**Research conducted on: Implementation Challenges**

**For**

**Software-Deﬁned Networks, Openflow and Architecture**

*by: Team Dasein*

**ABSTRACT**

Cloud services are exploding and organizations are converging their data centers in order to take advantage of the predictability, continuity, and quality of service delivered by virtualization technologies. In parallel, energy-eﬃcient and high-security networking is of increasing importance. Network operators, service and product providers require a new network solution to eﬃciently tackle the increasing demands of this changing network landscape. Software-Deﬁned Networking has emerged as an eﬃcient network technology capable of supporting the dynamic nature of future network functions and intelligent applications while lowering operating costs through simpliﬁed hardware, software, and management. In this article, the question of how to achieve a successful carrier grade network with Software-Deﬁned Networking is raised. Speciﬁc focus is placed on the challenges of network performance, scalability, security and interoperability with the proposal of potential solution directions.

**INTRODUCTION**

**What is Software-Deﬁned Networking?**

Network conﬁguration and installation requires highly-skilled personnel adept at conﬁguration of many network elements. Where interactions between network nodes (e.g. switches, routers, etc.) are complex, a more systems-based approach encompassing elements of simulation is required. With the current programming interfaces on much of today’s networking equipment, this is diﬃcult to achieve.

In addition, operational costs involved in provisioning and managing large, multi-vendor networks covering multiple technologies have been increasing over recent years, whilst the pre- dominant trend in revenue for operations has been decreasing. Coupled with increasing scarcity of human resources and increasing costs of real-estate, this “perfect storm” for service providers is leading to renewed interest in solutions that can unify network management and provisioning across multiple domains. A new network model is required to support this.

The term Software-Deﬁned Networking (SDN) has been coined in recent years. However, the concept behind SDN has been evolving since 1996 driven by the desire to provide user controlled management of forwarding in network nodes. Implementations by research and industry groups include Epsilon (proposed General Switch Management protocol, 1996), the tempest (a framework for safe, resource-assured, programmable networks, 1998) and IETF Forwarding and Control Element Separation, 2000, and Path Computation Element, 2004. Most recently, Ethane (2007) and Open Flow (2008) have brought the implementation of SDN closer to reality. Ethane is a security management architecture combining simple ﬂow-based switches with a central controller managing admittance and routing of ﬂows. Open Flow enables entries in the Flow Table to be deﬁned by a server external to the switch. SDN is not, however, limited to any one of these implementations, but is a general term for the platform.

For clarity, SDN is described in this article with the Open Networking Foundation (ONF) [1] deﬁnition: “In the SDN architecture, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications.”

**SDN focuses on four key features:**

• Separation of the control plane from the data plane,

• A centralized controller and view of the network,

• Open interfaces between the devices in the control plane and the data plane, and

• Programmability of the network by external applications.

The vision of the future SDN architecture is described in Figure 1.

This architecture encompasses the complete network platform.

The bottom tier of Figure 1 involves the physical network equipment including Ethernet switches and routers. This forms the data plane.

The central tier consists of the controllers that facilitate setting up and tearing down ﬂows and paths in the network. The controllers use information about capacity and demand obtained from the networking equipment through which the traﬃc ﬂows. The central tier links with the bottom tier via an Application Programming Interface (API) referred to as the southbound API. Connections between controllers operate with east and westbound APIs. The controller- application interface is referred to as the northbound API.

Functional applications such as energy-eﬃcient networking, security monitoring and access control for operation and management of the network are represented at the top of Figure 1.

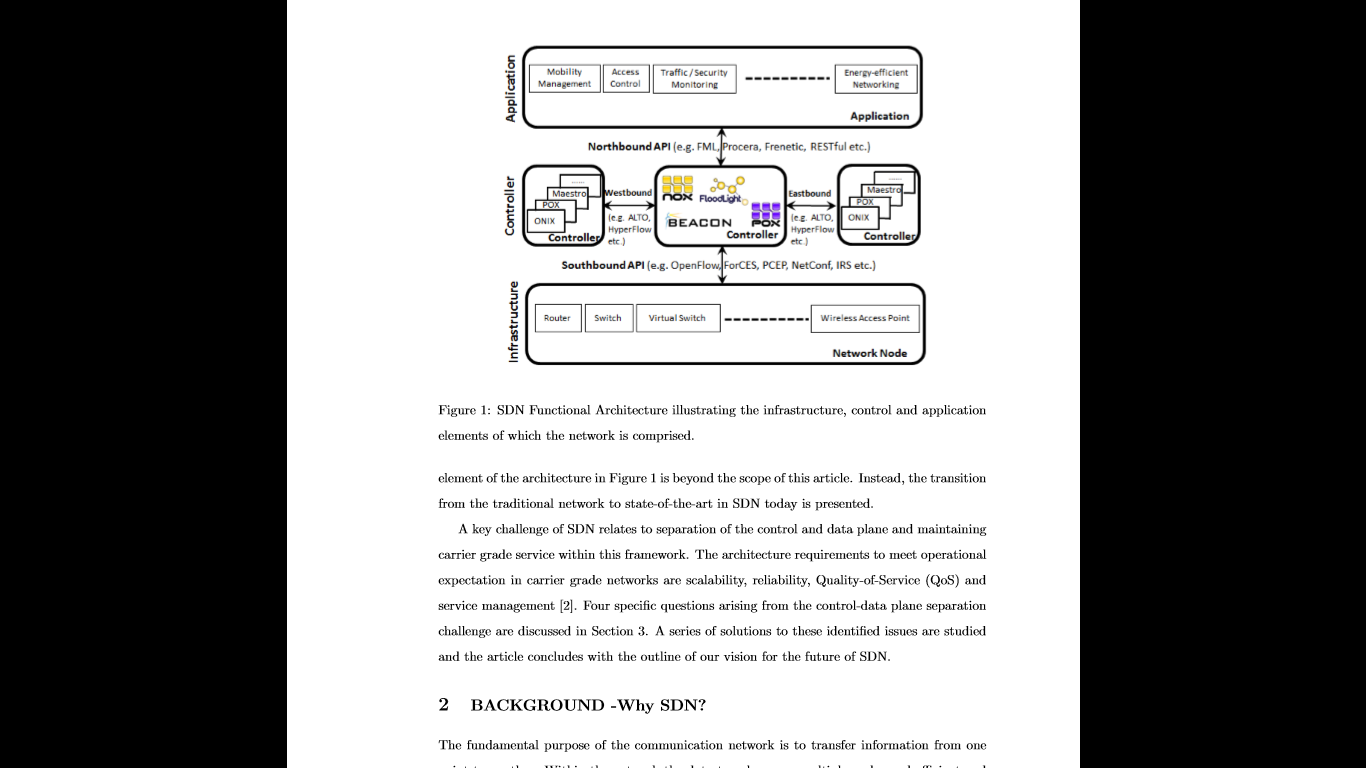
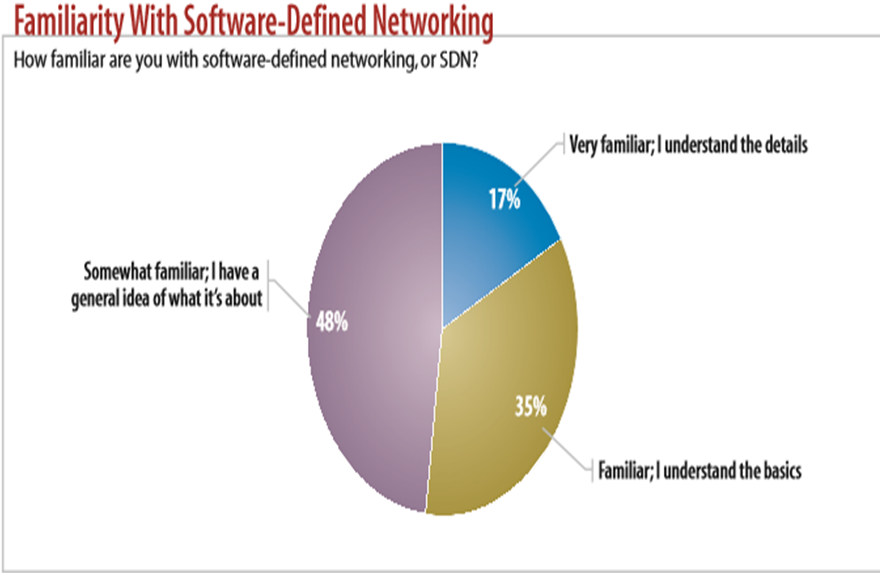


Figure 1: SDN Functional Architecture illustrating the infrastructure, control and application elements of which the network is comprised.

