Bio-inspired routing algorithms for Software Defined Networking

Giovanni Cammarata

Department of Engineering
University of Catania

This dissertation is submitted for the degree of
Doctor of Engineering

University of Catania - Department of
Engineering November 2017

I would like to dedicate this thesis to my girlfriend and my sister ...

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 20,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 50 figures.

Giovanni Cammarata
November 2017

Acknowledgements

I would like to acknowledge Professor Antonella Di Stefano, Dr. Giovanni Morana, Dr. Daniele Zito, Dr. Rao Mikkilineni and the C3DNA Inc. (SANTA CLARA, US)

Abstract

In recent years, several projects have emerged with the aim of addressing the challenge of providing solutions for the deployment, monitoring, adaptation, and management of clouds belonging to different enterprises/organisations.
In addition, the concept of Green Computing and, more specifically, energy-aware solutions have gained attention in the last few years in many fields of ICT: network management is one of those.
Today's power consumption is considered a fundamental parameter to take into account when a new routing strategy is designed, just as with latency, bandwidth or error rate.
In such scenarios network connectivity is one of the key factors.
SDN technologies represent a good solution to face this challenge. By separating the control plane from the data plane, they can give the cloud provider the opportunity to specify both the network topologies and routing schemes on-the fly, guaranteeing, at same time, a specific level of isolation.
My work produced an algorithm for traffic engineering on an SDN based on the Alienated Ant Algorithm, a heuristic solution inspired by a non-natural behaviour of ants colonies.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

In recent years there has been a gradual and steady increase in the use of virtualization techniques that has led to the emergence of the concept of Software Defined Infrastructure , namely an infrastructure in which all elements (servers, storage, networking) are virtualized and made configurable and accessible via APIs.
Cloud Computing [22] and the concept of **“as-a-service”** solutions are the pillars of this new IT trend.
The flexibility in the management of the resources (both hardware and software) provided by the Clouds and, more recently, SDN [15] and virtual networking technologies [5] have completely modified the way applications are designed and, more important, managed.
The opportunity to select and modify, on the fly, both the hardware configuration (in terms of CPU/memory/disk size, IO disk throughput) and the number of machines hosting the applications, along with the ability to configure the load balancer, floating IP (usually used to reach the application/cloud instance from outside) and the integration of specific tools for continuous monitoring, allows for significant flexibility in the management of different levels of Quality of Service (QoS), i.e. different degrees of performance, safety, robustness and availability.
Network management plays a fundamental role in this scenario.
The performance of internet-accessible, geographically-distributed applications are in fact closely related to the performance of the underlying network as well as how they are related to the hardware performance of the machines on which they are hosted: the better the network performance, the better the overall performance of the applications.
Unlike in the management of hardware (whether virtualized or not), for which many solutions that reach and maintain the desired degree of performance already exist, the management of the network within the cloud environments still today poses considerable criticality.
Every network link is a "shared channel": it is subject to continuous interactions with entities

(communicating machines) that are independent (not coordinated by the same organisation), and parallel (simultaneous access to the network). If not properly managed, these interactions often cause significant degradation of the overall system performance and thus need to adopt complex management strategies.
Network Functions Virtualization (NFV) and Software-Defined Networking (SDN) technologies provide a new ways to build, configure and manage networks, improving the network's performance and making them predictable.
However virtualized network management has yet to be explored in greater detail.
The SDN solutions currently proposed in the scientific literature cover only the case of single cloud environments, where computing, storage and network resources are locally-distributed and owned by a single organization.
This aim of this work is to extend the concept of an SDN to build robust, reliable and performant networks across geographically-distributed resources owned by different providers, that can be competitive or federated.
After a brief overview of the SDN terminology and a review of the underlying technology 2, the framework that makes it possible to create, handle and monitor virtual networks spanning across several heterogeneous environments will be introduced (Chapter 3). The proposed framework, named ASSDN, offers an API to configure the routing and the performance-related network parameters independently from both the specific SDN technology adopted and from the peculiarities and heterogeneities of the network providers. It allows for the planning activities to be written in a high-level language (Java in this discussion) and to develop, maintain and easily change the routing algorithms.
It allows any cloud provider managing its own control plane functions to not only configure its own network but also to offer capabilities to make them accessible as a configuration service to the owner of the applications or even to other cloud providers, thus extending the concept of large-scale Network-as-a-Service.
Chapter 4 describes the A4SDN, a distributed, adaptive, load-balancing algorithm for traffic engineering on an SDN. A4SDN (A4 stands for Adaptive Alienated Ant Algorithm) is based on the Alienated Ant Algorithm (AAA), a stochastic-based, heuristic approach used to solve combinatorial and multi-constraint optimisation problems. Based on a non natural ants' behaviour, the AAA forces the ants in search of food to distribute themselves over all the available paths rather than converge to a single one.
A4SDN applies this strategy to the packets it routes, obtaining a load balancing solution that supplies an autonomic dynamic routing and leads to a better exploitation of the network bandwidth, enforcing best effort traffic and improving network performance.
The Chapter 6 proposes a comparison between A4SDN with two Dijkstra-based routing

solutions is also provided in terms of throughput, delay and packet loss rate; the Chapter 7 shows the results of this comparisons in different scenarios.
The Chapter 8 copes with one of the major challenges in ICT, i.e. the design of energy efficient solutions for network services. In the chapter eA4SDN (Chapter 8) is introduced, an energy-aware extension of the A4SDN. As any other AAA-based solutions, eA4SDN makes its decisions by minimising the value of the pheromone on the available paths, emulated in this case by the energy needed to manage the routing operations.
In order to evaluate its performance, the proposed energy-aware approach is compared with the standard A4SDN and with two deterministic solution based on Dijkstra's Algorithm.
Through preprocessing the traffic, the energy-saving models achieve better energy efficiency and reduce the energy consumption in SDN data centers while maintaining high levels of throughput, low delay and packet loss.
Conclusion is given in Chapter 9.

The popularity of the cloud and the variety of infrastructures and platforms widespread on the worldwide market has increased the need of networking resources and capabilities to manage them flexibly.
Control and orchestration of network capabilities will be a key factor for success in taming the complexity of an infrastructure executing millions of software transactions or service chains.
The opportunity to select and modify both the hardware configuration on-the-fly (in terms of CPU/memory/disk size, IO disk throughput) and the number of machines hosting the applications, along with the ability to configure the load balancer, floating IP and the integration of specific tools for continuous monitoring, allows for significant flexibility in the management of different levels of Quality of Service (QoS), i.e. different degrees of performance, safety, robustness and availability.
The performance of the distributed applications, in fact, are closely related to the performance of the underlying network and the hardware performance of the machines on which they are hosted: the better the network performance, the better the overall performance of the cloud application.
One of the most important requirements will be ensuring ultra-low application latency and ultra-high application throughput.
Load balancing plays a key role in computer networks. In this work was introduced a bio inspired mechanism, based on the well know ant colony optimization algorithm (ACO) [9], a probabilistic technique for solving computational problems which can be reduced to finding good paths through graphs. Assessments and feedback are described in more detail in Chapter 4.

Unlike in the management of hardware (whether virtualized or not), for which many solutions that reach and maintain the desired performance already exist, currently the management of the network within the cloud environments still presents considerable criticality. In such a scenario, the network resources play a key role.
Providing new ways to build, configure and manage networks, Network Functions Virtualization (NFV) [5] and Software-Defined Networking (SDN) [15] technologies allows for the improvement of the performance and guarantee the QoS of specific flows and, as a consequence, the ones of the specific distributed applications, making them predictable.
NFV and SDN are playing a key role in the rapid evolution of this wide-area. They are not mutually dependent, but both certainly can benefit both from a combined use.
In particular SDN is recognized as one of the most important paradigm shift from traditional networking towards the future of the Internet by providing new ways to build, configure and manage networks.
However this field has yet to be explored. Actually, there are few solutions that cover only the case of single cloud environments, where computing, storage and network resources are owned by a single organization.
My work extends these concepts to multicloud environments: it takes into account the issue to create, handle and monitor virtual networks that are able to span across several heterogeneous cloud environments, that can be competitive or federated, so as to guarantee a specific level of QoS.
My aim is to address this problem by creating a framework which offers an API to configure the routing and the performance-related network parameters independently from both the specific SDN technology adopted and from the peculiarities and heterogeneities of the network providers. It allows to write in a high-level language (Java in this Thesis) the planning activities and to develop, maintain and easily change the routing algorithms.
This allows any cloud provider managing its own control plane functions not only to configure its own network but also to offer capabilities for making them accessible as a configuration service to the owner of the applications or even to other cloud providers, thus extending the concept of large-scale Network-as-a-Service.
These will be addressed in Chapter 3.
The use of ICT technologies can contribute to environmental sustainability in many areas, for example through paper reduction and physical displacement of people. There are several initiatives aimed at improving the energy efficiency of data centers, although interest remains limited to large-scale organizations.
In Chapter 8 I will propose an energy-aware extension of the A4SDN. Through preprocessing

the traffic, the energy-saving models achieve better energy efficiency and reduce energy in SDN data centers while maintaining high levels of throughput, low delay and packet loss.

1.1 List of acronyms

A4SDN Alienated Ant Algorithm for Software Defined Networking
ASSDN Adaptive Strategies for Software Defined Networking
ACO Ant Colony Optimization
API Application Programming Interface
BFS Breadth First Search
eASDN Energy-Aware Routing in A4SDN
ICT Information and Communication Technologies
IP Internet Protocol
IT Information Technology
NFV Network Functions Virtualization
ONF Open Networking Foundation
QoS Quality of Service
SDN Software Defined Networking

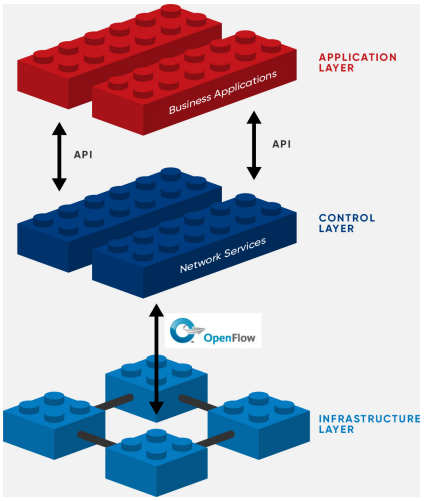
Chapter 2

SDN: Background and Motivation

2.1 Software Defined Networking: challenges and opportunities for future networks

According to the Open Networking Foundation (ONF) [sdn] , software-defined networking (SDN) is a networking architecture in which the control and forwarding (also know as data plane), which is responsible for the traffic forwarding, are decoupled.

Fig. 2.1 SDN Paradigm



The goal of SDN is to allow network engineers and administrators to respond quickly to changing business requirements.

In a software-defined network, a network administrator can shape traffic from a centralized control console without having to touch individual switches, and can deliver services to wherever they are needed in the network, without regard to what specific devices a server or other hardware components are connected to.

The key technologies for SDN implementation are functional separation, network virtualization and automation through programmability.

The SDN architecture is remarkably flexible; it can operate with different types of switches and at different protocol layers.

