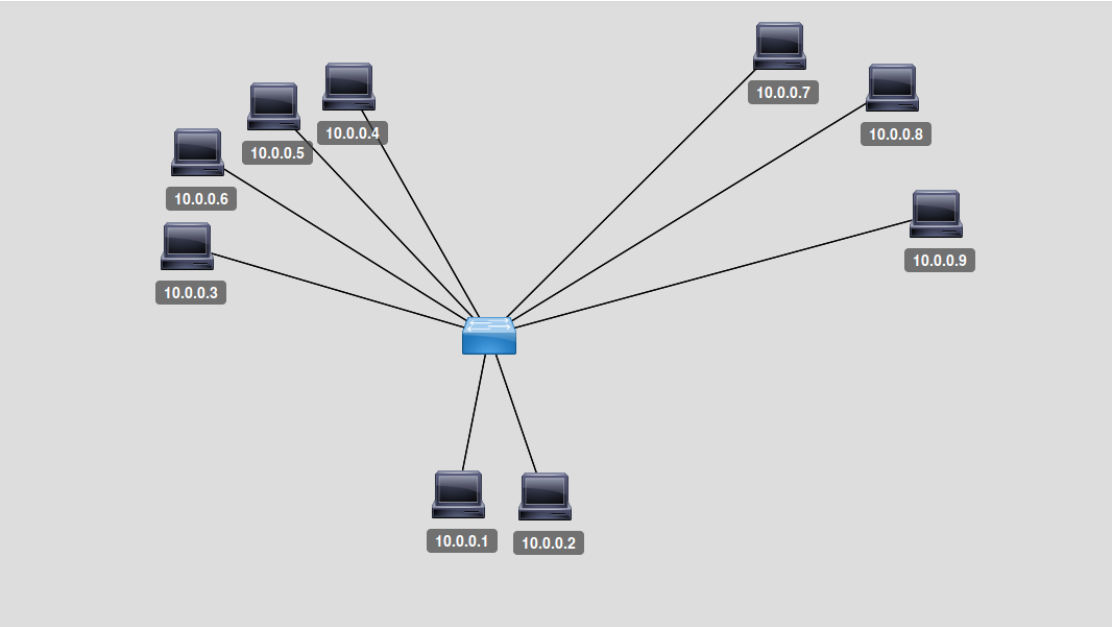
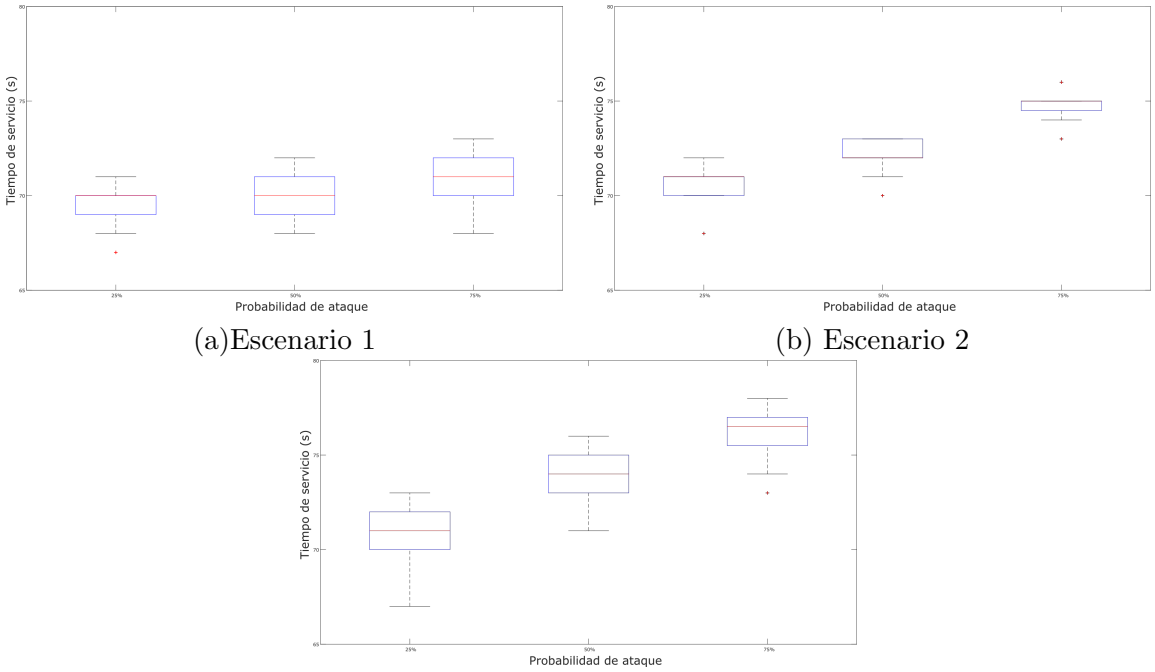