The transition from the traditional network to state-of-the-art in SDN today is presented.

A key challenge of SDN relates to separation of the control and data plane and maintaining carrier grade service within this framework. The architecture requirements to meet operational expectation in carrier grade networks are scalability, reliability, Quality-of-Service (QoS) and service management. Four speciﬁc questions arising from the control-data plane separation challenge are discussed in Section 3. A series of solutions to these identiﬁed issues are studied and the article concludes with the outline of our vision for the future of SDN.

**How familiar are people and organizations with SDN?**



**BACKGROUND -Why SDN?**

The fundamental purpose of the communication network is to transfer information from one point to another. Within the network the data travels across multiple nodes and eﬃcient and eﬀective data transfer (forwarding) is supported by the control provided by network applications/services.

**Networking - The Old Way:**

In traditional networks, as shown in Figure 2, the control and data planes are combined in a network node.

The control plane is responsible for conﬁguration of the node and programming the paths that will be used for data ﬂows. Once these paths have been determined they are pushed down to the data plane. Data forwarding at the hardware level is based on this control information. In this traditional approach, once the ﬂow management (forwarding policy) has been deﬁned, the only way to make an adjustment to the policy is via changes to the conﬁguration of the devices. This has proven restrictive for network operators who are keen to scale their networks in response to changing traﬃc demands, increasing use of mobile devices and the impact of “Big Data”.

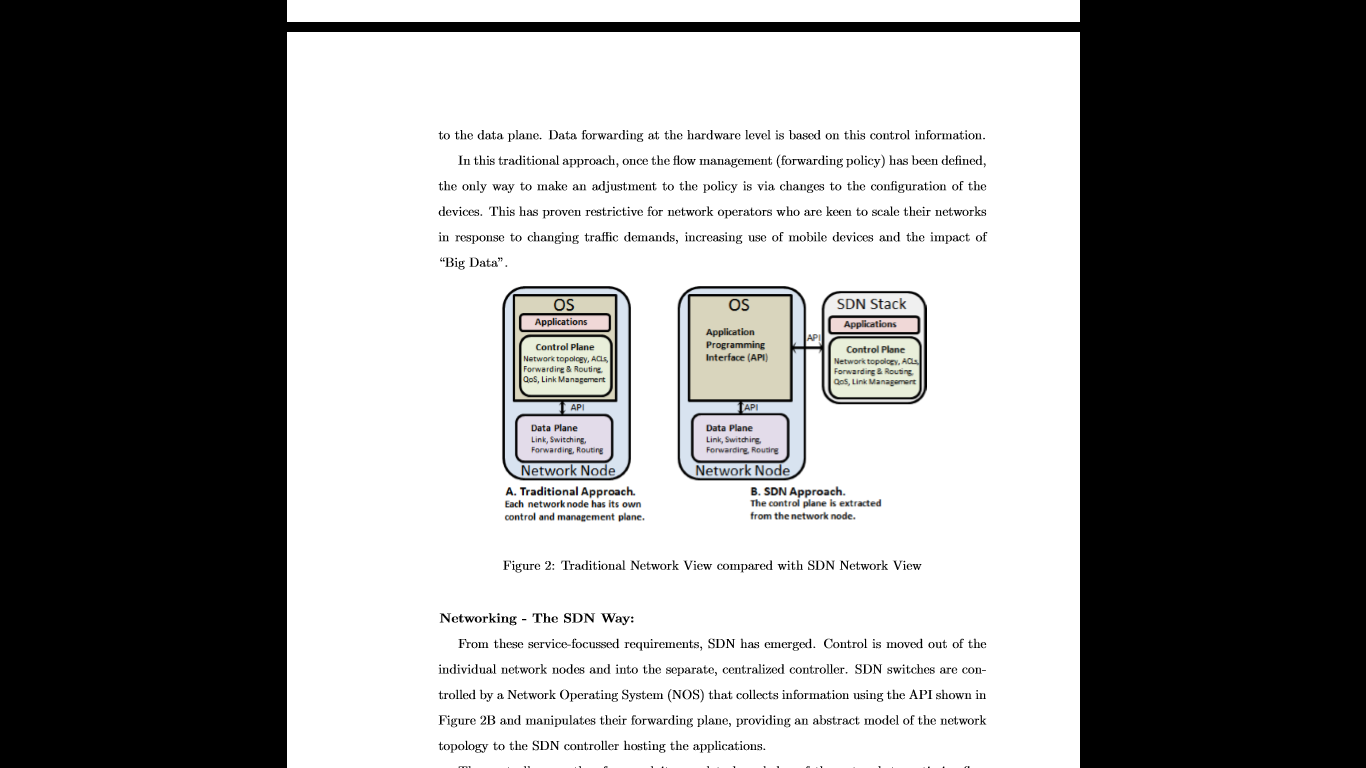
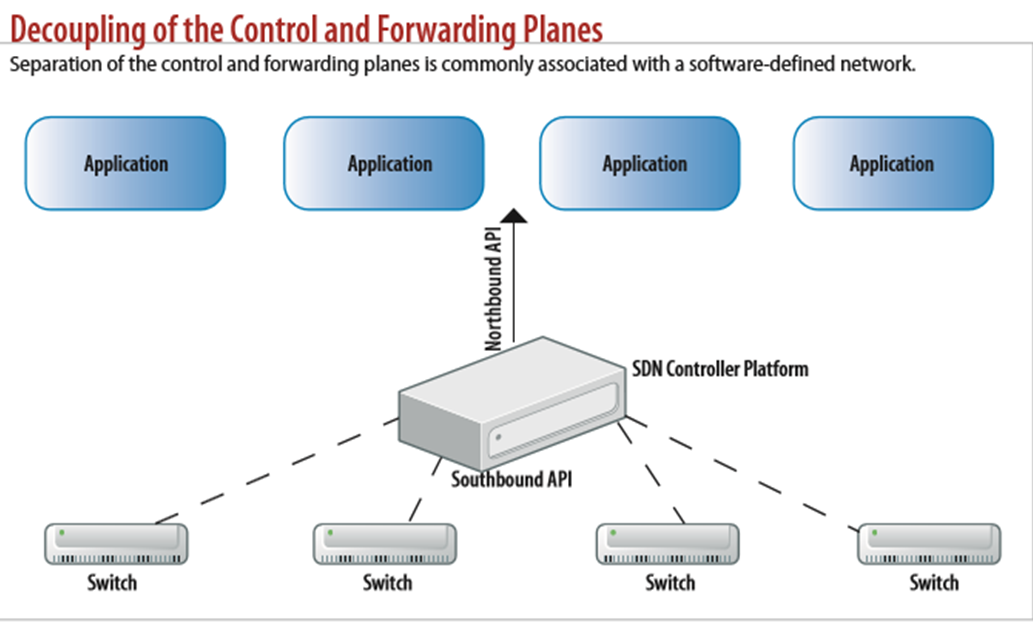


Figure 2: Traditional Network View compared with SDN Network View

**Networking - The SDN Way:**

****

From these service-focused requirements, SDN has emerged. Control is moved out of the individual network nodes and into the separate, centralized controller. SDN switches are controlled by a Network Operating System (NOS) that collects information using the API shown in

Figure 2B and manipulates their forwarding plane, providing an abstract model of the network topology to the SDN controller hosting the applications.

The controller can therefore exploit complete knowledge of the network to optimize ﬂow management and support service-user requirements of scalability and ﬂexibility. For example, bandwidth can be dynamically allocated into the data plane from the application.

In Figure 3, once the ﬁrst packet of a new ﬂow arrives at the switch from the sender (Step

1) The switch checks for a ﬂow rule for this packet in the SDN cache (Step 2). If a matching entry is found, the instructions associated with the speciﬁc ﬂow entry are executed e.g. update counter, packet/match ﬁelds, action set, and metadata. Packets are then forwarded to the receiver (Step 5).