The network intelligence is logically centralised in software-based SDN controllers, which maintains a global view of the network, while the underlying network infrastructure is abstracted from the applications. As a result, the network appears to the applications (and policy engines, too) as a single, logical switch.

When a packet arrives at a switch in the network, rules built into the switch's proprietary firmware tell the switch where to forward the packet. The switch sends every packet going to the same destination along the same path, and treats all the packets the exact same way.
In a classic SDN scenario, the packet handling rules are sent to the switch from a controller, an application running on a server somewhere, and the switches (also known as data plane devices) query the controller for guidance as needed and provide it with information on the traffic they are handling. SDN controllers and switches can be implemented for Ethernet switches (Layer 2), Internet routers (Layer 3), transport (Layer 4) switching, or application layer switching and routing. The result is an extremely dynamic, manageable, cost-effective, and adaptable architecture that gives administrators unprecedented programmability, automation, and control. The communication, between the SDN Controller and the services and the applications running over the network, is allowed through the northbound APIs. The SDN controllers (e.g., OpenDayLight [27], FloodLight [13], Ryu [30], Pox/Nox [28], Beacon [3], Onos [25]) are able to apply suitable policies to map out the routing tables of the physical switches. Northbound APIs enables the networked to be programmed with basic network functions such as path computation, loop avoidance, routing, security and many other tasks. The SDN's northbound APIs are, currently, the most nebulous component in an SDN environment, because many different sets of northbound APIs are emerging.
The switches query the controller for guidance as needed, and provide it with information about traffic they are handling.
Controllers and switches communicate via a controller's "south bound" interface, usually OpenFlow [Specification], although other protocols exist.
OpenFlow is the first standard communication interface defined between the control and forwarding layers of an SDN architecture. OpenFlow allows direct access to and the manipulation of the forwarding plane of network devices such as switches and routers, both physical and virtual (hypervisor-based). In a classical router or switch, the fast packet forwarding (data path) and the high level routing decisions (control path) occur on the same device. An OpenFlow Switch separates these two functions. The data path portion still resides on the switch, while high-level routing decisions are moved to a separate controller, typically a standard server. The data path of an OpenFlow Switch presents a flow table abstraction. A flow table contains a list of flow entries. Each flow entry contains a set of packet fields to match, and an action (such as send-out-port, modify-field, or drop). When a packet arrives at an OpenFlow switch, the header fields are compared to flow table entries. If a match is found, the packet is either forwarded to the specified port(s) or dropped, depending on one or more actions stored in the flow table, otherwise it sends this packet to the controller. The

controller then makes a decision on how to handle this packet. It can drop the packet, or it can add a flow entry directing the switch on how to forward similar packets in the future.

OpenFlow-based SDN is currently being rolled out in a variety of networking devices and software, delivering substantial benefits to both enterprises and carriers, including:

- Centralized control of multi-vendor environments: SDN control software can control any OpenFlow-enabled network device from any vendor;
- Improved automation and management by using common APIs;
- Higher rate of innovation: SDN accelerates innovation through the ability to deliver new network capabilities and services without the need to configure individual devices or wait for vendor releases;
- Reduced complexity through automation: OpenFlow-based SDN offers a flexible network automation and management framework
- Increased network reliability and security as a result of centralized and automated management of network devices;
- More granular network control with the ability to apply comprehensive and wide-ranging policies at the session, user, device, and application levels;
- Better end-user experience as applications exploit centralized network state information to seamlessly adapt network behavior to user needs.

Chapter 3

ASSDN: Adaptive Strategies for Software Defined Networking

3.1 The ASSDN Reference Architecture

The architecture of ASSDN is shown in Fig. 3.1. It is composed by three layers:

1. The bottom layer consists of an SDN-enabled network, typically a set of connected OpenFlow switches;
2. In the middle layer the component *jFlowLight* [jflowlight] decouples the ASSDN's highest layer from the SDN Controller Northbound API, allowing us to support different SDN controllers without substantial switching costs or inconvenience;
3. The top level, the *ASSDN Orchestrator*, implements the policies of the routing strategies. It dynamically decides which flows to direct to the switches without downtime and service disruption.

3.1.1 JFlowLight

The SDN Northbound API, provided by the SDN Controller, is currently the most nebulous component in an SDN environment: many different sets of northbound APIs are emerging, typically one set for each controller.
Programming the SDN through different environments can be a complex challenge.
jFlowLight decouples the control from the Northbound API provided by the SDN-Controllers (see Fig. 3.1).
It offers a common high level view of differents SDN-Controllers in a multicloud environment.

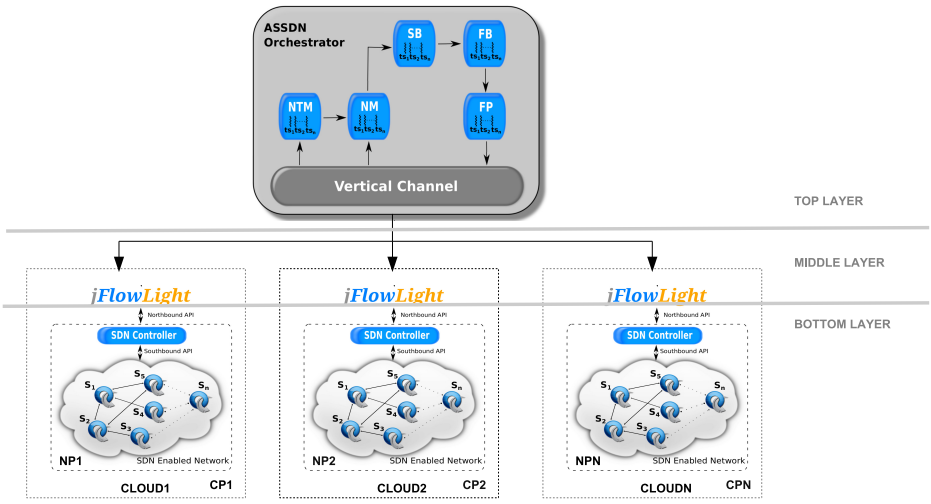


Fig. 3.1 ASSDN Architecture

jFlowLight allows for the creation of applications that are portable across the SDNs allowing users to program the network transparently from the specific REST-like APIs provided by each controller.

jFlowLight [jflowlight] is an open source library that helps users get started in an SDN. The jFlowLight API gives the freedom to create applications that are portable across SDN while giving full control to use SDN-specific features. It is an SDN-controller client that allows for the provisioning and control of an SDN deployment. This includes support for flows, statistics and topology. jFlowLight decouples the control from the northbound API provided by SDN-Controllers. The SDN northbound APIs (provided by the SDN-Controller) are, currently, the most nebulous component in an SDN environment, because many different sets of northbound APIs are emerging. Programming within SDN environments can be challenging. jFlowLight focuses on the mentioned issue so that users can get started without dealing with REST-like APIs provided by each controller. Users are allowed to manage a multitude of controllers without changing the APIs or their software.
jFlowLight gives full control of SDN specific features by providing a high level programming language.
JFlowLight has been as an opensource library. The binding language considered here is Java, but the same solution can be adopted for other languages.
The java language was chosen because it has a substantial community support, it is free, it is platform independent, and has excellent documentation. However I'm considering whether or not to release a REST-based API.

jFlowLight supports *OpenDayLight*, *Floodlight* and *Onos*. The architecture is designed to allow users to easily implement, modify, customize and enhance any SDN-Controller proxy (SDNCP) by using the plugins features. Through the use of the APIs provided by the SDN-Controller, generally REST-based, the SDNCP provides a single common API. Users can manage multitude of controllers without changing the API or their software. We are planning to extend the support to some community driven initiatives like OpenContrail [26], *Ryu*, *Beacon*.
The jFlowLight controller proxy allows users to fetch topological and statistical data and manage the flow entries. In substance it supports all the features provided by Openflow.
Here is the Java code for the controller creation:

```
Controller controller = ControllerFactory.builder().
            setControllerType("opendaylight").
            setAddress("192.168.1.59").
            setPort(8181).
            setUser("USER").
            setPwd("STRONG_PWD").
            build();
```

Below are some examples of the API usage to retrieve statistical values such as bytes sent/received/drop, packets sent/received/drop, adding, deleting and retrieving flow entries.

- **Topology operations**

  Getting all nodes (switches and hosts)

  ```
  controller.getAllNode()
  ```

  Getting all hosts

  ```
  controller.getHosts()
  ```

- **Statistics operations**

  Getting statistics

  ```
  controller.getNodeConnection()
  ```

- **Flow operations**

  Getting flow entries

  ```
  controller.getFlow(
  switchID,
  tableID)
  ```

Getting a flow entry

```
controller.getFlow(
switchID,
tableID,
entryID)
```

Adding a flow entry

```
controller.addFlowEntry(
flowname,
switchID,
priority,
tableID,
entryID,
ipv4Srcs,
ipv4Dsts,
ipBit,
macSrcs,
macDsts,
outputPort)
```

Deleting a flow entry

```
controller.deleteFlowEntry(
switchID,
tableID,
entryID)
```

The jFlowLight [jflowlight] wiki provides a better explanation about the architecture and features offered.

3.1.2 The ASSDN Orchestrator

The *ASSDN Orchestrator* (see Fig. 3.1), is the key component. It dynamically implements the routing strategy by optimizing the network traffic.
It consists of five multithreaded components:

- the **Network Topology Mapper (NTM)**. It discovers and maps the network devices, such as switches, hosts and links. It automatically detects new devices and changes

in network topology. It maintains information about the entire graph representing the network infrastructure.

- the **Network Monitor (NM)**. It collects the real-time data about network performance/statistics and stores them for future analysis, collecting the network monitoring history.
- the **Strategy Builder (SB)**. It is the main component of the framework since it allows users to develop their own custom routing strategies.
- the **Flow Builder (FB)**. It is responsible for computing and validating routes.
- the **Flow Pusher (FP)**. It carries out the proactive OpenFlow *flow-rule* insertion.

3.1.3 The ASSDN Execution Phases

The ASSDN performs a cyclical execution of the dynamic workload balancing algorithm. Each cycle, called *round*, consists of four stages:

- **Snapshot**. This stage includes: i) the fetching by the SDN controller of the statistics about the traffic on all the links (switch-to-switch and switch-to-leaf); ii) the aggregation of these statistics.
- **Path Evaluation**. It is the the algorithm, or pool of algorithms, defined by customers in order to adapt the traffic according their own custom policies.
- **Flow creation**. This stage defines flows according to the result provided by the previous stage.
- **Flow push**. It is the final stage. The forwarding instructions are based on the concept of flow. The criteria for defining a flow includes the subnet source/destination IP address and the output port.
  Supposing a path from node *A* to node *F* were to be installed, where the full path is *(A - B - C - D - E - F)*. The definition of the flow from A to F requires the installation of 5 flow entries as shown in table 3.1. The flow creation is done in reverse order: it starts from the node closest to the destination, flowing towards the source node; for the considered example the installation goes from E to A *(E - D - C - B - A)*, excluding loss of traffic.
  Conversely, the removal of a previous old flow (generated by a previous run) will occur starting from switch A to E *(A - B - C -D - E)* excluding any loops or loss of traffic.