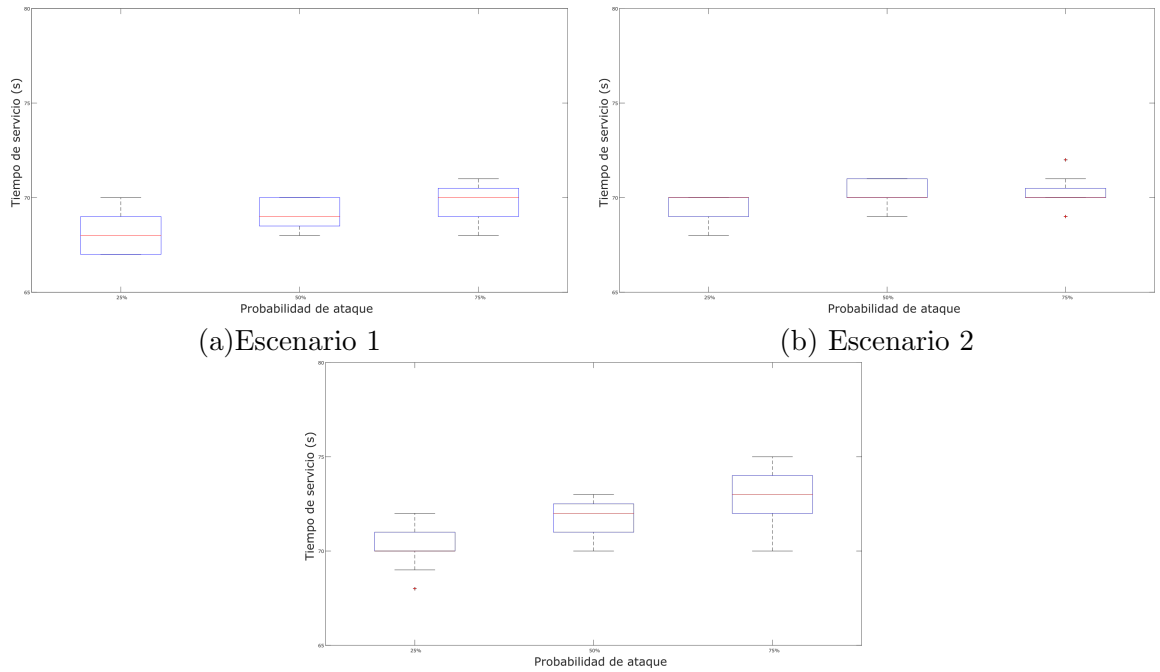
Figura 6.1: Topología de la red de pruebas simulada.



(a)Escenario 1 (b) Escenario 2

(c) Escenario 3

Figura 6.2: Comparativa tiempos de servicio con el mecanismo de defensa desactivado.



(a)Escenario 1 (b) Escenario 2

(c) Escenario 3

Figura 6.3: Comparativa tiempos de servicio con el mecanismo de defensa activado.

Round-Robin de modo que reparte los flujos teniendo mayor prioridad aquel servidor con menor cantidad. Cuando se realizan los rejuvenecimientos de los nodos del servidor tanto en la casuística proactiva como reactiva los flujos se balancean hacia el nodo activo.

Sobre cada una de ellas se ha aplicado el método de Montecarlo (véase Sección 2.7) de modo que se han realizado 10 pruebas sobre cada configuración. En cada una de éstas pruebas se ha descargado un fichero de 105 MB a una tasa límite de 1,46MB/s, de este modo cuatro clientes legítimos en paralelo han solicitado la descarga del fichero durante 10 iteraciones en cada una de las pruebas de modo que se han obtenido 400 muestras para cada configuración.

Para un fichero de dicho tamaño en caso de que el sistema se encontrase con la capacidad de servir a su tasa límite, el fichero tardaría entre 66 y 68 segundos en descargarse. Se puede observar cómo aumentar la frecuencia de los ataques y el número de los ataques es directamente proporcional al aumento del tiempo de servicio llegando a provocar en la configuración más crítica que se tarde hasta 10 segundos más en completar la descarga.