If no match is found in the ﬂow table, (Step 3) the packet may be forwarded to the controller over a secure channel. Using the southbound API (e.g. Open Flow, ForCES, PCEP etc.), the controller can add, update, and delete ﬂow entries, both reactively (in response to packets) and proactively. The controller executes the routing algorithm and (Step 4) adds a new forwarding

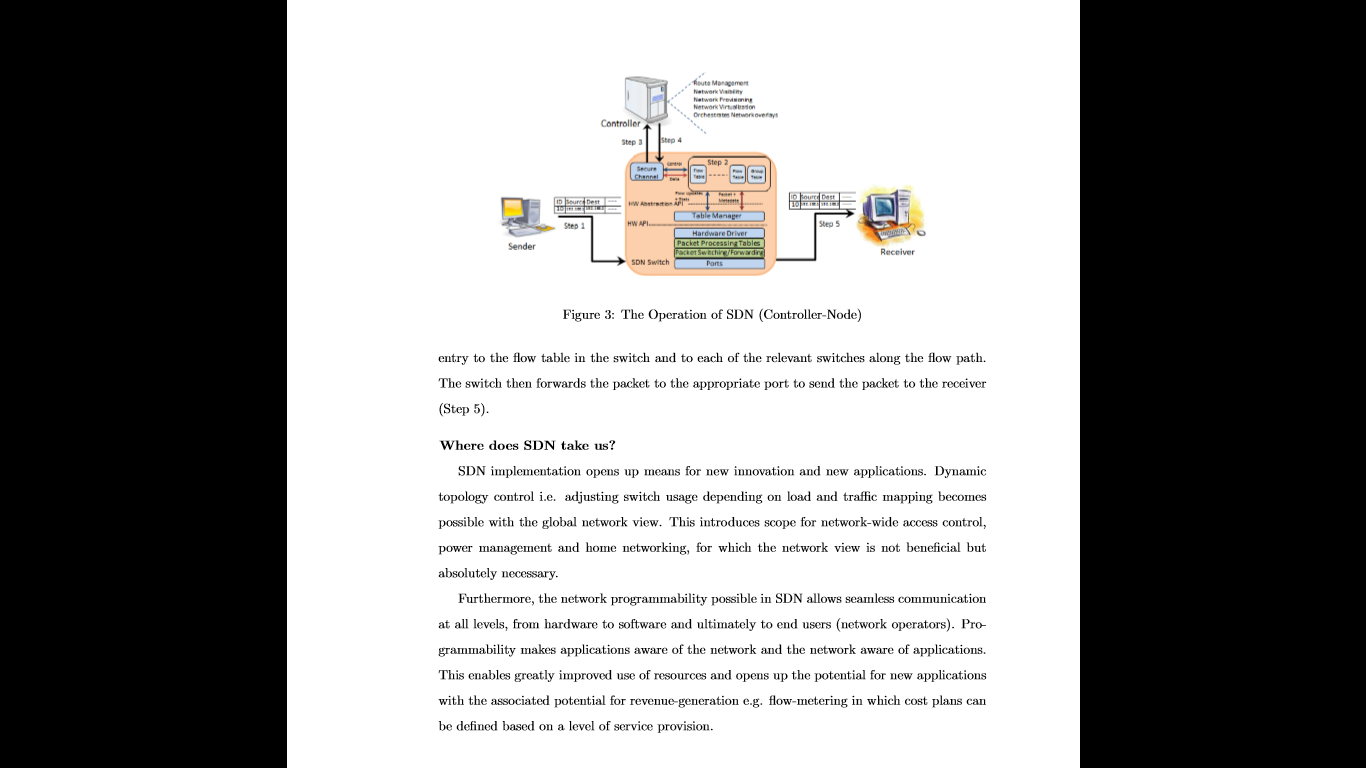


Figure 3: The Operation of SDN (Controller-Node) entry to the ﬂow table in the switch and to each of the relevant switches along the ﬂow path.

The switch then forwards the packet to the appropriate port to send the packet to the receiver (Step 5).

**Where does SDN take us?**

SDN implementation opens up means for new innovation and new applications. Dynamic topology control i.e. adjusting switch usage depending on load and traﬃc mapping becomes possible with the global network view. This introduces scope for network-wide access control, power management and home networking, for which the network view is not beneﬁcial but absolutely necessary.

Furthermore, the network programmability possible in SDN allows seamless communication at all levels, from hardware to software and ultimately to end users (network operators). Programmability makes applications aware of the network and the network aware of applications.

This enables greatly improved use of resources and opens up the potential for new applications with the associated potential for revenue-generation e.g. ﬂow-metering in which cost plans can be deﬁned based on a level of service provision.

**3 KEY CHALLENGES**

SDN holds great promise in terms of simplifying network deployment and operation along with lowering the total cost of managing enterprise and carrier networks by providing programmable network services. However, a number of challenges remain to be addressed. This section focuses on four speciﬁc questions arising from the challenges of SDN.

Performance vs. Flexibility: How can the programmable switch be achieved?

One fundamental challenge of SDN is how to handle high touch, high security, and high performance packet processing ﬂows in an eﬃcient manner. There are two elements to consider; performance and programmability/ﬂexibility.

In this section, performance refers speciﬁcally to the processing speed of the network node considering both throughput and latency. Programmability means the capability to change and/or accept a new set of instructions in order to alter functional behavior. Flexibility is the ability to adapt systems to support new unforeseen features (e.g. applications, protocols, security measures).

There are a number of initiatives [3, 4] underway to allow programmability of existing network technologies in a manner conformant with the goals of SDN. Beyond these, the SDN programmability and performance problem remains a challenge to achieve node bandwidth beyond 100Gbps.

Figure 4 outlines the main technologies used for network processing in terms of their relationship (trade-oﬀ) between programmability/ﬂexibility and performance.

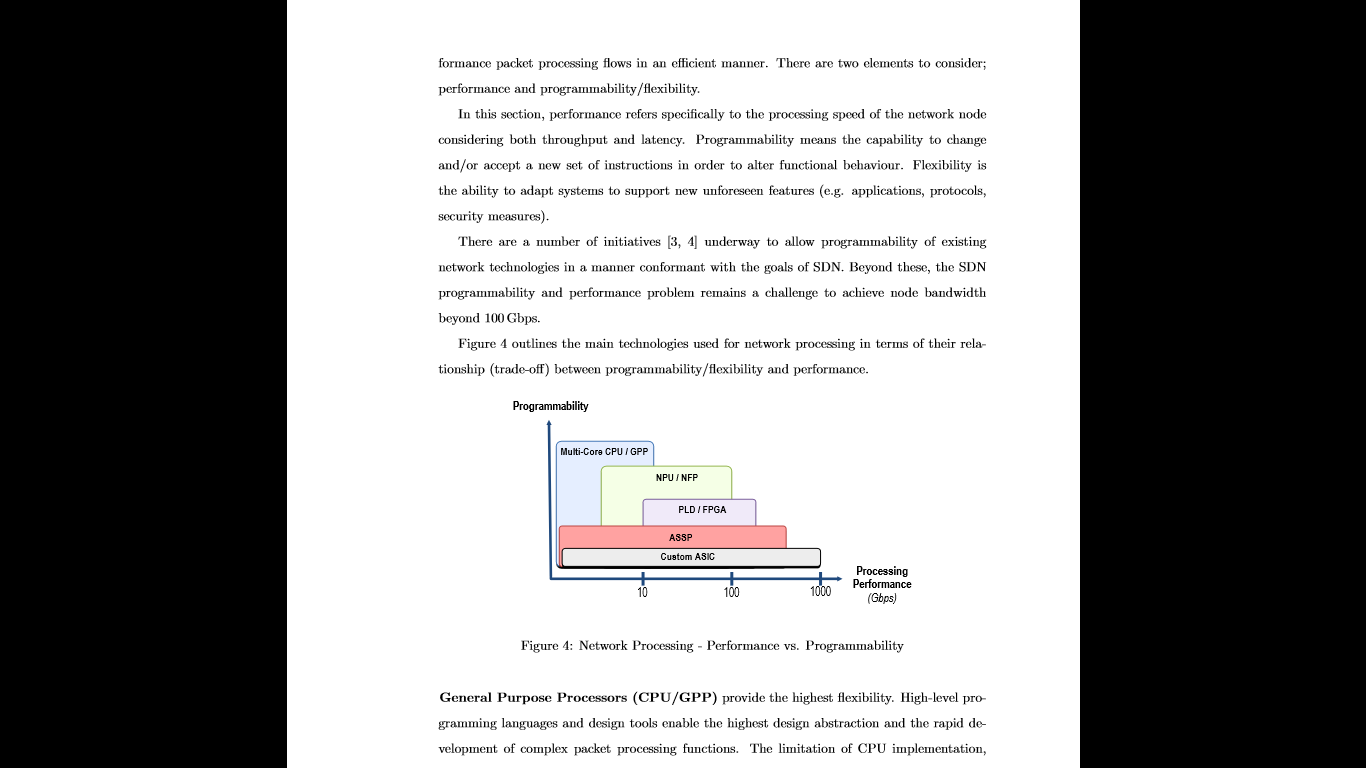


Figure 4: Network Processing - Performance vs. Programmability.