Table 3.1 Flow Push results

OPENFLOW SWITCH	SRC IP	DST IP	OUTPUT PORT
A	10.0.1.1/24	10.0.4.1/24	1
B	10.0.1.1/24	10.0.4.1/24	3
C	10.0.1.1/24	10.0.4.1/24	2
D	10.0.1.1/24	10.0.4.1/24	3
E	10.0.1.1/24	10.0.4.1/24	3

The ASSDN framework can be easily extended by creating its own custom routing strategy. Users can run it simply by putting the plugin into the framework by exploiting the offered APIs.

3.1.4 The ASSDN APIs and Configuration

The ASSDN framework provides APIs to help users create their own strategy without taking into account the underlying SDN technology and the network topology.

It expose some informations, about the underlying network, such as:

- network graph
- data link's capacity and delay
- statistics for switches
- statistics for hosts
- statistics for links
- flow rules installed

Users have only to decide the routing strategy by implementing the high level function that we call *calculatepath* through an interface which exploits our provided APIs.

```
function calculate_path(graph, stats, src, dst){
    // CUSTOM CODE HERE
    // Return the path computation
}
```

This component is the core of the *Strategy Builder*.
The output of *calculate path* is a list of hops to reach the *dst* from the *src*.

The framework validates the output provided in order to avoid downtime and service disruption.
It accepts, at runtime and without downtime, some configuration parameters such as sampling period and what strategy to use.
Chapter 6 gives a better understanding of how to use the APIs provided and the implementation of a custom strategy

Chapter 4

A4SDN: Alienated Ant Algorithm for Software Defined Networking

4.1 A4SDN

The Alienated Ant Algorithm for Software Defined Networking (A4SDN) introduces a dynamic routing approach to the ASSDN framework, already presented in Section 3.1.
Here we introduce our own algorithm exploiting an ACO-based approach. We implement the **Pheromone Evaluator (PE)** by extending the **Strategy Builder (SB)**. It is responsible for the pheromone evaluation according to the AAA algorithm.
It evaluates the pheromone quantity of each OpenFlow switch based on the information collected by the Network Monitor.
The architecture of A4SDN is shown in Fig. 4.1.
A4SDN aims to improve the end-to-end latency and throughput in the SDN-based cloud infrastructure by extending the Alienated Ant Algorithm (AAA) [1, 7, 2].
The AAA is a heuristic approach based on a *non-natural* behaviour of ants, where the ants spread out over all available paths rather than be forced to converge on a single one (as it happens in traditional ACO)
The main characteristics of A4SDN that makes it different from the ACO-based routing algorithms are two:

- the opposite interpretation of the pheromone trails;
- the sub-path pheromone evaluation.

The first characteristic guarantees the load-balancing capability. In fact, the alienated ants smell the trails of pheromone and, instead of following the path where it is strongest and

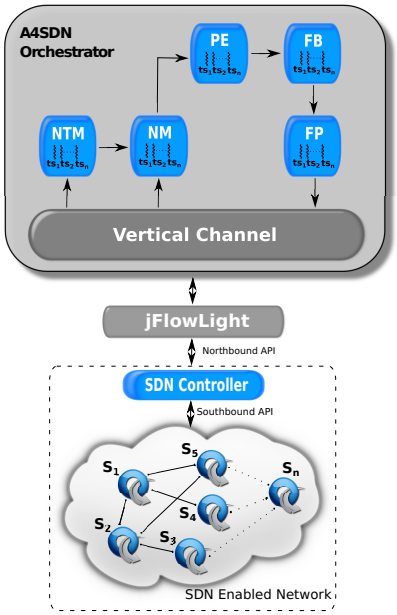


Fig. 4.1 A4SDN Framework

converging on it, they follow the path where the pheromone is weakest: the latter, due to the release-evaporation process, changes over time

The second characteristic, instead, avoids the scalability and convergence issues of the ACO-inspired algorithms. The ACO algorithms, in fact, could require a long time to converge to a single path because they require multiple iterations over all the available paths from the source to the destination [1].

A4SDN makes its decision based on the pheromone laid on the sub-paths (i.e the links between each switch and the next ones): this makes it suitable for on-line decision problems such as the routing (or scheduling) one. Moreover, A4SDN considers only a subset of possible paths, speeding up the evaluation process.

4.1.1 Ant Colony Optimization algorithms

Starting from the late 90's, many algorithms ([9, 16, 12, 4, 31]) have been developed, in different scenarios, in order to simulate the self-organization ability of ants so as to use this form of shared intelligence in a multitude of ways. This led to the definition of a well-structured class of stochastic, population-based meta-heuristics algorithms, known as Ant Colony Optimization (ACO), mainly used to solve combinatorial and multi constraint

optimization problems.
To do this, first the optimization problem is transformed into a problem of finding the best path on a weighted graph; then a set of software agents, called *artificial ants*, incrementally build solutions by moving on a graph imitating their biological counterpart in the natural world, i.e. trying to maximize a given metric expressed in terms of pheromone quantity.
All ACO algorithms follows a simple three-steps pattern: Selection, Reinforcement and Evaporation. The Selection step consists of the selection of a set of candidates among the available paths which connect the nest to the food source, based on a probabilistic function. This evaluates the pheromone on each node and assigns it a probability to be selected proportional to the quantity of associated pheromone.
Once a path is identified, each ant marks it, thus reinforcing the current pheromone trail. Through this reinforcement each ant increments the pheromone level only on one path, increasing the probability of the same path being chosen again in the next iteration. This simple mechanism guarantees the convergence of all ants on the paths that optimize the distance between the nest and the food.
Finally, the Evaporation mechanism consists in the progressive decrease of the pheromone over all the paths proportionally to the elapsed time. Forcing the reduction of the pheromone level in all paths makes it possible to completely extinguish the less travelled ones. As a consequence, the paths where the pheromone is stronger are highlighted. These are the best paths and often it is a unique path.
Notice that ACO based algorithms are particularly suitable and easy-to-apply for all those scenarios where the conditions over the paths (e.g. the cost, the weight) are fixed or, at most, they change slowly. This is mainly due to its selection mechanism (that forces the convergence to a unique path) and its off-line pheromone updating method (that needs to know all results before choosing the best one).

4.1.2 The Alienated Ant Algorithm

As discussed, ACO works well in graphs where the topology and/or the load among the nodes are fixed.
Conversely, in scenarios where these conditions change quickly and unpredictably, ACO based solutions often require additional mechanisms that increase the complexity of the algorithm. This is true especially for flow management in time-variant networks where, in general, the path representing the solution changes over time with a high degree of unpredictability. In these cases, an efficient load distribution and run-time adaptation is needed.
AAA is an ACO derived algorithm introduced to tackle these issues, based on a different

interpretation of pheromone trails. In ACO it is an aggregator while in AAA it is a repulsive substance, something that decreases the probability that a path will be chosen in the future. By means of this assumption, the authors obtained a behavior that was complementary to the ACO approach.
AAA, in fact, takes into account the behavior of an ideal alienated ant. Such an ant prefers the paths where it can find the fewest ants. In order to achieve this, it smells the pheromone trails but, instead of following the path where the pheromone trail is stronger (as the other normal ants), it takes the path where the trail is weakest. Therefore, marking the path with an additional quantity of pheromone is meant to reduce the probability of it being selected by other alienated ants.
At the same time, conversely to what classical ACO approaches do, the AAA approach uses the evaporation to recreate the equilibrium among all the paths giving them a chance to be selected again.
Such strategy allows the AAA:

- to rapidly explore all of the graph and perform dynamic load balancing among all its edges;
- to provide a reasonable response time related to the routing among the edges;
- to be able to adjust itself as the edges, load and network conditions change.

In [2] the authors demonstrated that AAA has a strong distributive ability, especially within highly dynamic environments. For this reason it is already successfully applied to grid and cloud systems.
This is the first time it has been applied to SDN environments.

4.1.3 The A4SDN Algorithm

We also discuss the proposed Adaptive Alienated Ant Algorithm for Software-Defined Networking A4SDN [6], a distributed load balancing algorithm for traffic engineering on SDN, based on the bio-inspired AAA.
The choice of a bio-inspired algorithm is not accidental: many approaches to problem solving are inspired by the social behaviors of insects and other animals. In particular, ants have inspired a number of methods and techniques among which the most studied and successful are the general purpose optimization techniques based on ACO.
The ACO-based algorithms have been successfully applied in many real world scenarios like internet and telecommunication traffic routing management techniques [34, 10, 11, 29].

They are often used to find the minimum path between two nodes within a network; by utilizing them it is possible to optimize the routing performance of a network, especially in term of communication delay. On the other hand, their strategies do not resolve the load balancing issues. For this reason, the authors have switched to the Alienated Ant Algorithm solution. It is a heuristic algorithm freely inspired by the Ant Colony Optimization (ACO) [9, 33] strategy that exploits a *non-natural* behaviour of ants, by means of an inverse interpretation of pheromone trails
Conversely to the ACO's traditional routing schemes that route all traffic along a single path the AAA routing strategy splits the traffic among several paths in order to ease congestion, increase the network traffic and achieve network loading balance thus reducing the likelihood of congestion.
Thanks to its features the AAA has been effectively implemented in the scheduling management in different distributed computing scenarios, by providing an online distributed scheduling solution for a dynamic distributed system.
A4SDN is the first architecture that uses a bio-inspired approach on an SDN system. It leverages the AAA to dynamically redirect the traffic on the paths, according to specific policies of load balancing.
By considering the packets as ants and the load on network switches as pheromone, it is possible to optimise the network performance in terms of throughput, communication delay and packet loss rate redirecting the traffic flows to the appropriate path, i.e. the least loaded one based on the current network state.

Classical Dijkstra based algorithms and their dynamic adaptations are not well suited for networks with a high number of nodes and edges with such computational complexity. Dijkstra's original algorithm does not use a min-priority queue and runs in time: $O(|V|^2)$ (where $|V|$ is the number of nodes). If the input graph is represented using an adjacency list, it can be reduced to $O(|E| + |V|log|V|)$ (where $|V|$ is the number of nodes and $|E|$ is the number of edges) with the help of a binary heap. This implementation is based on a min-priority queue implemented by a Fibonacci heap [14].
The intrinsic parallel nature of A4SDN drastically reduces the time complexity: for each node, the algorithm has a time complexity equal to $O(|E'|)$, with $|E'| << |E|$, where E' is the number of a sub-set of the node's edges.
Accessibility is always guaranteed without the use of loops.

The A4SDN algorithm consists in the cyclical execution of the AAA. At every cycle, called *round*, the four stages are executed to balance the workload among the switches/nodes.