En los casos en que la defensa se encuentra activa se puede observar que el sistema es incluso capaz de en algunas ocasiones mitigar el ataque hasta el punto que el tiempo de servicio es el mismo que el ideal. En un escenario mucho más agresivo como es el escenario 3 con una probabilidad de ataque del 75 % el mecanismo es capaz de mitigar ataques pero cuando se producen más ataques que nodos disponibles el mecanismo de defensa no es capaz de soportarlo adecuadamente. Esto se solucionaría aumentando la cantidad de nodos en función de la carga de los ataques tal y como se expone en el

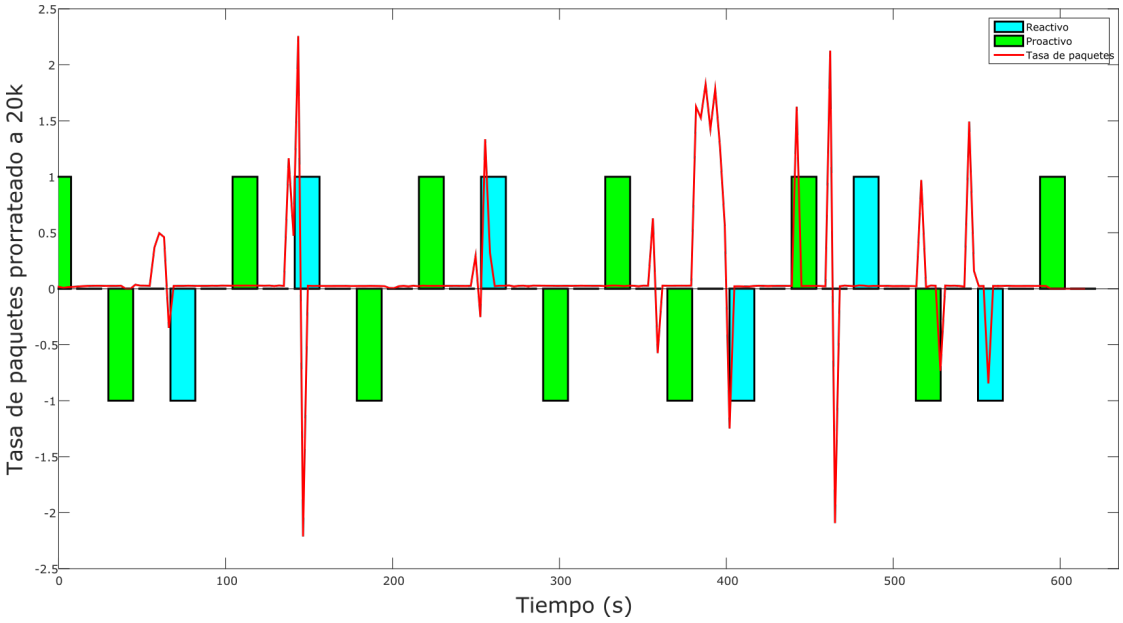


Figura 6.4: Funcionamiento del sistema ante ataques DDoS.

Capítulo 5 y que en la evaluación no se ha podido implementar por limitaciones de *hardware* y del simulador respecto a la topología.

En la Figura 6.4 se puede observar el funcionamiento del sistema a lo largo de una configuración, en este caso representa el escenario 2 en la que se cuenta con 2 nodos atacantes con una probabilidad del 50 % cada 60s. En la figura se puede diferenciar el cambio de estado de los dos nodos servidores, representado el servidor 1 en el intervalo discreto [0,1] indicando el estado 0 que el servidor se encuentra en funcionamiento y el estado 1 que se encuentra en rejuvenecimiento, de forma análoga se representa el servidor 2 en el intervalo discreto [-1,0] donde en esta ocasión se representa con el estado -1 su rejuvenecimiento.

Se pueden diferenciar varios cambios de estado debido a los intervalos proactivos y reactivos del sistema, la línea roja representa la tasa de paquetes que sirve el sistema en cada segundo normalizada a 20.000 paquetes/segundo. Se pueden observar distintas fluctuaciones bruscas que son producidas por los distintos ataques DDoS. En particular se puede observar cómo cuando se produce una mitigación de los ataques se produce una tasa negativa dado que se eliminan todos los paquetes de dicho cliente del sistema. Cabe destacar que el sistema de detección implementado tarda alrededor de unos 10 segundos en actuar pero en ocasiones según la posición temporal y el intervalo en el que se encuentre (proactivo o reactivo) puede tardar mayor cantidad de tiempo en actuar, como es el caso del ataque en el instante 380 segundos que dura bastante tiempo hasta que el sistema lo mitiga (concretamente, 25s).

En algunas situaciones coincide que el intervalo proactivo mitiga ataques DDoS porque justo toca el rejuvenecimiento de dicho servidor en prácticamente el mismo instante

proactivo que le toca rejuvenecerse (véase el instante temporal de 515s).

Capítulo 7

Conclusiones

Este capítulo presenta algunas conclusiones obtenidas de la elaboración de este trabajo de fin de grado así como posibles líneas de trabajo futuras con los que mejorarlo.

La arquitectura SDN vio la luz hace muy poco tiempo (2014) y por lo tanto todavía queda mucho desarrollo por delante. El potencial de esta arquitectura de red es enorme y se está viendo reflejado entre la gran mayoría de empresas tecnológicas. Paralelamente se están presentando un gran aumento de ataques DDoS a todo tipo de servicios en Internet; de modo que aprovechar la nueva arquitectura de red para mitigar y solucionar los problemas que estos están causando puede resultar muy beneficioso. En este proyecto se pueden apreciar las mejoras conseguidas respecto a un sistema sin defensa y cómo es necesario implementar algún tipo de sistema de mitigación ante ataques DDoS.