General Purpose Processors (CPU/GPP) provide the highest ﬂexibility. High-level programming languages and design tools enable the highest design abstraction and the rapid development of complex packet processing functions. The limitation of CPU implementation, however, is its performance and power dissipation, constrained by the general purpose architecture. Nevertheless, multi-core processors such as those of the Intel Xeon family can achieve several tens of Gigabits of throughput by load balancing traﬃc onto multiple cores.

Network Flow Processors (NPU/NFP) are optimized processor architectures for network processing. Instructions and interconnects are tailored for processing packetized data. Dedicated hardware accelerators and various interface technologies are used for acceleration while reducing power dissipation. However, the ﬂexibility of implementation is reduced as more detailed knowledge of the device is required in order to deﬁne the packet/ﬂow processing functions and to take full advantage of the device’s parallel processing capabilities. State of the art NPUs such as the Metronome NFP6xxx oﬀer 216 micro-cores promising ﬂow processing performance of over 200Gbps line-rate per device and well over 100Mpps.

Programmable Logic Devices (PLD) or Field Programmable Gate Arrays (FPGAs) have evolved into a technology for telecommunication and network processing. In comparison to microprocessors, PLDs are conﬁgured using hardware design tools. This technology is ideal for implementing highly parallel and pipelined data paths that are tailored for individual network processing functions. PLD technologies such as the Tabula ABAX2 [7] can achieve custom data-path processing of over 200Gbps per device e.g. 200Mpps switching.

Application Speciﬁc Standard Products (ASSP) are the cornerstone of high-performance networks. They are designed and optimized for widely used functions or products aiming for high-volume. The drawback of ASSPs is their limited ﬂexibility. Core ASSP domains are physical and data-link layer products, switching and wireless products. In recent years, SDN- speciﬁc ASSPs have been introduced by Intel, Broadcom and Marvell targeting primarily high- performance Ethernet switching with virtualization and Open Flow support for over 500Gbps switching.

Application Speciﬁc Integrated Circuits (ASIC) are proprietary devices custom-built by system vendors (e.g. Cisco, Huawei, Juniper etc.) when standard products are unavailable and programmable solutions are unable to meet performance constraints. As an application-speciﬁc solution, ASICs oﬀer the lowest ﬂexibility while providing the highest performance, power and cost beneﬁts. SDN products are expected to be comprised of proprietary ASICs to implement the SDN data plane.

Taking into account the programmability/performance trade-oﬀ of data processing technologies, it is evident that only a hybrid approach will provide an eﬀective technology solution for SDN. Main SDN node functions can be decomposed into clusters of sub-functions such that feature-speciﬁc technologies (within or across nodes) are used to satisfy the best performance versus programmability trade-oﬀ in terms of power dissipation, cost and scalability.

For example, building a platform based on custom-built devices (e.g. PLDs/ASSPs) combined with NPUs/NFPs and a CPU/GPP presents a hybrid programmable architecture. Such a platform can support fast forwarding on established ﬂows in the network along with programmability and controlled processing for encapsulated traﬃc and new ﬂows.

One goal of SDN is to develop networks built on general purpose hardware. The combination of technologies as described in the hybrid architecture supports this goal. With a programmable interface built on standard hardware, a multi-vendor equipped network becomes a possibility.

Scalability: How to enable the Controller to provide a global network view?

Assuming that the performance requirements can be achieved within the hybrid programmable architecture, a further issue that has seen some discussion but limited solution is scalability in SDN.

The issue can loosely be split into controller scalability and network node scalability. The focus here is on controller scalability in which three speciﬁc challenges are identiﬁed. The ﬁrst is the latency introduced by exchanging network information between multiple nodes and a single controller. The second is how SDN controllers communicate with other controllers using the east and westbound APIs. The third challenge is the size and operation of the controller back-end database.

Considering the ﬁrst issue, a distributed or peer-to-peer controller infrastructure would share the communication burden of the controller. However, this approach does not eliminate the second challenge of controller-to-controller interactions for which an overall network view is required.

Traditional packet networks lend themselves to scalable solutions because they do not require extensive state to be held between system units. Each network node is autonomous, requiring only limited knowledge of its neighbors. Routing protocols have been designed to control traﬃc with this in mind. In order to create resilient networks, alternative paths and secondary equipment are required. It may then be necessary to hold some state between systems to ensure that should a failure occur, there is little or no interruption in service. Typical systems that require this functionality include network elements such as Load Balancers and Firewalls.

Within a pure SDN environment, a single controller or group of controllers would provide control plane services for a wider number of data-forwarding nodes, thus allowing a system-wide view of network resources.

Other approaches that match the goals of SDN with existing routing protocols involve addition of an orchestration layer exposing an API that application elements may use to request desired performance from the transport layer.

An extension to the Application Layer Traﬃc Optimization (ALTO) data model has been proposed by various organizations in which the ALTO server hosts aggregated information to which each controller has a link. The goal of ALTO is to guide applications in their selection of one of several hosts capable of providing the desired resource. A vertical architecture with bi-directional information ﬂow between each SDN controller and the ALTO server is proposed in to support the global network view. In terms of improving application performance,

ALTO with SDN would be a powerful tool.

A speciﬁc solution to controller scalability is HyperFlow. HyperFlow is a controller application that sits on the NOX controller and works with an event propagation system. The HyperFlow application selectively publishes events that change the state of the system and other controllers replay all the published events to reconstruct the state. By this means all the controllers share the same consistent network-wide view.

Indeed, this concept of providing the network view by distributing the state over multiple controllers is highlighted in which a series of solutions to controller scalability are described. Notably, the authors conclude that the ﬂexibility of SDN provides an opportunity in terms of network manageability and functional scalability.

On the way to achieving full scalability for SDN, an evolutionary approach to network programmability will be necessary. For example, with the hybrid architecture a volume of queries can be resolved in the node CPU, which would otherwise be transferred to the controller for processing. This can potentially reduce the database size at the controller and simultaneously reduce communication between the controller and its nodes.

Security: How can the Software-Deﬁned Network can be protected from malicious attack?

There has been limited industry and research community discussion to date on the security issues associated with SDN. A greater focus on security is therefore required if SDN is going to be acceptable in broader deployment. Indeed a security working group has been set up within ONF with this in mind. A number of issues are highlighted here that underscore the need for further study and development of security solutions.

Potential security vulnerabilities exist across the SDN platform. At the controller-application level, questions have been raised around authentication and authorization mechanisms to enable multiple organizations to access network resources while providing the appropriate protection of these resources. Not all applications require the same network privileges and a security model must be put in place to isolate applications and support network protection.

One potential solution is role-based authorization. FortNox is proposed to resolve the situation when a controller receives conﬂicting ﬂow rules from two diﬀerent applications. Role based authorization alone, however, does not present a solution for the complexity of SDN requiring isolation of applications or resources.

The controllers are a particularly attractive target for attack in the SDN architecture open to unauthorized access and exploitation. Furthermore, in the absence of a robust, secure controller platform, it is possible for an attacker to masquerade as a controller and carry out malicious activities. In the past, such attacks have targeted DNS servers e.g. Kaminski DNS attack.

Considerably greater damage could be done by such an attack on an SDN controller.

A security technology such as Transport Layer Security (TLS) with mutual authentication between the controllers and their switches can mitigate these threats. Current speciﬁcations of OpenFlow describe the use of TLS. However, the security feature is optional and the standard of TLS is not speciﬁed. A full security speciﬁcation for the controller-switch interface must be deﬁned to secure the connection and protect data transmitted across it.

With a single controller controlling a set of network nodes, implementation of authentication with TLS may provide the necessary security. However, with multiple controllers communicating with a single node or multiple control processes communicating with a single, centralized controller, authorization and access control becomes more complex. The potential for unauthorized access increases and could lead to manipulation of the node conﬁguration and/or traﬃc through the node for malicious intent.

One potential malicious attack is the Denial of Service (DoS) attack. Within the operation of SDN, as illustrated in Figure 3, there are two options for the handling of a new ﬂow when no ﬂow match exists in the ﬂow table. Either the complete packet or a portion of the packet header is transmitted to the controller to resolve the query. With a large volume of network traﬃc, sending the complete packet to the controller would absorb high bandwidth.

However, if only header information is transmitted to the controller, the packet itself must be stored in node memory until the ﬂow table entry is returned. In this case, it would be easy for an attacker to execute a DoS attack on the node by setting up a number of new and unknown ﬂows. As the memory element of the node can be a bottleneck due to high cost, an attacker could potentially overload the switch memory.