Here we replaced the *Pheromone Evaluation* with the *Path Evaluation*.
The Pheromone Evaluation is the main stage of the algorithm. It evaluates, for each switch, the "the distribution of traffic forwarded on the output links for the next cycle", based on data collected in the snapshot stage.
The node pheromones are given by the traffic volume (incoming plus outgoing).
The pheromone P_n is defined as:

$$P_n = (Rx_n) - (Tx_n + Dx_n) \tag{4.1}$$

where n is the round number, Rx_n , Tx_n and Dx_n represent, respectively, the number of bytes received, sent and dropped as evaluated and aggregated in the snapshot stage at time t_n. The difference between the two last sampling instants avoids the issues on the pheromone estimation related with its time fluctuation (e.g. controller/network overload). The amount of pheromone of each switch is affected by the OpenFlow protocol itself (e.g. Packet-IN, Packet-OUT). In addition, all the traffic generated on the network is strongly dependent on both the topology and degree of coupling.
Here is the pseudocode for the evaluation function:

```
function evaluate_pheromone() {
https://elasticbox.com/explore   foreach switch in switches{
                    thread(evaluate_pheromone(switch))
        }
}

function evaluate_pheromone(switch){
        $P_{switch}$ = $Rx_{switch}$ - ( $Tx_{switch}$ + $Dx_{swi
        if ($P_{switch}$ < 0)
                $P_{switch}$= 0

        return $P_{switch}$
}
```

The Flow creation stage defines how traffic is distributed over the network and how the traffic is re-routed in the event of load balancing. It is performed step by step by evaluating the amount of pheromone on each switch node. Notice that we bind the resulting set of computed paths with a bound the maximum length, discarding the paths that can be too expensive and thus guaranteeing a specified delivery time limit.

Table 4.1 Flow Creation results

SWITCH NODE	PHEREMONE	IP ADDRESS
A	5	10.0.1.1
B	1	10.0.2.1
C	8	10.0.3.1
D	9	10.0.4.1
E	4	10.0.5.1
F	7	10.0.6.1
G	10	10.0.7.1

Suppose we have the following pheromone information after n snapshots ($Snap_n$). Table 4.1 shows the pheromone information for a certain snapshot t.

All the possible paths from A to F are listed here:

1. A – B – C – D – E – F
2. A – B – C – D – E – G – F
3. A – B – C – D – F
4. A – B – D – E – F
5. A – B – D – E – G – F
6. A – B – D – F
7. A – C – B – D – E – F
8. A – C – B – D – E – G – F
9. A – C – B – D – F
10. A – C – D – E – F
11. A – C – D – E – G – F
12. A – C – D – F

The AAA builds the routing path based on the pheromone evaluation and selects, at each stage, the sub-path with the lowest pheromone value.
Starting from A, the next-hop candidates are B and C: since B's pheromone level i less than C's, the sub-path toward B will be chosen: the first link selected is A - B and the number of feasible paths are now reduced from the initial 12 to 6:

1. B – C – D – E – F
2. B – C – D – E – G – F
3. B – C – D – F
4. B – D – E – F
5. B – D – E – G – F
6. B – D – F

At the next hop the selected link will be B-C, since C's pheromone level is less than D's. The feasible paths are now reduced to these three:

1. C – D – E – F
2. C – D – E – G – F
3. C – D – F

The sole candidate for the next hop is D and, after, between E and F, AAA selects E. At the last hop, between the candidate F and G the choice is F.

1. E – F
2. E – G – F

The full path (A - B - C - D - E - F) is completed.
To avoid fluctuations was considered the opportunity to evaluate an alternate path, only if the standard deviation of the switch's traffic (in bytes) exceeded a minimum threshold (configurable at run-time). When the routing path has to be modified, the algorithm applies the Flow Push (stage 4) described in the following:

```
function calculate_path(source,destination,visited){

        if (source == destination)
                        return visited

        // Initialize "hopset" as a set of possible hops
        // from source to destination

        foreach hop in hopset{
                if ("hop" has a minor pheromone AND
                        "visited" NOT contains the "hop")

                        nexthop = hop
                }
        }

        calculate_path(nexthop,destination,visited)
}
```

At this point the ASSDN framework will regain control and execute the *Flow Creation* and *Flow Push* phases (see Section 3.1).

Chapter 5

ASSDN: use cases

5.1 ASSDN: use cases

Below we show how to model some common routing algorithms by exploiting the ASSDN framework. We start from some basic use cases, which do not use all the capabilities offered by the ASSDN framework, and then we present some more complex routing algorithms. After presenting a static BFS routing algorithm we model the classical Dijkstra algorithm [8]. Finally we show how it is possible to build dynamic routing solutions by introducing the A4SDN algorithm [6].

5.1.1 BFS

The Breadth First Search (BFS) [23] is a graph traversal strategy for search of the shortest path, processing vertices in ascending order relative to their distance from the root vertex. BFS is optimal and is guaranteed to find the best solution. It is a variation of a static routing algorithm which is more practical than the one for the selection of the shortest path based on of the number of hops.

```
function calculate_path(graph, stats, src, dst){
    // create empty queue Q
    Q.enqueue(src, [src])

    while Q is not empty:
        (vertex, path) $\leftarrow{}$ Q.pop(0)
        for next in graph[vertex] - set(path):
            if next = dst:
```

```
                return path + [next]
            else:
                Q.append((next, path + [next]))
}
```

5.1.2 Dijkstra

Here we show the implementation of the well-known Dijkstra algorithm, which finds the shortest paths from a source vertex to all the other vertices in a graph when all edges have non-negative weights. Using the Dijkstra algorithm, it is possible to determine the shortest distance (or the least effort/lowest cost) between a starting node and any other node in a graph excluding the longest distances. Here, we suppose that the link weights reflect the bandwidth for each link, that all the link weights are positive and that a smaller link weight means wider bandwidth.

```
function calculate_path(graph, stats, src, dst):

        dist[src] $\leftarrow{}$ 0
        prev[src] $\leftarrow{}$ undefined

        // create vertex set Q
           for each vertex v in graph:
            if v $\neq{}$ src:
               dist[v] $\leftarrow{}$ $\infty$
               prev[v] $\leftarrow{}$ undefined
           add v to Q

       while Q is not empty:
           u $\leftarrow{}$ calculate_hop(graph, stats, Q, u)
           remove u from Q

           u $\leftarrow{}$ target

           path $\leftarrow{}$ empty sequence

           if u = dst:
             return path
```

```
            while prev[u] is defined:
              insert u at the beginning of path
              u $\leftarrow{}$ prev[u]
              insert u at the beginning of path
}

function calculate_hop(graph, stats, Q, u){
   return the vertex in Q with min dist[u]
}
```

5.1.3 A4SDN

The Adaptive Alienated Ant Algorithm for Software-Defined Networking (A4SDN) is a distributed, adaptive, load-balancing algorithm for traffic engineering on Software-Defined Networks.
Alienated Ant Algorithm belongs to the ACO (Ant Colony Optimisation, [9, 33]) algorithms class and it leverages a *non natural* behaviour of an ant.
Unlike standard ant-based solutions where the ants searching for food converge to a single path, it forces the ants searching for food to distribute themselves over all the available paths. The alienated ant, as opposed to a natural one, chooses the path based on the pheromone trails but, instead of covering the path where it is strongest, as a natural ant does, it explores the path where the pheromone is the weakest (searching for food were the other do not go).

The algorithm was explained more in depth in chapter 4.

Chapter 6

Case Study

6.1 Case Study

To evaluate the effectiveness of our proposed adaptive solution, we compared it with two Dijkstra-based algorithms. The tests that have been carried out show that ASSDN, and A4SDN, is able to guarantee a higher throughput associated with a lower delay and packet loss rate.

6.1.1 Reference Scenario

In this section we compare the performance of A4SDN with Dijkstra's algorithm (Dijkstra's algorithm, DA) and an extended, dynamic, version of it (Extended Dijkstra's algorithm, EDA).
Both of these solutions base the creation of the routing tables on the research of the shortest path among each couple of switches available in the network: the first one calculates the shortest path based on the links' bandwidth only when the network is established while the second one calculates it cyclically, while also taking into consideration the actual load on the links as the A4SDN does.

Fig. 6.1 Reference Scenario (image from [Internet 2 L3.])

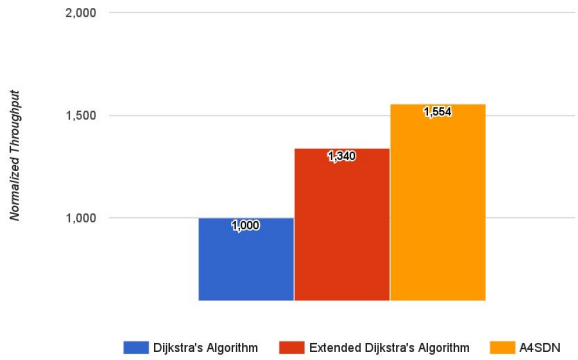


Fig. 6.2 Internet 2 Throughput

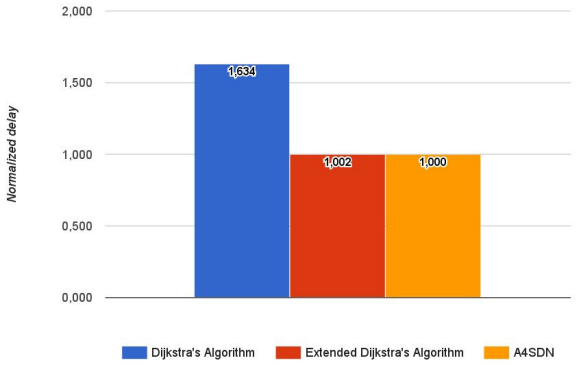


Fig. 6.3 Internet 2 Delay

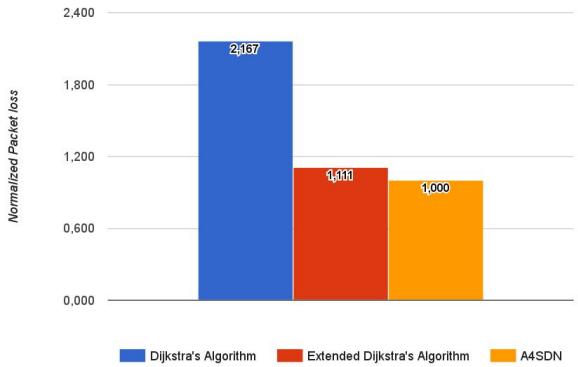


Fig. 6.4 Internet 2 Packet Loss

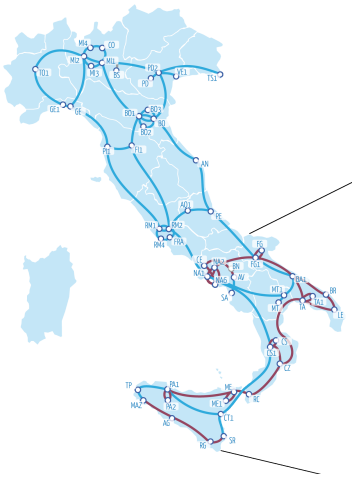


Fig. 6.5 Garr-X Topology

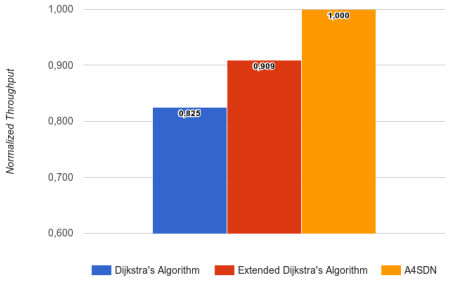


Fig. 6.6 GarrX Throughput

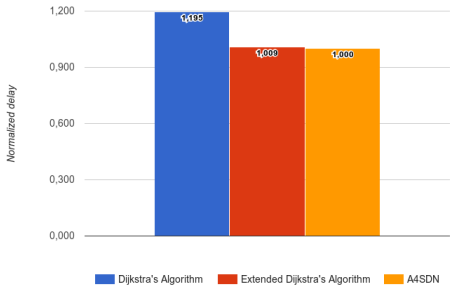


Fig. 6.7 GarrX Delay

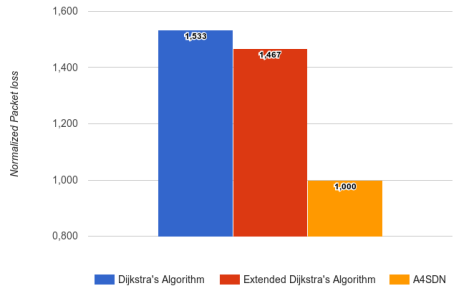


Fig. 6.8 GarrX Packet Loss

Table 6.1 Internet 2 testbed

Link Bandwidth	10 Mbps
Link Propagation Delay	10 ms
Number of servers	2
Number of switches	10
Number of edges	15
SDN Controller	OpenDayLight 0.2.3 Helium-SR3
SDN enabled network	Mininet 2.2.1
OpenFlow version	1.3.0
jFlowLight version	0.9.1
Iperf version	3.0.7
Ping	-
Testing time	3600 sec