Como resultado global del proyecto se ha implementado un mecanismo de defensa viable y extrapolable a infraestructuras de mayor tamaño y carga. Además, se han adquirido conocimientos sobre la arquitectura de red SDN y sobre los tipos de ataques DDoS. Pese a los inconvenientes acontecidos debido a las limitaciones del *hardware* y *software* se ha probado la viabilidad y funcionamiento del sistema propuesto. Mediante simulaciones se ha podido demostrar cómo la defensa proporciona una capa de seguridad adicional al sistema de modo que es capaz de mitigar ataques DDoS.

Otro concepto adquirido a raíz de este trabajo es que un buen diseño, y una buena organización y planificación son elementos esenciales a la hora de realizar cualquier proyecto ya que facilitan enormemente su desarrollo y ahorran gran cantidad de tiempo y esfuerzo.

Las líneas de trabajo futuros que se pueden realizar a partir de este proyecto con el objetivo de ampliar o complementar este trabajo se pueden dividir en función de los ámbitos de este proyecto. En primer lugar, se pueden centrar en formalizar dicho mecanismo de defensa mediante modelos matemáticos basados en cadenas de Markov. También se puede actualizar la versión del controlador OpenDayLight y desarrollar la defensa para la nueva versión, orientada en *model-driven*. Por último, se puede profundizar en los mecanismos de detección que se aplican en el mecanismo de defensa desarrollando nuevos patrones o implementando los más extendidos en la comunidad científica.

Bibliografía

[AKK+13] Marios Anagnostopoulos, Georgios Kambourakis, Panagiotis Kopanos, Georgios Louloudakis, and Stefanos Gritzalis. DNS amplification attack revisited. *Computers & Security*, 39:475–485, 2013.

[AL14] Javed Ashraf and Saeed Latif. Handling intrusion and DDoS attacks in Software Defined Networks using machine learning techniques. In *Software Engineering Conference (NSEC), 2014 National*, pages 55–60. IEEE, 2014.

[Arb10] Arbortnetworks. The Internet Goes to War. [Online], Diciembre 2010. `https://asert.arbornetworks.com/the-internet-goes-to-war/`.

[Ben13] Benny Har-Even. CTO, SK Telecom, South Korea: "SDN and network virtualisation hold great promise for mobile carriers". [Online], Abril 2013. http://telecoms.com/interview/cto-sk-telecom-south-korea-sdn-and-network-virtualisation-hold-great-promise-for-mobile-carriers/.

[BMP10] Rodrigo Braga, Edjard Mota, and Alexandre Passito. Lightweight DDoS flooding attack detection using NOX/OpenFlow. In *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, pages 408–415. IEEE, 2010.

[Chu15] Chun-Jen Chung. *SDN-based Proactive Defense Mechanism in a Cloud System*. PhD thesis, ARIZONA STATE UNIVERSITY, 2015.

[Cis11] Cisco. Open Networking Foundation Formed to Speed Network Innovation. [Online], Marzo 2011. `http://newsroom.cisco.com/press-release-content?articleId=5973381`.

[Cis15] Cisco. Cisco controller APIC. [Online], 2015. `http://www.cisco.com/c/en/us/products/cloud-systems-management/application-policy-infrastructure-controller-apic/index.html`.

[CMW13] Yonghong Chen, Xinlei Ma, and Xinya Wu. DDoS detection algorithm based on preprocessing network traffic predicted method and chaos theory. *Communications Letters, IEEE*, 17(5):1052–1054, 2013.

[CSD14] CSDN. sqx2011. OVS. [Online], 2014. `http://blog.csdn.net/sqx2011/article/details/39344869`.

[DM04] Christos Douligeris and Aikaterini Mitrokotsa. DDoS attacks and defense mechanisms: classification and state-of-the-art. *Computer Networks*, 44(5):643–666, 2004.

[EAJ+] Egbenimi Beredugo Eskca, Omar Abuzaghleh, Priya Joshi, Sandeep Bondugula, Takamasa Nakayama, and Amreen Sultana. Software Defined Networks' Security: An Analysis of Issues and Solutions.

[FRZ13] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to SDN. *Queue*, 11(12):20, 2013.

[Gar00] Lee Garber. Denial-of-service attacks rip the Internet. *Computer*, (4):12–17, 2000.

[Goo12] Google. OpenFlow @ Google. [Online], Abril 2012. `http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf`.

[HBPG15] Akram Hakiri, Pascal Berthou, Prithviraj Patil, and Aniruddha Gokhale. Towards a Publish/Subscribe-based Open Policy Framework for Proactive Overlay Software Defined Networking. *ISIS*, pages 15–115, 2015.