Furthermore, with the introduction in SDN of open interfaces and known protocols to simplify network programming by any application provider, the door is thrown open to attackers.

With full knowledge of how to control the network, with access to the controller, the operation of the network can quickly and easily be subverted to the beneﬁt of the attacker. Even at a lower level, individual network nodes, hosts or users could be targeted undermining the desired network performance. Such issues must receive due consideration in the SDN platform design.

On the plus side, the SDN architecture supports a highly reactive security monitoring, analysis and response system. From the security perspective SDN can support:

• Network Forensics: facilitate quick and straightforward, adaptive threat identiﬁcation and management through a cycle of harvesting intelligence from the network, analyzing it, updating policy and then reprogramming to optimize from network experience.

• Security Policy Alteration: allow you to deﬁne a security policy and have it pushed out to all the infrastructure elements, reducing the frequency of misconﬁguration and conﬂicting policies across the infrastructure.

• Security Service Insertion: facilitate security service insertion where applications like

ﬁrewalls and Intrusion Detection Systems (IDS) can be applied to speciﬁed traﬃc according to the organization’s policies.

However, the security of SDN will only be as good as the deﬁned security policy. Implementation of existing authentication and authorization mechanisms can resolve some aspects of the security challenge. Meanwhile, threat detection and protection techniques will continue to evolve. The key, though, is for individual organizations to eﬀectively and comprehensively deﬁne their security policies in order to exploit the full extent of available network protection.

Interoperability: How can SDN solutions be integrated into existing networks?

To answer this question requires consideration of interoperability and standardization to support the transition from the traditional network model to SDN.

It would be straightforward to deploy a completely new infrastructure based on SDN technology. For this, all elements and devices in the network would be SDN-enabled. However, there exists a vast, installed-base of networks supporting vital systems and businesses today.

To simply “swap-out” these networks for new infrastructure is not going to be possible and is only well suited for closed environments such as data centers and campus networks.

The transition to SDN therefore requires simultaneous support of SDN and legacy equipment. The IETF Path Computation Element (PCE) [14] could help in gradual or partial migration to SDN. With PCE, the path computation component of the network is moved from the networking node to a centralized role while traditional network nodes not using PCE continue to use their existing path computation function. A speciﬁc protocol (PCEP) enables communication between the network elements. However, PCE does not provide complete SDN.

The centralized SDN controller supports complete path computation for the ﬂow across multiple network nodes.

Further development is required to achieve a hybrid SDN infrastructure in which traditional,

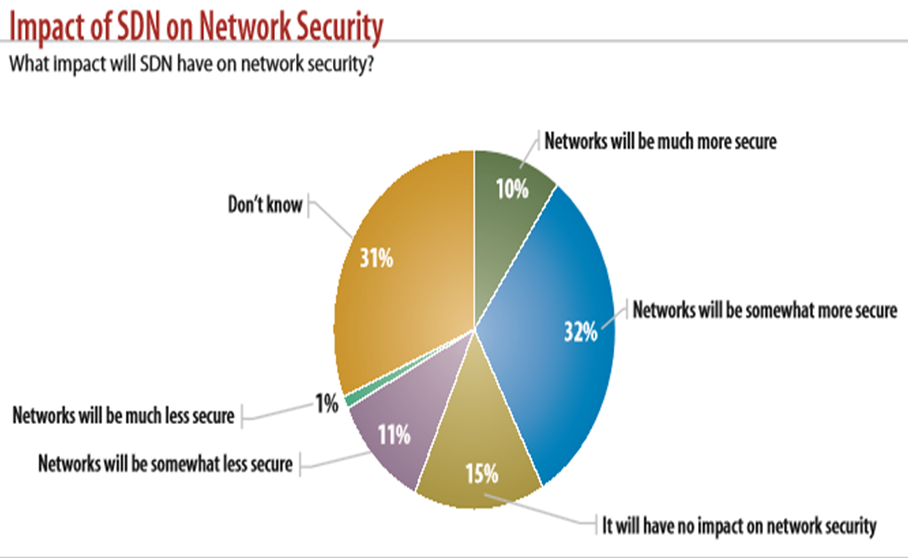
SDN-enabled and hybrid network nodes can operate in harmony. Such interoperability requires the support of an appropriate protocol which both introduces the requirements for SDN communication interfaces and provides backward compatibility with existing IP routing and MPLS control plane technologies. Such a solution would reduce the cost, risk and disruption for enterprise and carrier networks transitioning to SDN.

Introducing a new protocol requires consideration of standardization and where this standardization will be of most beneﬁt. ETSI Network Function Virtualization (NFV) Industry

Speciﬁcation Group intends to standardize components within the core network that may be virtualized to provide eﬃcient scalability and placement of those services. IETF’s forwarding and Control Element Separation (ForCES) WG has been working on standardizing interfaces, mechanisms and protocols with the goal of separating the control plane from the forwarding plane of IP routers. ONF is standardizing OpenFlow as a communication protocol within the network and is driving the standards of related protocols, such as the OpenFlow management and conﬁguration protocol. Many programming languages such as Frenetic, Procera etc. are also being proposed to resolve the northbound API link.

The work of the IETF, ETSI, ONF and other industry working groups must be coordinated in order to take advantage of existing standards in networking while proposing and developing the most eﬀective standards to support migration from the traditional network model to SDN.

**Security**

****

**CONCLUSION**

SDN has emerged as a means to improve programmability within the network to support the dynamic nature of future network functions. As bandwidth demand escalates, the provision of additional capabilities and processing power with support for multiple 100GE channels will be seamless through an SDN-based update and/or upgrade. SDN promises ﬂexibility, centralized control and open interfaces between nodes enabling an eﬃcient, adaptive network.

In order to achieve this goal, a number of outstanding challenges must be resolved. In this article we have presented a discussion of a number of challenges in the area of performance, scalability, security and interoperability. Existing research and industry solutions could resolve some of these problems and a number of working groups are also discussing potential solutions.

In addition to these, the hybrid programmable architecture could be a means to counter performance and scalability issues introduced by SDN. The objective of the model is to optimize ﬂow processing in the network.

The original data networks were formed out of a combination of computing devices with data and network nodes to transfer this data between the source and destination. The ability to provide “X”-as-a-service (XaaS) through virtualization technology has increased the volume of data in the network. This has set a baseline for a new communication method by pushing computation into the network devices increasing machine-to-machine communications.

The future of networks will be shaped around this progression. The goal is to provide eﬀective communications and services where network, data and computation are fused into a service architecture. In the future, for a speciﬁc process, data will request the computing, storage and connection that it requires before launching the application. The location of the network elements might be distributed physically and virtually but this will be entirely opaque to the end user. All the user will observe is the quality of delivery of the requested service.

SDN will contribute to this vision of future communications. However, signiﬁcant issues must be addressed in order to meet expectations. Indeed consideration of the potential for application-driven networks might lead us to wonder whether SDN as currently envisioned is even suﬃcient. Nevertheless, it is certain that SDN is here to stay as an evolutionary step for paving the way for a highly optimized ubiquitous service architecture.

**SDN Architecture Overview**

The SDN architecture allows an SDN controller to manage a wide range of data plane resources. A number of different data planes exist; SDN offers the potential to unify and simplify the configuration of this diverse set of resources.

The architecture also recognizes the reality that if SDN is to be successful, it must be deployable within the context of largely pre-existing multi-player environments, comprising many organizations or businesses, with the consequent need for policy and security boundaries of information sharing and trust. Real-world constraints include the need to co-exist with existing business and operations support systems, and other administrative or control technology domains. In less complex environments, such as limited scale enterprise networks, suitable functional subsets may be profiled from the architecture

The SDN architecture recommends that common models and mechanisms be employed wherever possible to reduce standardization, integration and validation efforts. This also implies utilizing existing standards or accepted best practices where feasible.

A systems architecture partitions a complex system into modular parts, typically used to manage complexity, to allow for independent implementation and component reuse, or to meet other technical or business goals. However, there is no such thing as value-neutral design. The choice of component partitioning, which interfaces are defined, which protocols are open or proprietary, can have a profound influence on the types of services ultimately delivered to the end user. Thus, an architecture necessarily makes choices; the choices and their rationale are presented in this document. This architecture contents itself with principles, rather than detail, expecting that clearly enunciated principles facilitate the myriad decisions required by working groups and implementers. At the same time, the architecture recognizes that SDN addresses environments sufficiently complex to require future extensions and clarifications. Implementation considerations are described, along with topics for further study

**1 SDN overview**

This section describes the architecture in two ways. Section 1.1 is a high-level descriptive overview, while Section 1.2 describes the essentials of the architecture as concisely as possible. The remainder of the document derives and explains the architecture, and expands on some of its implications.