Table 6.2 GarrX testbed

Link Bandwidth	10 Mbps
Link Propagation Delay	10 ms
Number of servers	1
Number of switches	26
Number of edges	33
SDN Controller	OpenDayLight 0.2.3 Helium-SR3
SDN enabled network	Mininet 2.2.1
OpenFlow version	1.3.0
jFlowLight version	0.9.1
Iperf version	3.0.7
Ping	-
Testing time	3600 sec

The traffic used for the evaluation has been generated by using Iperf [18] and configured to overload the network.
All the algorithms have been implemented using OpenDaylight as an SDN controller and Mininet [21] to create a realistic network composed by hosts, links, and switches.
In particular, the A4SDN Orchestrator has been implemented atop OpenDaylight, interfacing them through jFlowLight 0.9.0 [jflowlight] has already been introduced in section section 3.1.
In the comparison, were considered three parameters: throughput, delay and packet loss rate. The network latency and the packet loss were measured by using *ping* to send packets from the clients to the servers for 3600 seconds; the throughput was measured using the Iperf *bandwidth measurement* tool.

6.1.2 Internet 2

The A4SDN and the Dijkstra-based algorithms are evaluated over the Internet2's Advanced Layer 3 Service topology [Internet 2 L3.] (Internet2's IP Network) (see Fig. 8.1), i.e. an SDN composed by 1 controller and 10 OpenFlow switches.
The Internet2 Network connects over 60,000 U.S. educational, research and governmental institutions, ranging from primary and secondary schools to community colleges and universities, public libraries and museums to health care organizations.
It uses optical fiber that delivers network services for research and education purposes, and provides a secure network testing and research environment.

Tab 8.1 shows the value of the parameters considered in the test environment.

6.1.3 GARR-X

The GARR-X is the project for a next-generation multi-service telecommunication network for the Italian Academic and Research community. This network shall gradually replace the existing infrastructure, *GARR-G*.
The backbone is based on high-bandwidth circuits. The meshed topology (see Fig. 6.5), interconnecting 45 network Points of Presence (PoP), ensures a high level of resilience and reliability of the network; additionally, thanks to its wide coverage of the country, it interconnects more than 400 user organizations all over the national territory.

Tab 6.2 shows the value of the parameters considered in the test environment.

6.1.4 Results

In both topologies we obtained similar results. In the case of Internet2, as shown in Fig. 8.4, the throughput of A4SDN is more than 55% higher that the DA's and about 16% higher than the EDA's. Instead, in the case of GARR-X (Fig. 6.6), A4SDN has a throughput 17% higher than the DA's and about 9% higher than the EDA's.
This result is a consequence of the ability of A4SDN to exploit different network paths and to provide more effective network bandwidth thanks to the adaptive load balancing.

Figs. 8.5 and 6.7 show the average delay for each transmitted packet: in this case the performance of A4SDN and EDA is very close (A4SDN is 0.2% better than EDA in the case of Internet2 and 1% in the case of GARR-X) and both are better than DA. The worst performance of this last case can be explained with the fact that it is static and not able to react to congestions: when a path tagged as "shortest" becomes congested, the DA is not able to modify it, continuing to push packets onto it and feeding the congestions. EDA, instead, is able to react to congestion by searching for a new shortest path while the approach adopted by A4SDN tries to avoid, by using multiple paths, the creation of a congestion.
The last parameter evaluated is the packet loss rate. As shown in Figs. 6.4 and Fig. 6.8, the number of packets lost measured using EDA is 11% and 47% higher than A4SDN's, while the one measured using DA is more than twice as high as the proposed approach.
As for delays, this result is mainly related to the ability of A4SDN to better manage congestions, which represent, in this scenario, the only cause of packet loss.

Chapter 7

Performance Evaluations

7.1 Performance Evaluations

The performance of A4SDN are influenced by the underlying network graph.
In order to evaluate the degree of network-performance correlation, we created six networks (see Fig. 7.1) that have the same number of nodes (16) but that differ by the number of links, by the length of the longest path and by the average number of available shortest paths for each couple of nodes in the network.
The characteristic of each network is summarised in the table 7.1.

The Fig. 7.2 shows the performance of A4SDN in terms of throughput (MB/s), packet loss (%) and delay over the different networks subject to the same load.
The figure clearly shows that when the number of links in the network (i.e. the ratio links/nodes) increases, the throughput increases. This can be explained with the fact that the increment of the number of links causes an increment of the number of neighbours for each node: when this increases, the number of available paths for the ants increases making them able to find more alternative paths from source to destination.
The increment of the links is also correlated with a decrement of the average delay, directly

Table 7.1 Networks' parameters

links/nodes	links	Neighbours	Longest path	avg nb. of shortest paths
1.125	18	2.25	6	1.25
1.500	24	3	6	3.75
1.688	27	3.38	5	1.75
1.875	30	3.75	4	1.25
2.000	32	3.85	4	1.25
2.250	36	4.75	3	2.25

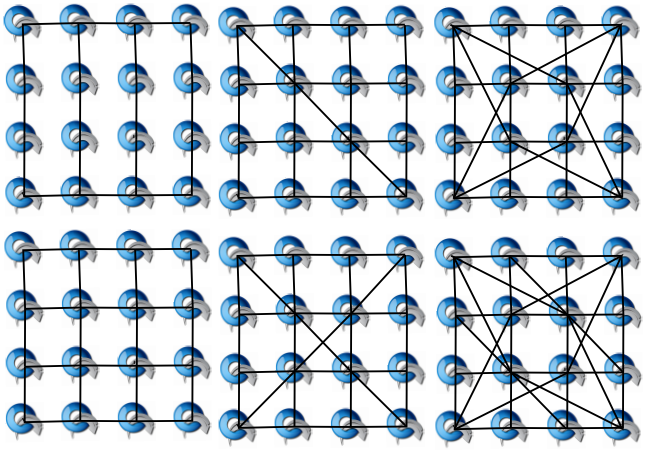


Fig. 7.1 16 nodes topologies

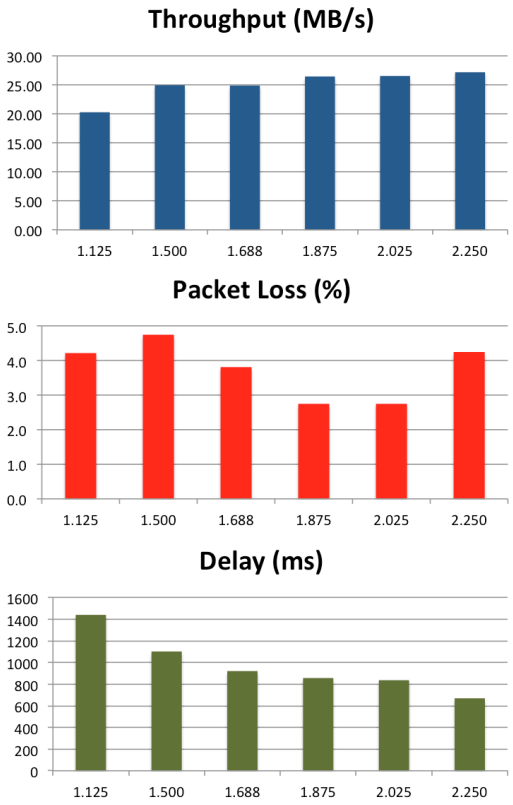


Fig. 7.2 The performance of A4SDN on different networks

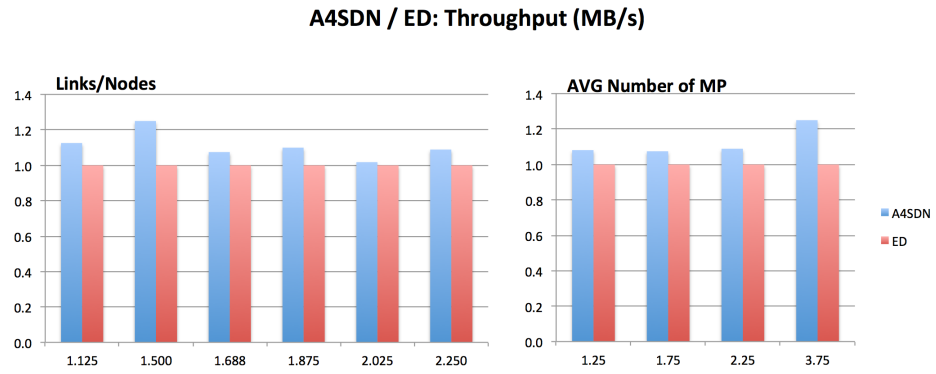


Fig. 7.3 Throughput: Comparison between A4SDN and ED

related to the reduction of the longest path as it shown in the table 7.1.

The packet loss, instead, seems to be independent by the variation in networks parameters. We also compared the performance of the A4SDN algorithm to the ones of the Extended Dijkstra approach for each considered network (See the section 5.1.2).
The results of these comparisons is shown in the figures 7.3, 7.4,7.5.
Fig. 7.3 shows how the throughput varies when both the links/node ratio and the average number of shortest paths per node increase. In particular, the throughput seems to be independent by the first parameter (i.e. the links/nodes ratio) but it is correlated with the average number of shortest paths per node: the greater the number of shortest paths, the bigger the advantages in using the proposed approach instead of the Extended Dijkstra one.
This behaviour can be easily explained: even if the ants try to cover different paths, a big number of available shortest paths increases the probability that a big number of ants uses one of them. Simplifying, if only 1 out of 4 is a shortest path, probably only the 25% of the ants will cover it. If, instead, 3 out of 4 are shortest paths, then the 75% of the ants will cover one of them.

The A4SDN has better performance in term of packet loss in all the considered networks except that in the last one (both links/nodes and average number of shortest paths equals to 2.25). Notice that also in this case, the best performance, i.e. a difference between A4SDN and ED higher then the 50%, is measured in the network having the biggest value for the average number of shortest paths per nodes.