[HP15] HP. Hp controller VAN. [Online], 2015. `http://h17007.www1.hp.com/us/en/networking/products/network-management/HPVANSDNControllerSoftware/index.aspx#.VXA2pq3tmko`.

[KBP14] Ankunda R Kiremire, Matthias R Brust, and Vir V Phoha. Using network motifs to investigate the influence of network topology on PPM-based IP traceback schemes. *Computer Networks*, 72:14–32, 2014.

[KKSG] Karamjeet Kaur, Krishan Kumar, Japinder Singh, and Navtej Singh Ghumman. Programmable Firewall Using Software Defined Networking.

[KREV+15] Diego Kreutz, Fernando MV Ramos, P Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, 103(1):14–76, 2015.

[KRV13] Diego Kreutz, Fernando Ramos, and Paulo Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 55–60. ACM, 2013.

[KS13] P. Arun Raj Kumar and S. Selvakumar. Detection of distributed denial of service attacks using an ensemble of adaptive and hybrid neuro-fuzzy systems. *Computer Communications*, 36(3):303–319, 2013.

[Kum07] Sanjeev Kumar. Smurf-based distributed denial of service (ddos) attack amplification in internet. In *Internet Monitoring and Protection, 2007. ICIMP 2007. Second International Conference on*, pages 25–25. IEEE, 2007.

[KVF+12] Sanjeev Khanna, Santosh S Venkatesh, Omid Fatemieh, Fariba Khan, Carl Gunter, et al. Adaptive selective verification: An efficient adaptive countermeasure to thwart dos attacks. *Networking, IEEE/ACM Transactions on*, 20(3):715–728, 2012.

[KZMB14] Rahamatullah Khondoker, Adel Zaalouk, Ronald Marx, and Kpatcha Bayarou. Feature-based comparison and selection of Software Defined Networking (SDN) controllers. In *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*, pages 1–7. IEEE, 2014.

[LBZ+14] Jun Li, Skyler Berg, Mingwei Zhang, Peter Reiher, and Tao Wei. Drawbridge: software-defined DDoS-resistant traffic engineering. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 591–592. ACM, 2014.

[LHK+14] Sharon Lim, Jung-Ik Ha, Heonhwan Kim, Youngjae Kim, and Songping Yang. A SDN-oriented DDoS blocking scheme for botnet-based attacks. In *Ubiquitous and Future Networks (ICUFN), 2014 Sixth International Conf on*, pages 63–68. IEEE, 2014.

[LJP+08] Jae-Seo Lee, HyunCheol Jeong, Jun-Hyung Park, Minsoo Kim, and Bong-Nam Noh. The activity analysis of malicious http-based botnets using degree of periodic repeatability. In *Security Technology, 2008. SECTECH'08. International Conference on*, pages 83–86. IEEE, 2008.

[LWLP15] Shibo Luo, Jun Wu, Jianhua Li, and Bei Pei. A Defense Mechanism for Distributed Denial of Service Attack in Software-Defined Networks. In *Frontier of Computer Science and Technology (FCST), 2015 Ninth International Conference on*, pages 325–329. IEEE, 2015.

[MDC+14] Nishat Mowla, Inshil Doh, Kijoon Chae, et al. Multi-defense Mechanism against DDoS in SDN Based CDNi. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2014 Eighth International Conference on*, pages 447–451. IEEE, 2014.

[Min15] Mininet. An Instant Virtual Network on your Laptop (or other PC). [Online], 2015. `http://mininet.org/`.

[Net12] NetworkComputing. SDN Is Business, OpenFlow Is Technology. [Online], 2012. `http://www.networkcomputing.com/networking/sdn-business-openflow-technology/53316220`.

[Nol13] Alejandro Nolla. Amplification DDoS attacks with games servers from the perspective of both the attacker and the defender. GreHack, 2013.

[NOX15] NOXRepo.org. Pox controller. [Online], 2015. `http://www.noxrepo.org/pox/about-pox/`.

[ÖB15] İlker Özçelik and Richard R Brooks. Deceiving entropy based DoS detection. *Computers & Security*, 48:234–245, 2015.

[OIY+09] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of EC2 cloud computing services for scientific computing. In *Cloud computing*, pages 115–131. Springer, 2009.

[Ope14] Open vSwitch. What is Open vSwitch? [Online], 2014. `http://openvswitch.org/`.

[Ope15a] Open Networking Foundation. Openflow switch specification, version 1.3.5. [Online], Marzo 2015. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf`.

[Ope15b] Open Networking Foundation. SDN-resources: OpenFlow. [Online], 2015. `https://www.opennetworking.org/sdn-resources/openflow`.

[Ope15c] OpenFlow. OpenFlow. [Online], 2015. `http://archive.openflow.org/wp/learnmore/`.