* 1. Descriptive overview

The aim of SDN is to provide open interfaces that enable the development of software that can control the connectivity provided by a set of network resources and the flow of network traffic though them, along with possible inspection and modification of traffic that may be performed in the network. These primitive functions may be abstracted into arbitrary network services, some of which may not be presently apparent.

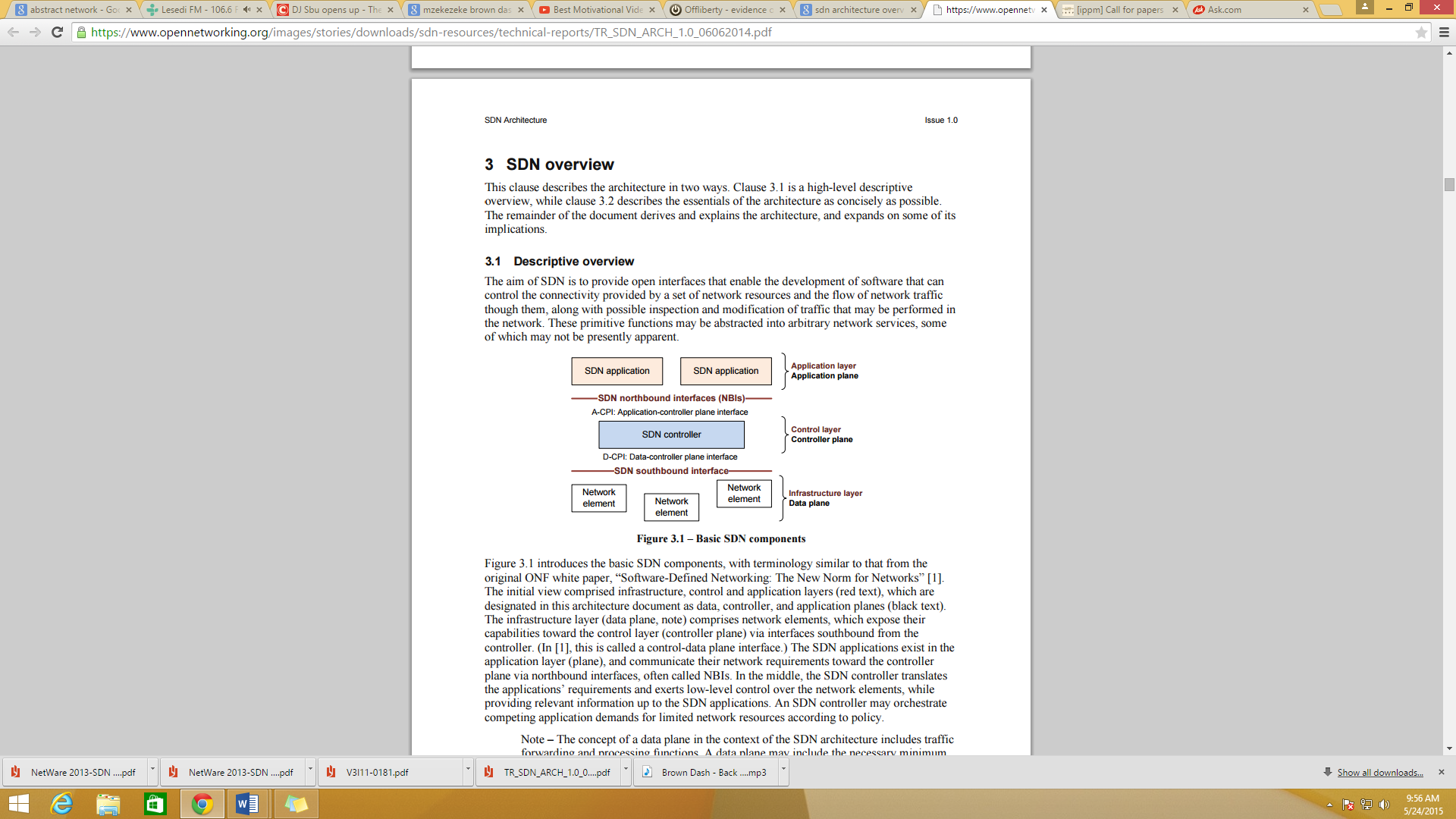


Figure 1.1 Basic SDN Components

Introduces the basic SDN components, with terminology similar to that from the original ONF white paper, “Software-Defined Networking: The New Norm for Networks”. The initial view comprised infrastructure, control and application layers (red text), which are designated in this architecture document as data, controller, and application planes (black text). The infrastructure layer (data plane, note) comprises network elements, which expose their capabilities toward the control layer (controller plane) via interfaces southbound from the controller. (In [1], this is called a control-data plane interface.) The SDN applications exist in the application layer (plane), and communicate their network requirements toward the controller plane via northbound interfaces, often called NBIs. In the middle, the SDN controller translates the applications’ requirements and exerts low-level control over the network elements, while providing relevant information up to the SDN applications. An SDN controller may orchestrate competing application demands for limited network resources according to policy.

This view requires further development and precision if it is to provide a rigorous technical SDN architecture that can inform technically versed network architects inside and outside of ONF. This architecture document therefore defines functions, interfaces and components, explains their relations and guides the development of information models, while not over-specifying.

Terminology modifications reflect the fact that some aspects of control inevitably reside in all layers, but the interface of interest is that between an SDN controller and its adjacent entities. The major horizontal groupings are called planes to avoid confusion with the term layer, which is used in the sense of layer networks, for example when packets are mapped to MPLS, further into Ethernet, and further into wavelengths.

With that in mind, figure 1.2 adopts the revised terminology and adds the management function, which is often omitted from simplified SDN representations. Although many traditional management functions may be bypassed by the direct application-controller plane interface (ACPI), certain management functions are still essential. In the data plane, management is at least required for initially setting up the network elements, assigning the SDN-controlled parts and configuring their SDN controller. In the controller plane, management needs to configure the policies defining the scope of control given to the SDN application and to monitor the performance of the system. In the application plane, management typically configures the contracts and service level agreements (SLAs). In all planes, management configures the security associations that allow distributed functions to safely intercommunicate.