The ED approach, as shown in Fig. 7.4 has instead better performance in terms of delay in all the considered network except for the first one (links/nodes equals to 1.125 and average number of shortest paths equals to 1.25). This difference on delay can be explained

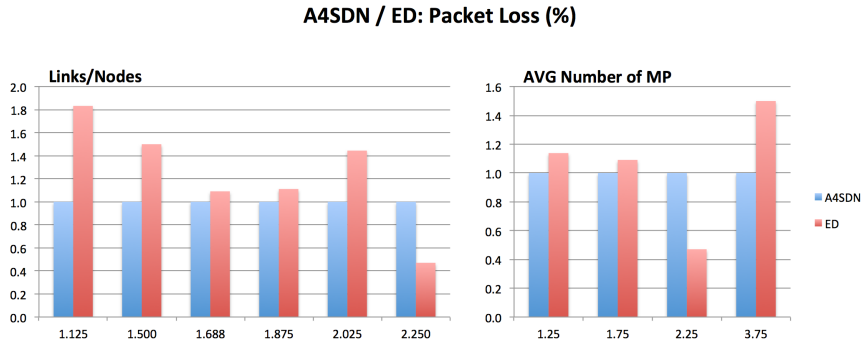


Fig. 7.4 Packet loss: Comparison between A4SDN and ED

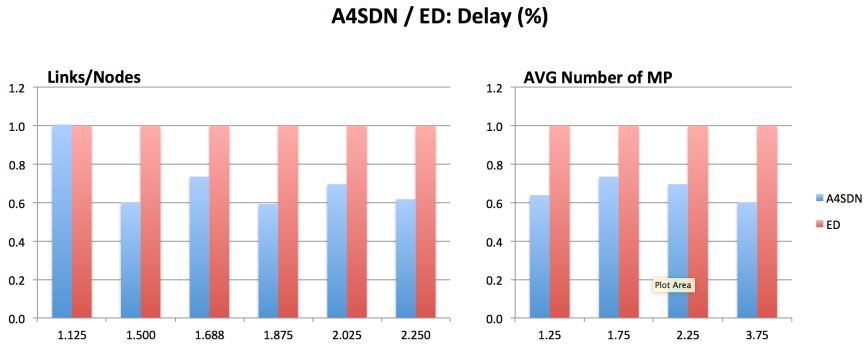


Fig. 7.5 Delay: Comparison between A4SDN and ED

considering that the path chosen using the ED approach is always the shortest one while the A4SDN can use longer ones: the exception is due to the particular structure of the considered network that forces both routing algorithms to adopt very similar solutions.
The simulations led show clearly that on the network controlled by A4SDN, when the number of links increases, the average bandwidth increases, while the average delay decreases. Moreover, if the average number of available shortest paths for each couple of nodes rises, the performance of A4SDN are the best.

Chapter 8

Energy-Aware Routing in A4SDN

The concept of Green Computing and, more specifically, energy-aware solutions have gained more and more attention in the last years in many fields of ICT: network management is one of those.
Today, power consumption is considered a fundamental parameter to take into account, as well as latency, band-width or error rate, when a new routing strategy is designed.
In the last few years, the volume of data and the variety of internet applications have become a rather serious problem.
In an article titled "Power, Pollution and the Internet" [24], the New York Times pointed out that energy consumption of Data Centers is close to 30 billion watts worldwide, the equivalent combined the equivalent output of 30 nuclear power plants.
There are many levels of communication, internally and externally, to and from the Data Centers (DCs). Ensuring that this communication happens seamlessly, efficiently and in a secure manner is a critical role of the network that ties all these components together.
The energy consumption of the networks cannot be ignored and how to save energy has become a meaningful endeavor. As the provider of computing, storage and various other services, DCs play vital roles in the networks. In order to guarantee the reliability of the DC, there are often a lot of redundant switches and servers which waste huge amounts of power.
Through the preprocessing of the traffic, the energy-saving models achieve better energy efficiency and reduce the energy consumption in SDN data centers while maintaining high levels of throughput, low delay and packet loss.
Through these considerations we introduced an energy-aware extension of the A4SDN and, in order to evaluate its performance, the proposed energy-aware approach is compared with the standard A4SDN and with two deterministic solution based on Dijkstra's Algorithm.

8.1 Energy Model

The energy model considered was introduced by Kaup et al. in [20], where it is described in details.
The authors provide a model for the power consumption of SDN-enabled networking devices, with a specific focus on two devices, i.e. an OpenFlow-based hardware switch and a server running Open vSwitch.
The model, which considering of all the operations related with the routing process (not limited only to packet forwarding but including the ones for switches configuration and management), is able to approssimate the consumed energy with an error of less than 8% for the software switch and less than 1% for the hardware switch.
According to that model, the power consumption of a switch, here defined as P_{switch}, is:

$$P_{switch} = P_{base} + P_{config} + P_{control} + P_{OF} \tag{8.1}$$

where:

- P_{base} is the static power needed to keep the device active;
- P_{config} is the power used by the assigned configuration, i.e. related with the number of active ports or with the configured line speed;
- $P_{control}$ is the power needed to control the network traffic, i.e. the packets involved in the network management;
- P_{OF} is the power consumed by the traffic processed by OpenFlow.

Going into details, the P_{config} is:

$$P_{config} = \sum_{i}^{N_{activePorts}} s_i \cdot P_{port} \tag{8.2}$$

where:

- $N_{activePorts}$ is the number of active ports;
- s_i is a value proportional to the configured speed of the port;
- P_{port} is the power consumption of the i port at full speed.

The $P_{control}$ is:

$$P_{control} = r_{packetIn} \cdot E_{packetIn} + r_{FlowMod} \cdot E_{FlowMod} \tag{8.3}$$

where:

- $r_{packetIn}$ is the rate of outgoing *packetIn* messages;
- $E_{packetIn}$ is the energy needed to manage a *packetIn* message;
- $r_{FlowMod}$ is the rate of incoming *FlowMod* messages;
- $E_{FlowMod}$ is the energy needed to manage a *FlowMod* message.

Finally, P_{OF} is :

$$P_{OF} = \sum_{i}^{Nflows} r_{packets}(i)[\sum_{j}^{Nmatches} \mu_{match}(i \cdot j) \cdot e_{match}(j)) + \sum_{k}^{Nactions} \mu_{action}(i \cdot k) \cdot e_{action}(k)] \tag{8.4}$$

where:

- N_{flows} is the number of active flows;
- $r_{packets}(i)$ is the packet rate for the *i*th flow ;
- $e_{match}(j))$ is the energy consumed for each match ($\mu_{match}(i \cdot j) \neq 0$ only if the match happens);
- $e_{action}(j))$ is the energy consumed for each action undertaken if the match takes place ($\mu_{action}(i \cdot j) \neq 0$ only if the action is performed).

As explained in [20], this last component can be removed due to its small impact on the overall power consumption.
The power model we adopted, as a consequence, is:

$$P_{switchHW} = P_{base} + P_{config} + P_{control} \tag{8.5}$$

8.2 eA4SDN

eA4SDN is an evolution of the Adaptive Alienated Ant Algorithm for Software-Defined Networking (A4SDN) already presented in section 4.1.

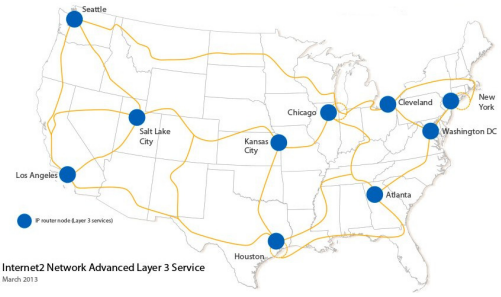


Fig. 8.1 Reference Scenario (image from [Internet 2 L3.])

In eA4SDN we focused our work on an energy aware traffic engineering algorithm for SDN networks. Through the monitoring of the network performance parameters such as throughput, delay and packet loss, we have conducted a study on the energy consumption of an SDN and on how to reduce the cost per Megabyte and balance the utilization.

8.2.1 Algorithms, testbed and settings

In order to evaluate the effectiveness of the proposed solution, we compared the energy-aware version of the A4SDN (eA4SDN) with the standard A4SDN solution and with two other different solutions based on Dijkstra's shortest path algorithm, named respectively DA (i.e., Dijkstra's algorithm) and EDA (i.e., Extended Dijkstra's algorithm, EDA).
A detailed list of the values of the parameters considered in the test environment are given in table 8.1.
The load used for evaluating the behaviour of each algorithm has been generated by using Iperf and configured to maintain network overload.
All the algorithms have been compared by taking into account 4 different aspects: Throughput, Delay, Packet loss and Energy consumption. Throughput has been measured by using the Iperf *bandwidth measurement* tool. Delay and Packet loss have been measured by using the *ping* tool. Energy consumption has been measured by using the model in Section 8.1.

8.2.2 Performance Evaluations

Fig. 8.2 shows the average cost (in terms of energy, Joule) for delivering 1 MB of data. The best result is obtained by the A4SDN solution that uses about 5.5% less energy than eA4SDN and EDA (the difference between them is less than 1%) and about 21% than the DA. This result is easy to explain: A4SDN is designed to balance and, as a conseguence, to minimize the amount of packets crossing (IN/OUT) any switch, i.e. the $P_{control}$, which represents the biggest components in Eq. 8.5 when, as it happens in the considered testing