[Ope16a] OpenDayLight. OpenDaylight Platform. [Online], 2016. `https://www.opendaylight.org/`.

[Ope16b] OpenStack. Scenario: Legacy with Open vSwitch. [Online], Enero 2016. `http://docs.openstack.org/liberty/networking-guide/scenario_legacy_ovs.html`.

[PDBAA15] Thomas Pfeiffenberger, Jia Lei Du, Pedro Bittencourt Arruda, and Alessandro Anzaloni. Reliable and flexible communications for power systems: Fault-tolerant multicast with SDN/OpenFlow. In *New Technologies, Mobility and Security (NTMS), 2015 7th International Conference on*, pages 1–6. IEEE, 2015.

[PLR07] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Survey of network-based defense mechanisms countering the DoS and DDoS problems. *ACM Computing Surveys (CSUR)*, 39(1):3, 2007.

[Pro15] Project Floodlight. Floodlight controller. [Online], 2015. `http://www.projectfloodlight.org/floodlight/`.

[RH15] Christian Röpke and Thorsten Holz. SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks. In *Research in Attacks, Intrusions, and Defenses*, pages 339–356. Springer, 2015.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[SBC+10] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *Parallel and Distributed Systems, IEEE Transactions on*, 21(4):452–465, 2010.

[sdx13] sdxCentral. What are sdn controllers? [Online], 2013. `https://www.sdxcentral.com/resources/sdn/sdn-controllers/`.

[SNK12] Myung-Ki Shin, Ki-Hyuk Nam, and Hyoung-Jun Kim. Software-defined networking (SDN): A reference architecture and open APIs. In *ICT Convergence (ICTC), 2012 International Conference on*, pages 360–361. IEEE, 2012.

[SPY+13] Seungwon Shin, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, Guofei Gu, and Mabry Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *NDSS*, 2013.

[SS77] Robert Solovay and Volker Strassen. A fast Monte-Carlo test for primality. *SIAM journal on Computing*, 6(1):84–85, 1977.

[SSKS10] Monika Sachdeva, Gurvinder Singh, Krishan Kumar, and Kuldip Singh. Measuring impact of DDOS attacks on web services. 2010.

[SY13] Ahmad Sanmorino and Setiadi Yazid. Ddos attack detection method and mitigation using pattern of the flow. In *Information and Communication Technology (ICoICT), 2013 International Conference of*, pages 12–16. IEEE, 2013.

[TYZM14] Theerasak Thapngam, Shui Yu, Wanlei Zhou, and S Kami Makki. Distributed Denial of Service (DDoS) detection by traffic pattern analysis. *Peer-to-peer networking and applications*, 7(4):346–358, 2014.

[Ver12] Verizon. Adoption of SDN: Progress Update. [Online], Abril 2012. `http://opennetsummit.org/archives/apr12/elby-tue-sdn.pdf`.

[WCXJ13] Wei Wei, Feng Chen, Yingjie Xia, and Guang Jin. A rank correlation based detection against distributed reflection DoS attacks. *Communications Letters, IEEE*, 17(1):173–175, 2013.

[WZLH15] Bing Wang, Yao Zheng, Wenjing Lou, and Y Thomas Hou. DDoS attack protection in the era of cloud computing and Software-Defined Networking. *Computer Networks*, 81:308–319, 2015.

[YM05] Jian Yuan and Kevin Mills. Monitoring the macroscopic effect of DDoS flooding attacks. *Dependable and Secure Computing, IEEE Transactions on*, 2(4):324–335, 2005.

[ZJT13] Saman Taghavi Zargar, Jyoti Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *Communications Surveys & Tutorials, IEEE*, 15(4):2046–2069, 2013.

[ZJW+14] Wei Zhou, Weijia Jia, Sheng Wen, Yang Xiang, and Wanlei Zhou. Detection and defense of application-layer DDoS attacks in backbone web traffic. *Future Generation Computer Systems*, 38:36–46, 2014.

Acrónimos

- API Application Programming Interface.
- DDoS Distributed Denial of Service.
- DOM Document Object Model.
- DRDoS Distributed Re-flection Denial of Service.
- DNS Domain Name System.
- HTTP Hypertext Transfer Protocol.
- ICMP Internet Control Message Protocol.
- IP Internet Protocol.
- ISO International Organization for Standardization.
- JSON JavaScript Object Notation.
- MAC Media Access Control.
- NAT Network Address Translation.
- OO Orientado a Objetos.
- OMG Object Management Group.
- ONF Open Networking Foundation.
- OSGi Open Services Gateway Initiative.
- PCEP Path Computation Element Protocol.
- QoS Quality of Service.
- REST Representational State Transfer.
- RPC Remote Procedure Call.
- SAL Service Abstraction Layer.