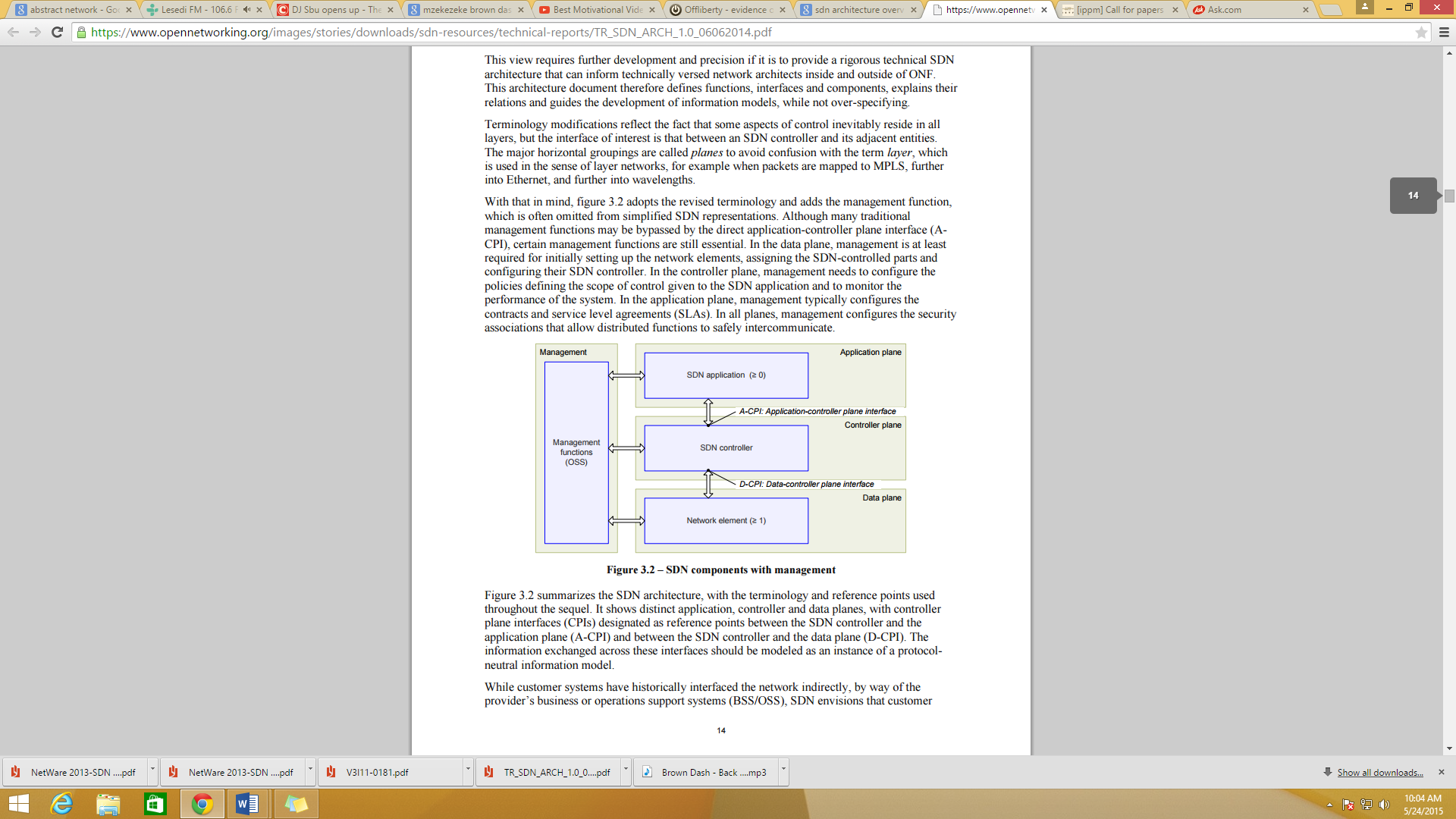


Figure 1.2 SDN components with management

Figure 1.2 summarizes the SDN architecture, with the terminology and reference points used throughout the sequel. It shows distinct application, controller and data planes, with controller plane interfaces (CPIs) designated as reference points between the SDN controller and the application plane (A-CPI) and between the SDN controller and the data plane (D-CPI). The information exchanged across these interfaces should be modeled as an instance of a protocol neutral information model.

While customer systems have historically interfaced the network indirectly, by way of the provider’s business or operations support systems (BSS/OSS), SDN envisions that customer applications may have dynamic and granular control of network resources through direct access to an SDN controller. Recognizing the likelihood of a business boundary between provider and customer, it is therefore essential that the architecture recognize a business or organizational boundary between the SDN controller plane and the applications that use it. Provider and customer exist in different trust domains

This architecture document uses colors as a visual aid to emphasize trust domains. Blue is the default, and may be thought of as a network provider, while other colors, such as green and red, indicate customers, tenants, or even distinct organizational or application entities within the overall Blue trust domain.

Figure 1.2 thus shows only a single trust domain. Figure 1.3 extends the idea to show multiple trust domains. Each trust domain is understood to have its own management functionality. Trust domains may logically extend into components of other trust domains, as exemplified by the green and red agents in the blue SDN controller.

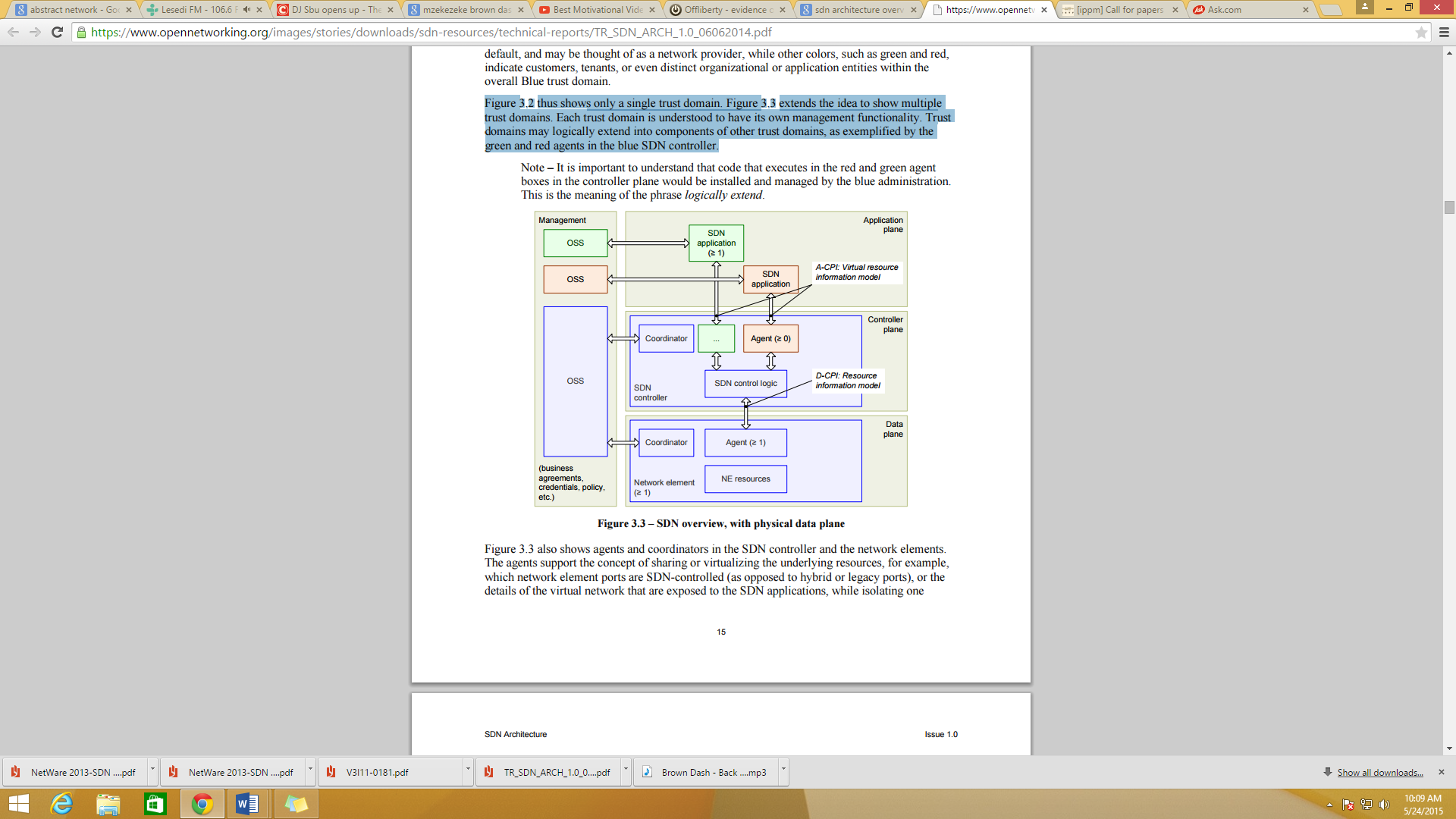


Figure 1.3 SDN overview, with physical data plane

Figure 1.3 also shows agents and coordinators in the SDN controller and the network elements. The agents support the concept of sharing or virtualizing the underlying resources, for example, which network element ports are SDN-controlled (as opposed to hybrid or legacy ports), or the details of the virtual network that are exposed to the SDN applications, while isolating one customer’s service from another’s. In the SDN controller, different agents may expose control over the network at different levels of abstraction (latitudes) or function sets (longitudes). It is the SDN control logic’s task to map and arbitrate between the networking requirements from all SDN applications and translate them into instructions for the network element (NE) resources exposed through the NE agents. The coordinators in both the network element and the SDN controller install customer-specific resources and policies received from management.

Multiple agents may exist at the same time in any one network element and SDN controller, but there is only one logical management interface, and therefore only one coordinator per network element or SDN controller.

Section 2 considers the meaning and implication of the SDN principles in further depth, and introduces the major entities whose functions and interactions comprise the architecture. Because the SDN controller is at the heart of the architecture, section 3 further expands controller plane functions and interactions, while section 4 describes implementation considerations.

The ONF SDN architecture is also summarized in a document entitled SDN architecture overview [4]. In the event of discrepancy between this document and the architecture overview, this document shall prevail.

1.2 Concise statement of architectural essentials

Figure 1.4 shows the major components and interfaces of the SDN architecture. The architecture makes no statement about the physical realization of the components.