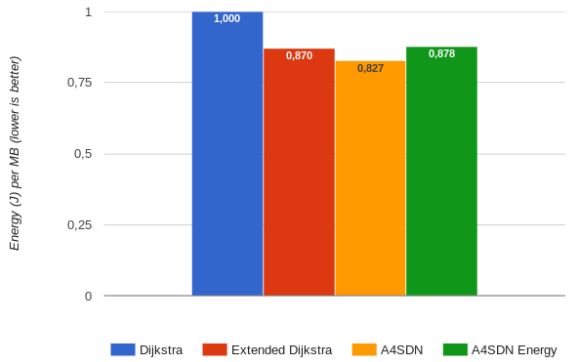


Fig. 8.2 Internet 2 Energy consumption Average per MB

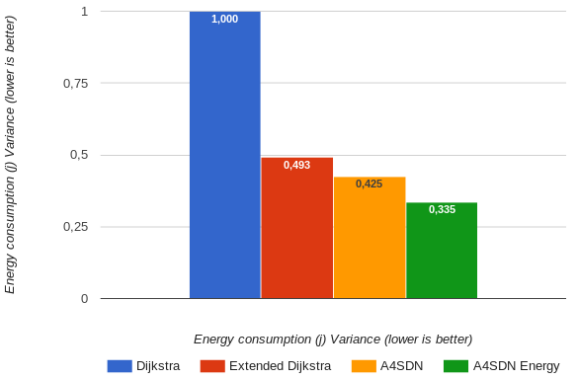


Fig. 8.3 Internet 2: Overall Energy Consumption Variance

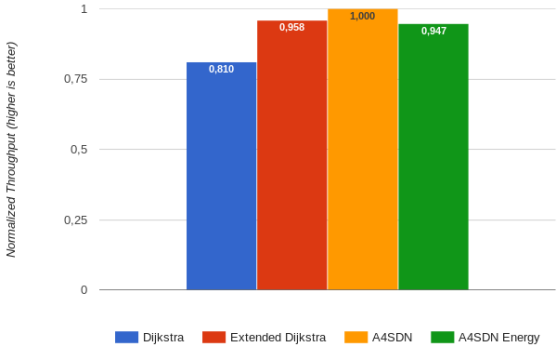


Fig. 8.4 Internet 2 Throughput

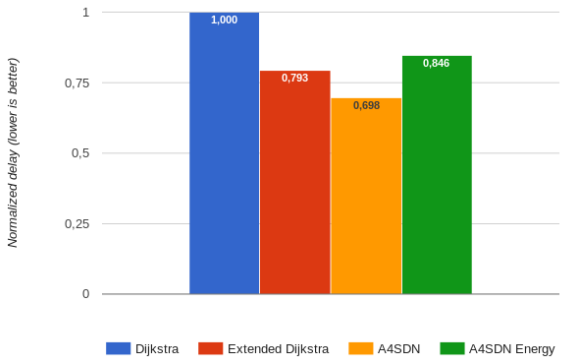


Fig. 8.5 Internet 2 Delay

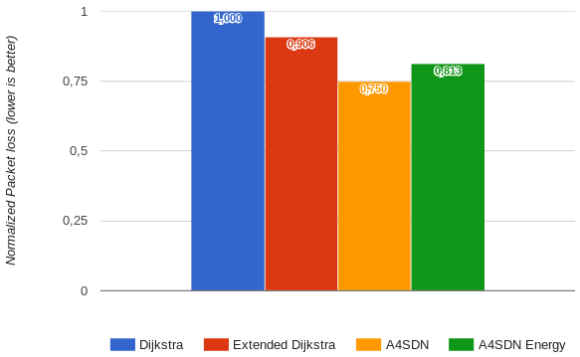


Fig. 8.6 Internet 2 Packet Loss

Table 8.1 Internet 2 testbed

Link Bandwidth	10 Mbps
Link Propagation Delay	10 ms
Number of servers	2
Number of switches	10
Number of edges	15
SDN Controller	OpenDayLight 0.2.3 Helium-SR3
SDN enabled network	Mininet 2.2.1
OpenFlow version	1.3.0
jFlowLight version	0.9.1
Iperf version	3.0.7
Ping	-
Testing time	300 s

scenario, the network is active and loaded. eA4SDN, instead, is designed to balance the energy consumption on each switch, i.e. the entire Eq. 8.5. Fig. 8.3 shows the variance of energy consumption across all the switches of the considered network: as it is easy to note, eA4SDN is able to arrange the routing in order to keep the level of energy consumption well balanced across all of the network. In a homogenous environment (as the one taken into account) characterised by a P_{base} (i.e. static consumption) equal in each switch, the difference in energy consumption is only due to the dynamic aspects of the routing: the higher the consumption, the higher the rules and packets managed, the more important the role of each switch in the routing schema. As a consequence, a high value of variance is directly related with both a lower number of switches involved in the routing activities and an asymmetric distribution of the workload: this explains the high value of EDA (47.2% higher than eA4SDN) and DA (about 300% higher than eA4SDN) that, being based on Dijkstra's algorithm, tend to use a well defined path (or paths) for their routing solutions.
From the energy consumption perspective, however, the lower the number of involved switches, the higher the amount of "passive" energy (i.e. the energy needed to keep those portions of the network that are under-utilised, or not used at all) wasted. From this viewpoint, the eA4SDN has the best performance in terms of energy consumption.
The A4SDN, instead, is the best solution for all of the other considered aspects. As expected, the dynamic and adaptive load balancing capability of A4SDN makes it the best solution in terms of throughput. In particular, as shown in Fig. 8.4, the throughput measured for A4SDN is 23.4% better the the one measured for DA, 4.36% better than EDA and 5.59% better than its Energy-aware version. eA4SDN and EDA have the same performance (EDA is about 1% better).
Fig. 8.5 summarises the results on the performance of each algorithm in terms of the average delay per transmitted packet. Also in this case, A4SDN performs better than the other solutions. Its ability to prevent congestion, along with its load balancing capability, makes A4SDN able to deliver packets, on average, 43.2% faster than DA, 13.6% than EDA and 21.2% than eA4SDN. The poor performance of DA is easily explained by the fact that it uses static paths and that it is not able to react to traffic congestions: when a path tagged as "shortest" becomes congested, the DA is not able to modify it, continuing to forward packets onto it and thus feed the congestions. On the contrary, EDA is able to react to congestions by searching for a new shortest path: this allows it to perform 26% better than its static version. The ability to avoid congestions also strongly influences the probability of packet loss during the routing activities. As shown in Fig. 8.6, both AAA-based solutions (A4SDN and eA4SDN) perform better that the solutions based on Dijkstra's algorithm. Although the standard version of the A4SDN has again the best performance (8.4% better than eA4SDN,

20.8% better than EDA and 33.3% than DA) also the eA4SDN also performs much better than EDA (11.2%) and DA(23%), highlighting the importance of avoiding congestions, which represent the only cause of packet loss in this scenario.

The results have demonstrated that eA4SDN is able to minimise the amount of "passive" energy, i.e. the energy needed to keep portions of network that is under-utilised or not used at all.

Chapter 9

Conclusions and Future Works

Software Defined Networking and, in general, all the virtual network technologies are completely transforming the way networks are managed.
By combining the ability to implement network services on top of general-purpose hardware resources and the ability to coordinate all involved resources with a centralized approach, these technologies provide simple and flexible way to build, configure and easily manage complex network services.

However, the current implementation of SDN solutions takes into account only single cloud environments, where computing, storage and network resources are locally-distributed and owned by a single organization.
This work aims to extend the concept of an SDN to build robust, reliable and performant networks across geographically-distributed resources owned by different providers, that can be competitive or federated.

The first step in this direction has been the definition of the Adaptive Strategy for Software Defined Networking (ASSDN), a framework that allows for the creation, monitoring and control of heterogeneous SDN networks.
ASSDN represents a fundamental tools for the integration of networks belonging to different providers and, as a consequence, for the development of QoS-controlled networks across multicloud environments.
ASSDN offers an API for the configurazion of the routing and the performance-related network parameters independently from both the specific SDN technology adopted and from the peculiarities and heterogeneities of the network providers.
It allows the planning activities to be written in a high-level language (Java in this discussion) and to develop, maintain and easily change the routing algorithms. This allows any cloud

provider managing its own control plane functions not only to configure its own network but also to offer capabilities for making them accessible as a configuration service to the owner of the applications or even to other cloud providers, thus extending the concept of large-scale Network-as-a-Service.
The main contribution of this work, however, is the definition and the development of the Adaptive Alienated Ant Algorithm for SDN (A4SDN), a distributed, adaptive, load-balancing algorithm for traffic engineering on an SDN. A4SDN is based on the Alienated Ant Algorithm (AAA), a bio-inspired, stochastic-based, heuristic approach based on a non-natural behaviour of an ant.
AAA forces the ants in search of food to distribute themselves over all the available paths rather than to converge to a single one.

A4SDN uses this non natural behaviour to build an effective routing algorithm. Considering the packets as ants and the load on network switches as pheromone, it is possible to optimise the network performance in terms of throughput, communication delay and packet loss rate by redirecting the traffic flows to the appropriate path, i.e. the least loaded one based on the current network state.
By means of this strategy, A4SDN is able to:

- reduce the network congestion;
- lead to high throughput and low delay;
- ensure a uniform distribution of the load across all the switches.

The results of comparisons have shown that A4SDN outperforms, in terms of load balance, network end-to-end latency, throughput and packet loss, the other Dijkstra's algorithms taken into account.
The last contribution of this work is the definition of an energy-aware extension of the A4SDN.
Unlike A4SDN that balances the packets across the links, eA4SDN is designed to balance the energy consumption on each switch: the pheromone, in this case, is represented by the energy consumed in the routing activities and, as a consequence, the path selection is executed by minimising the energy used by each switch at every step.
The performance of A4SDN has been compared with two deterministic solution based on Dijkstra's Algorithm: the results have demonstrated that the proposed approach is able to maximize the throughput and minimize the packet loss, while eA4SDN is able to minimise the amount of "passive" energy, i.e. the energy needed to keep active portions of network underutilised or not used at all.

This work has been mainly focused on the design and the development of the ASSDN framework and on the evaluation of the several versions of the A4SDN.
The topics covered, even if deeply analysed, can be extended to evaluate additional functionality and different optimization strategy, or compared with other heuristic-based approach in different scenarios.
The following interesting ideas could be explored:

- Evaluating the ASSDN approach to a real hybrid, multi-provider cloud scenario;
- Evaluating the capability of a NaaS controller on the network segments belonging to several, independent SDN islands;
- Extending the sensitivity analysis of the A4SDN for understanding the best match between its configuration parameters and the underlying scenario.

References

[1] Bandieramonte, M., Di Stefano, A., and Morana, G. (2010). Grid jobs scheduling: the alienated ant algorithm solution. *Multiagent and Grid Systems*, 6(3):225–243.

[2] Bandieramonte, M. and Stefano, A. D. (2010). Evaluating the robustness of the alienated ant algorithm in grids. pages 272–274.

[3] Beacon (2015). Beacon website. https://openflow.stanford.edu/display/Beacon/Home. Accessed: 2017-09-21.

[4] Chiang, C.-W., Lee, Y.-C., Lee, C.-N., and Chou, T.-Y. (2006). Ant colony optimisation for task matching and scheduling. In *Computers and Digital Techniques, IEE Proceedings-*, volume 153, pages 373–380. IET.

[5] Chiosi, M., Clarke, D., Willis, P., Reid, A., Feger, J., Bugenhagen, M., Khan, W., Fargano, M., Cui, C., Deng, H., et al. (2012). Network functions virtualisation introductory white paper. In *SDN and OpenFlow World Congress*.

[6] Di Stefano, A., Cammarata, G., Morana, G., and Zito, D. (2015). A4sdn - adaptive alienated ant algorithm for software-defined networking. pages 344–350. IEEE.

[7] Di Stefano, A. and Morana, G. (2012). A bio-inspired distributed algorithm to improve scheduling performance of multi-broker grids. *Natural Computing*, 11(4):687–700.

[8] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

[9] Dorigo, M. and Blum, C. (2005). Ant colony optimization theory: a survey. *Journal of Theoretical Computer Science*, 344(2):243–278.

[10] Dorigo, M., Caro, G. D., and Gambardella, L. M. (1999). Ant algorithms for discrete optimization. *Artificial life*, 5(2):137–172.

[11] Farooq, M. and Di Caro, G. A. (2008). Routing protocols for next-generation networks inspired by collective behaviors of insect societies: An overview. In *Swarm Intelligence*, pages 101–160. Springer.

[12] Fidanova, S. and Durchova, M. (2006). Ant algorithm for grid scheduling problem. In *Large-Scale Scientific Computing*, pages 405–412. Springer.