- SDN Software Defined Networking.
- SNMP Simple Network Management Protocol.
- SSH Secure Shell.
- TCP Transmission Control Protocol.
- TLS Transport Layer Security.
- UDP User Datagram Protocol.
- UML Unified Modeling Language.

Anexo A

Horas de trabajo

Con el objetivo de controlar el tiempo de desarrollo del proyecto, se ha realizado un seguimiento de las horas dedicadas a cada parte del mismo. En la Figura A.1 se puede ver el diagrama de Gantt por semanas y tareas.

La primera etapa del proyecto ha consistido en estudiar qué es la arquitectura SDN, cuáles son sus controladores disponibles, qué son los ataques DDoS y qué es un mecanismo proactivo y reactivo. Además, se ha seleccionado el controlador y qué ataque DDoS se estudiaría en mayor profundidad. Tras el análisis del problema y el estudio de las características de la arquitectura SDN, se ha planteado un mecanismo de defensa proactivo y reactivo sobre el que finalmente se evaluaron las pruebas.

Al final, de los 10 meses de proyecto se ha estimado un coste aproximado de 290 días de trabajo (véase Tabla A.1). Aunque el número de horas excede las previstas para el proyecto, este tiempo ha sido necesario para el estudio de la arquitectura SDN y cómo implementar un mecanismo de defensa con estas características.

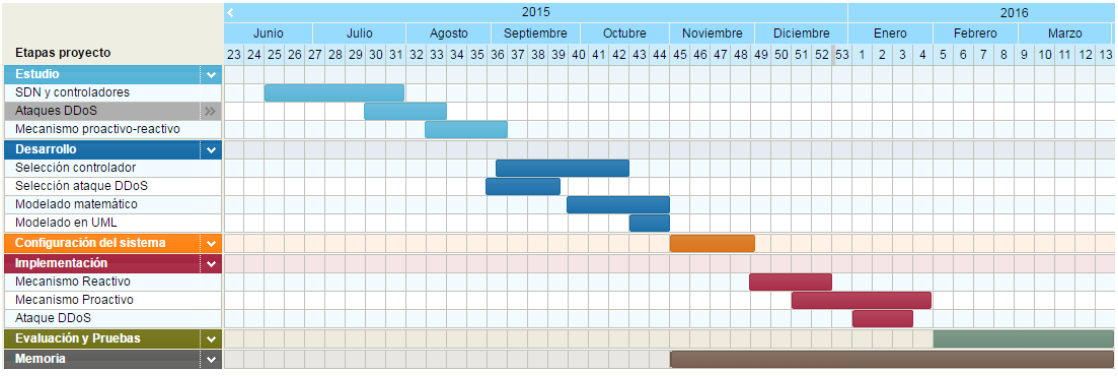


Figura A.1: Diagrama de Gantt.

Tarea	Comienzo	Fin	Duración
Estudio	15/06/2015	4/09/2015	**82d**
SDN y controladores	15/06/2015	31/07/2015	47d
Ataques DDoS	17/07/2015	14/08/2015	28d
Mecanismo proactivo-reactivo	7/08/2015	4/09/2015	28d
Desarrollo	1/09/2015	31/10/2015	**61d**
Selección controlador	2/09/2015	16/10/2015	50d
Selección ataque DDoS	1/09/2015	25/09/2015	25d
Modelado matemático	25/09/2015	30/10/2015	35d
Modelado en UML	19/10/2015	31/10/2015	12d
Configuración del sistema	2/11/2015	30/11/2015	**28d**
Implementación	27/11/2015	25/01/2016	**59d**
Mecanismo reactivo	27/11/2015	24/12/2015	27d
Mecanismo proactivo	8/12/2015	25/01/2016	48d
Ataque DDoS	4/01/2016	22/01/2016	18d
Evaluación y pruebas	01/02/2016	1/04/2016	**60d**
Memoria	01/11/2015	1/04/2016	**152d**

Tabla A.1: Duración de cada tarea.

Anexo B

Configuración del entorno de trabajo

En estos apéndices se recogen documentos para configurar el sistema correctamente antes de poder realizar las pruebas sobre el mismo.

B.1. Configuración de Mininet

Instalar Mininet en Ubuntu es una tarea sencilla que se puede realizar de dos maneras. Se puede hacer uso de los paquetes para Ubuntu ya preparados o bien compilar el código fuente.

En este caso se ha elegido usar la compilación de código fuente, ya que la última versión de Mininet no estaba disponible en los repositorios oficiales en el momento de su instalación.

Los pasos para su instalación son:

```
1          git clone git://github.com/mininet/mininet
2
3          cd mininet
4
5          sudo ./util/install.sh -a
```

Una vez instalado, para comprobar su funcionamiento podemos realizar el comando

```
1          mn --test pingall
```

Este comando ejecuta una topología básica y envía mensajes de ping entre todos los equipos simulados.