[13] FloodLight (2017). Floodlight website. http://www.projectfloodlight.org/floodlight/. Accessed: 2017-09-21.

[14] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615.

[15] Fundation, O. N. (2012). Software-defined networking: The new norm for networks. *ONF White Paper*.

[16] Gambardella, L. M. and Dorigo, M. (2000). An ant colony system hybridized with a new local search for the sequential ordering problem. *INFORMS J. on Computing*, 12(3):237–255.

[Internet 2 L3.] Internet 2 L3. Internet 2 layer 3. http://www.internet2.edu/products-services/advanced-networking/layer-3-services/. Accessed: 2017-09-21.

[18] Iperf (2015). Iperf website. https://github.com/esnet/iperf. Accessed: 2017-09-21.

[jflowlight] jflowlight. jflowlight website. https://github.com/giovannicammarata/jflowlight. Accessed: 2017-09-21.

[20] Kaup, F., Melnikowitsch, S., and Hausheer, D. (2014). Measuring and modeling the power consumption of openflow switches. In *Network and Service Management (CNSM), 2014 10th International Conference on*, pages 181–186. IEEE.

[21] Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA. ACM.

[22] Mell, P. and Grance, T. (2011). The nist definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD.

[23] Moore, E. F. (1959). *The shortest path through a maze*. Bell Telephone System.

[24] NYTimes (2015). The new york times. http://www.nytimes.com/2012/09/23/technology/data-centers-waste-vast-amounts-of-energy-belying-industry-image.html. Accessed: 2017-03-04.

[25] Onos (2015). Onos website. http://onosproject.org/. Accessed: 2017-09-21.

[26] OpenContrail (2017). Opencontrail website. http://http://www.opencontrail.org//. Accessed: 2017-09-21.

[27] OpenDayLight (2015). Opendaylight website. "http://www.opendaylight.org/. Accessed: 2017-09-21.

[28] Pox,Nox (2015). Pox/nox website. http://www.noxrepo.org/. Accessed: 2017-09-21.

[29] Reimann, M., Doerner, K., and Hartl, R. F. (2004). D-ants: Savings based ants divide and conquer the vehicle routing problem. *Computers & Operations Research*, 31(4):563–591.

[30] Ryu (2017). Ryu website. http://osrg.github.io/ryu/. Accessed: 2017-09-21.

[31] Schoonderwoerd, R., Holland, O., and Bruten, J. (1997). Ant-like agents for load balancing in telecommunications networks. In *Proceedings of the first international conference on Autonomous agents*, pages 209–216. ACM.

[sdn] sdn. Sdn website. https://www.opennetworking.org/sdn-resources/sdn-definition. Accessed: 2017-09-21.

[33] Sim, K. M. and Sun, W. H. (2003a). Ant colony optimization for routing and load-balancing: survey and new directions. *IEEE Trans. on Systems, Man and Cybernetics*, 33(5):560–572.

[34] Sim, K. M. and Sun, W. H. (2003b). Multiple ant colony optimization for load balancing. In *Intelligent Data Engineering and Automated Learning*, pages 467–471. Springer.

[Specification] Specification, O. S. Version 1.4.0, october 14, 2013.

Appendix A

How to install Mininet

The Mininet VM is meant to speed up Mininet installation, plus make it easy to run on non-Linux platforms. The VM works on Windows, Mac, and Linux, through VMware, VirtualBox, QEMU and KVM.
After downloading the VM, you'll run a few steps to customize it for your setup. This won't take long.

A.0.1 VM Setup

Download the Mininet VM from https://github.com/mininet/mininet/wiki/Mininet-VM-Images.
The VM comes out to 1GB compressed and 2GB uncompressed. It is an OVF (Open Virtualization Format) virtual machine image which can be imported by most virtual machine monitors.
Download and install a virtualization program such as: VMware Workstation for Windows or Linux, VMware Fusion for Mac, VirtualBox (free!, GPL) for any platform, or qemu (free!, GPL) for Linux. If you already have VMware, we find that it runs Mininet somewhat faster than VirtualBox. However, VirtualBox is free to download and distribute, which is a definite advantage!

A.0.2 Boot VM

Add the VM and start it up, in the virtualization program of your choice:

Virtualbox

Usually you can just double-click on the .ovf file and import it. If you get errors importing the .ovf file, you can simply create a new VM of the appropriate type (e.g. Linux, Ubuntu 64-bit) and use the .vmdk file as the virtual hard disk for the new VM.
Select "settings," and add an additional host-only network adapter that you can use log in to the VM image. Start the VM.
For more information on setting up networking in VirtualBox, you may wish to check out these VirtualBox specific instructions

VMware

Import the OVF file, then start the VM.
VMware may ask you to install VMware tools on the VM - if it asks, decline. Everything graphical in the tutorial is done via X forwarding through SSH (in fact, the VM doesn't have a desktop manager installed), so the VMware tools are unnecessary unless you wish to install an X11/Gnome/etc. environment in your VM.

Qemu/KVM

For Qemu, something like the following should work:

```
qemu-system-i386 -m 2048 mininet-vm-disk1.vmdk -net nic,model=virtio -net user,net=19
```

For KVM:

```
sudo qemu-system-i386 -machine accel=kvm -m 2048 mininet-vm-disk1.vmdk -net nic,model
```

The above commands will set up ssh forwarding from the VM to host port 8022.
Parallels: Use Parallels Transporter to convert the .vmdk file to an .hdd image that Parallels can use, and then create a new VM using that .hdd image as its virtual drive. Start the VM.

A.0.3 Log in to VM

Log in to the VM, using the following name and password:

```
mininet-vm login: mininet
Password: mininet
```

(some older VM images may use openflow/openflow instead) The root account is not enabled for login; you can use sudo to run a command with superuser privileges.

A.0.4 SSH into VM

First, find the VM's IP address, which for VMware is probably in the range 192.168.x.y. In the VM console:

```
ifconfig eth0
```

Note: VirtualBox users who have set up a host-only network on eth1 should use

```
sudo dhclient eth1   \# make sure that eth1 has an IP address
ifconfig eth1
```

You may want to add the address to your host PC's /etc/hosts file to be able to SSH in by name, if it's Unix-like. For example, add a line like this for OS X:
192.168.x.y mininet-vm
where 192.168.x.y is replaced by the VM's IP address.
SSH into the VM. We assume the VM is running locally, and that the additional precautions of ssh -X are unnecessary. ssh -Y also has no authentication timeout by default.

```
ssh -Y mininet@mininet-vm
```

If you're running the VM under QEMU/KVM with -net user and the hostfwd option as recommended above, the VM IP address is irrelevant. Instead you tell SSH to connect to port 8022 on the host:

```
ssh -Y -p 8022 mininet@localhost
```

Appendix B

Installing the OpenDaylight

You complete the following steps to install your networking environment, with specific instructions provided in the subsections below.

Before detailing the instructions for these, we address the following: Java Runtime Environment (JRE) and operating system information Target environment Known issues and limitations

B.0.1 Downloading and installing OpenDaylight

The default distribution can be found on the "http://www.opendaylight.org/software/downloads" page.

The Karaf distribution has no features enabled by default. However, all of the features are available to be installed.

For compatibility reasons, you cannot enable all the features simultaneously. We try to document known incompatibilities in the Install the Karaf features section below.

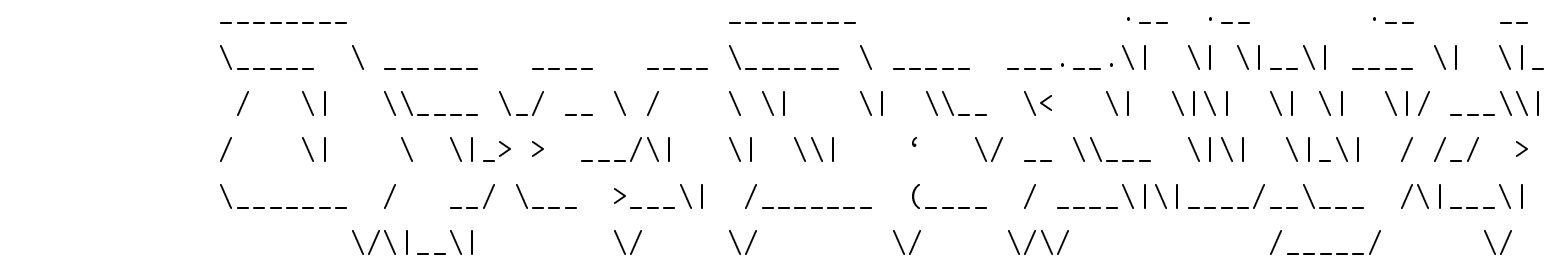
B.0.2 Running the karaf distribution

To run the Karaf distribution:

```
Unzip the zip file.
Navigate to the directory.
run ./bin/karaf.
```

For Example:

```
$ ls distribution-karaf-0.6.x-Carbon.zip
distribution-karaf-0.6.x-Carbon.zip
$ unzip distribution-karaf-0.6.x-Carbon.zip
Archive:  distribution-karaf-0.6.x-Carbon.zip
   creating: distribution-karaf-0.6.x-Carbon/
   creating: distribution-karaf-0.6.x-Carbon/configuration/
   creating: distribution-karaf-0.6.x-Carbon/data/
   creating: distribution-karaf-0.6.x-Carbon/data/tmp/
   creating: distribution-karaf-0.6.x-Carbon/deploy/
   creating: distribution-karaf-0.6.x-Carbon/etc/
   creating: distribution-karaf-0.6.x-Carbon/externalapps/
   ...
   inflating: distribution-karaf-0.6.x-Carbon/bin/start.bat
   inflating: distribution-karaf-0.6.x-Carbon/bin/status.bat
   inflating: distribution-karaf-0.6.x-Carbon/bin/stop.bat
$ cd distribution-karaf-0.6.x-Carbon
$ ./bin/karaf
```



Press tab for a list of available commands Typing [cmd] –help will show help for a specific command. Press ctrl-d or type system:shutdown or logout to shutdown OpenDaylight.

Please take a look at the Deployment Recommendations and following sections under Security Considerations if you're planning on running OpenDaylight outside of an isolated test lab environment.

B.0.3 Install the Karaf features

To install a feature, use the following command, where feature1 is the feature name listed in the table below:

```
feature:install <feature1>
```

You can install multiple features using the following command:

```
feature:install <feature1> <feature2> ... <featureN-name>
```

Note

For compatibility reasons, you cannot enable all Karaf features simultaneously. The table below documents feature installation names and known incompatibilities.Compatibility values indicate the following:

all - the feature can be run with other features. self+all - the feature can be installed with other features with a value of all, but may interact badly with other features that have a value of self+all. Not every combination has been tested.

Uninstalling features

To uninstall a feature, you must shut down OpenDaylight, delete the data directory, and start OpenDaylight up again.

Important

Uninstalling a feature using the Karaf feature:uninstall command is not supported and can cause unexpected and undesirable behavior.

B.0.4 Listing available features

To find the complete list of Karaf features, run the following command:

```
feature:list
```

To list the installed Karaf features, run the following command:

```
feature:list -i
```

Features to implement networking functionality provide release notes, which you can find in the Project-specific Release Notes section.