B.2. Configuración controlador OpenDayLight

Se ha utilizado la versión base 0.2.2, sin embargo se han desarrollado varias versiones posteriores que amplían las funcionalidades de ésta.

La ejecución de OpenDayLight puede presentar un problema y es que no conecten los *switches* emulados en Mininet, para solucionarlo es necesario reiniciar la topología y el controlador.

Para ejecutar el controlador tan solo debemos ejecutar el comando

```
sudo ./run.sh
```

De esta forma ponemos en marcha el controlador y ya se encontrará listo para conectarse con la topología de Mininet. OpenDayLight proporciona una interfaz web a través de la cual se pueden gestionar sus módulos y tablas de flujo.

B.3. Configuración escenario de pruebas

Los módulos de OpenDayLight funcionan a través del compilador Maven, de modo que la carpeta del módulo necesitará todos aquellos archivos adicionales necesarios para compilar como el caso del pom.xml. Este archivo XML es indispensable para Maven ya que contiene la información (dependencias, nombres de proyectos, paquetes y referencias) indispensable para poder construir el proyecto y compilarlo. Una vez se dispone de todo lo necesario ya se puede compilar

```
        sudo mvn clean install
```

Este comando nos proporciona un archivo .jar que ya se puede instalar en OpenDayLight.

El mecanismo de defensa se encuentra programado como un módulo de OpenDayLight, para ello será necesario instalarlo en el controlador y desactivar aquellos módulos que presentan un conflicto. Cuando el controlador se ha iniciado completamente se paran aquellos módulos.

```
        osgi>ss simple
        "Framework is launched"


        id State Bundle
        223 ACTIVE
            org.opendaylight.controller.samples.simpleforwarding_0.5.0.SNAPSHOT
        osgi&gt;

        osgi>stop 223

        osgi>ss load
        "Framework is launched."
        id State Bundle
```

```
14          181 ACTIVE
                org.opendaylight.controller.samples.loadbalancer.northbound_0.5.0.SNAPSHOT

15          212 ACTIVE org.apache.commons.fileupload_1.2.2
16          267 ACTIVE
                org.opendaylight.controller.samples.loadbalancer_0.6.0.SNAPSHOT
17          osgi&gt;
18
19          osgi>stop 267
```

Una vez se han parado estos módulos se puede proceder a instalar y activar el nuevo módulo desarrollado. Para poder ver los *logs* que se incluyen en el código adecuadamente es necesario activarlos en el controlador.

```
1           install file:/home/jorge/Escritorio/sdnproreactdefense/target/
                sdnproreactdefense-0.1.jar
2
3           setLogLevel es.unizar.disco.sdnproreactdefense.PacketHandler trace
4
5           start (id bundle)
```

A continuación se tiene que iniciar Mininet simulando la topología de red, en este caso se ha decidido por una topología sencilla en la que todos nodos están conectados directamente al *switch*. De los nueve nodos simulados, dos actúan como servidor, cuatro como clientes y tres como atacantes.

```
1
2   sudo mn --controller=remote,ip=127.0.0.1 --topo single,9 --mac --arp --switch=
        ovsk,protocol=OpenFlow13
```

Una vez encendidos todos los nodos, en primer lugar se inician los servidores web en el nodo h1 y h2.

```
1           sudo http-server 80
```

Posteriormente, en los demás nodos se abre un *xterminal* y en cada una de las consolas de cada equipo se añade la ruta ARP a la dirección pública del servidor.

```
1   arp -s 10.0.0.100 00:00:00:00:00:64
```

Según si el nodo es un nodo legítimo o un nodo atacante se ejecutara un *script* de Python distinto. Si se trata de un cliente legítimo se ejecuta Traffic.py, si por el contrario es un atacante se ejecuta DDoS.py. Ambos archivos han sido programados para evaluar el sistema. Con todo esto ya estará conectada la topología simulada de Mininet con el controlador OpenDayLigth y todos los nodos simulados funcionando correctamente.

Finalmente tras completar cada prueba se puede observar la tabla OpenFlow del *switch* para analizar lo sucedido.

```
1   sudo ovs-ofctl -O OpenFlow13 dump-flows s1
```

Adicionalmente el código cuenta con un *thread* encargado de escribir en el *log* cada segundo todos los datos correspondientes de cada flujo del *switch* de modo que el fichero de *log* se procesa y se obtienen los datos de los flujos y los servidores.

```
1 awk '/ @/ {printf "%s,%s,%s,%s\n",$2,$9,$11,$13}' /home/jorge/Escritorio/log >
      /home/jorge/Escritorio/Resultados/packets
2
3 awk '/ #/ {printf "%s,%s,%s\n",$2,$9,$11}' /home/jorge/Escritorio/log> /home/
     jorge/Escritorio/Resultados/server_state
```