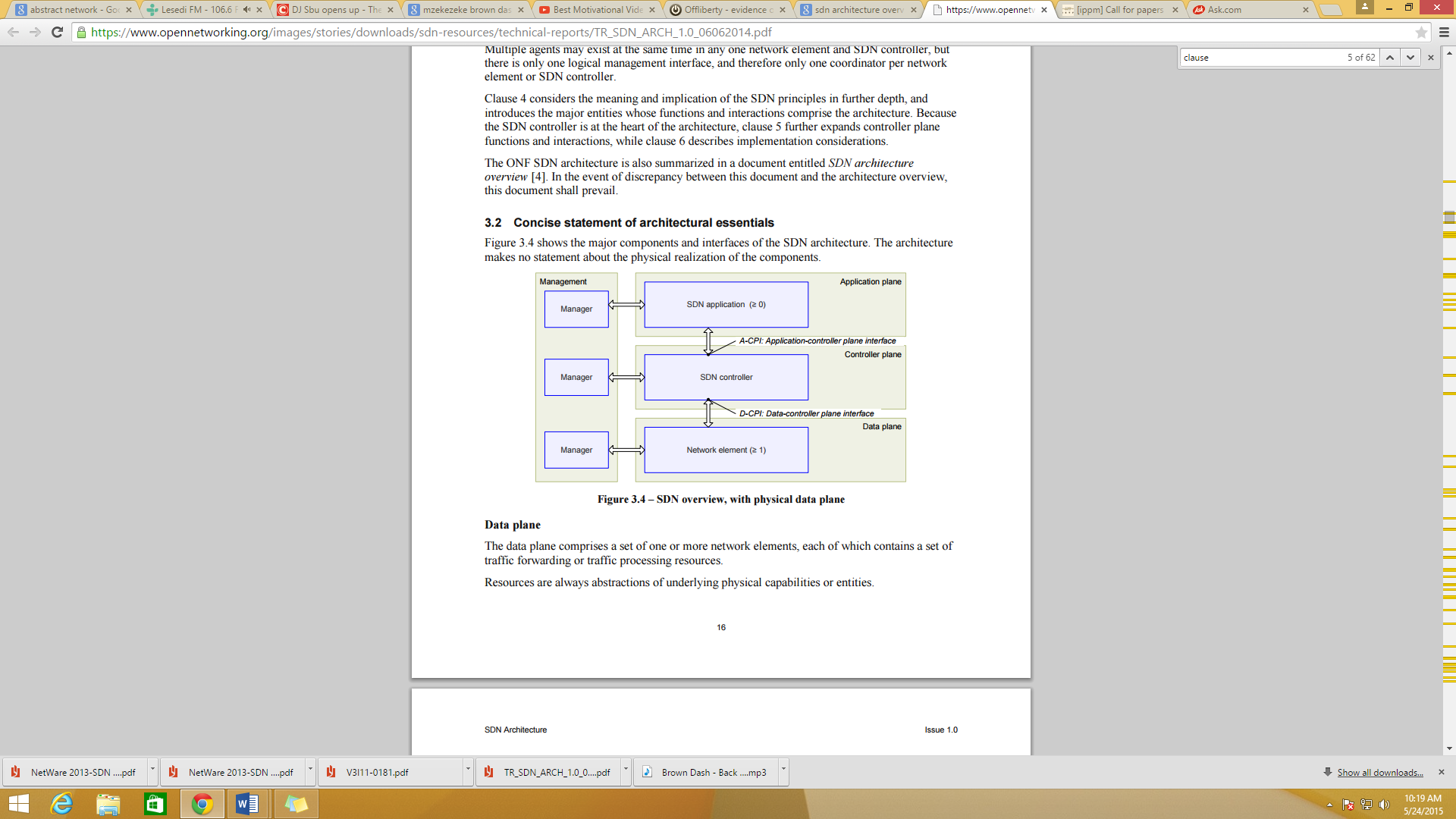


Figure 1.4 – SDN overview, with physical data plane

Data plane

The data plane comprises a set of one or more network elements, each of which contains a set of traffic forwarding or traffic processing resources.

Resources are always abstractions of underlying physical capabilities or entities.

Controller plane

The controller plane comprises a set of SDN controllers, each of which has exclusive control over a set of resources exposed by one or more network elements in the data plane (its span of control)

Section 1.3.5 explains how resources can be shared on a best-efforts or first-come-first-served basis.

Additional interfaces to SDN controllers are not precluded.

The minimum functionality of the SDN controller is to faithfully execute the requests of the applications it supports, while isolating each application from all others.

To perform this function, an SDN controller may communicate with peer SDN controllers, subordinate SDN controllers, or non-SDN environments, as necessary.

A common but non-essential function of an SDN controller is to act as the control element in a feedback loop, responding to network events to recover from failure, optimize resource allocations, or otherwise.

Application plane

The application plane comprises one or more applications, each of which has exclusive control of a set of resources exposed by one or more SDN controllers. Additional interfaces to applications are not precluded.

An application may invoke or collaborate with other applications. An application may act as an SDN controller in its own right.

Management

Each application, SDN controller and network element has a functional interface to a manager.

The minimum functionality of the manager is to allocate resources from a resource pool in the lower plane to a particular client entity in the higher plane, and to establish reachability information that permits the lower and higher plane entities to mutually communicate. Additional management functionality is not precluded, subject to the constraint that the application, SDN controller, or NE have exclusive control over any given resource.

Administration

Each entity in a north-south progression through the planes may belong to a different administrative domain. The manager is understood to reside in the same administrative domain as the entity it manages.

ONF protocols

The OF-config protocol is positioned to perform some of the functions that are needed at the management interface. The OF-switch protocol is positioned to perform some of the functions that are needed at the DCPI and possibly at the A-CPI.

**2 Principles and architectural components**

This clause introduces the principles of SDN, and the functional entities and relationships that form the SDN architecture. Subsequent clauses expand on the introductory material to derive additional constituent entities and relationships.

2.1 Principles

The ONF high-level view of SDN is described in [1]. From this and other sources, several basic principles of SDN may be adduced. Their implications are briefly summarized here, and are expanded in detail in subsequent clauses.

* Decoupling of controller and data planes

This principle calls for separable controller and data planes. However, it is understood that control must necessarily be exercised within data plane systems. The D-CPI between SDN controller and network element is defined in such a way that the SDN controller can delegate significant functionality to the NE, while remaining aware of NE state. Section 2.3 lists criteria for deciding what to delegate and what to retain in the SDN controller itself.

* Logically centralized control

In comparison to local control, a centralized controller has a broader perspective of the resources under its control, and can potentially make better decisions about how to deploy them. Scalability is improved both by decoupling and centralizing control, allowing for increasingly global but less detailed views of network resources. SDN controllers may be recursively stacked for scaling or trust boundary reasons, a topic described in clause 5.

* Exposure of abstract network resources and state to external applications

Applications may exist at any level of abstraction or granularity, attributes often described as differing latitudes, with the idea that further north suggests a greater degree of abstraction. Because an interface that exposes resources and state can be considered a controller interface, the distinction between application and control is not precise. The same functional interface may be viewed in different lights by different stakeholders. Just like controllers, applications may relate to other applications as peers, or as clients and servers

The principle of abstracting network resources and state to applications via the A-CPI allows for programmability of the network. With information about resources and their states, applications are able to specify requirements and request changes to their network services via the SDN controller, and to programmatically react to network states. Further, the concept of hierarchically recursive application/controller layers and trust domains also allows application programs to be created that may combine a number of component applications into a more comprehensive service.

The SDN architecture clarifies the meaning and implications of these principles by identifying the basic functional entities and the information and operations that need to be exchanged over various interfaces among them. The architecture further decomposes these functional entities into a not necessarily comprehensive set of functional components.

This architecture incorporates the concept of trust domain boundaries, which is vital to widespread commercialization. The architecture defines components entirely within particular trust domains, with well-defined reference points to other trust domains. Strong abstraction barriers help to protect the commercial and business interests of stakeholders, while recognizing and accommodating widely varying trust relationships. The uniformity of the architecture also facilitates the design and audit of security measures.

The high-level model of all vertical SDN architecture interfaces is the exposure of an information model instance by a server to a client, upon which the client can perform create read-update-delete (CRUD) and class-specific operations. This emphasizes the importance of a common information model throughout. From this perspective, the management function is responsible for instantiating information models and policies that define the capabilities exposed across interfaces between planes, especially across trust domain boundaries. Figure 4.1 illustrates the notion that the client-server (or controller-agent) model is applicable at as many levels of SDN controller hierarchy as may exist.

Hierarchical levels serve two purposes.

1. Scaling and modularity: each successively higher level has the potential for greater abstraction and broader scope.
2. Security: each level may exist in a different trust domain. The level interface is a standard reference point for inter-domain security enforcement.

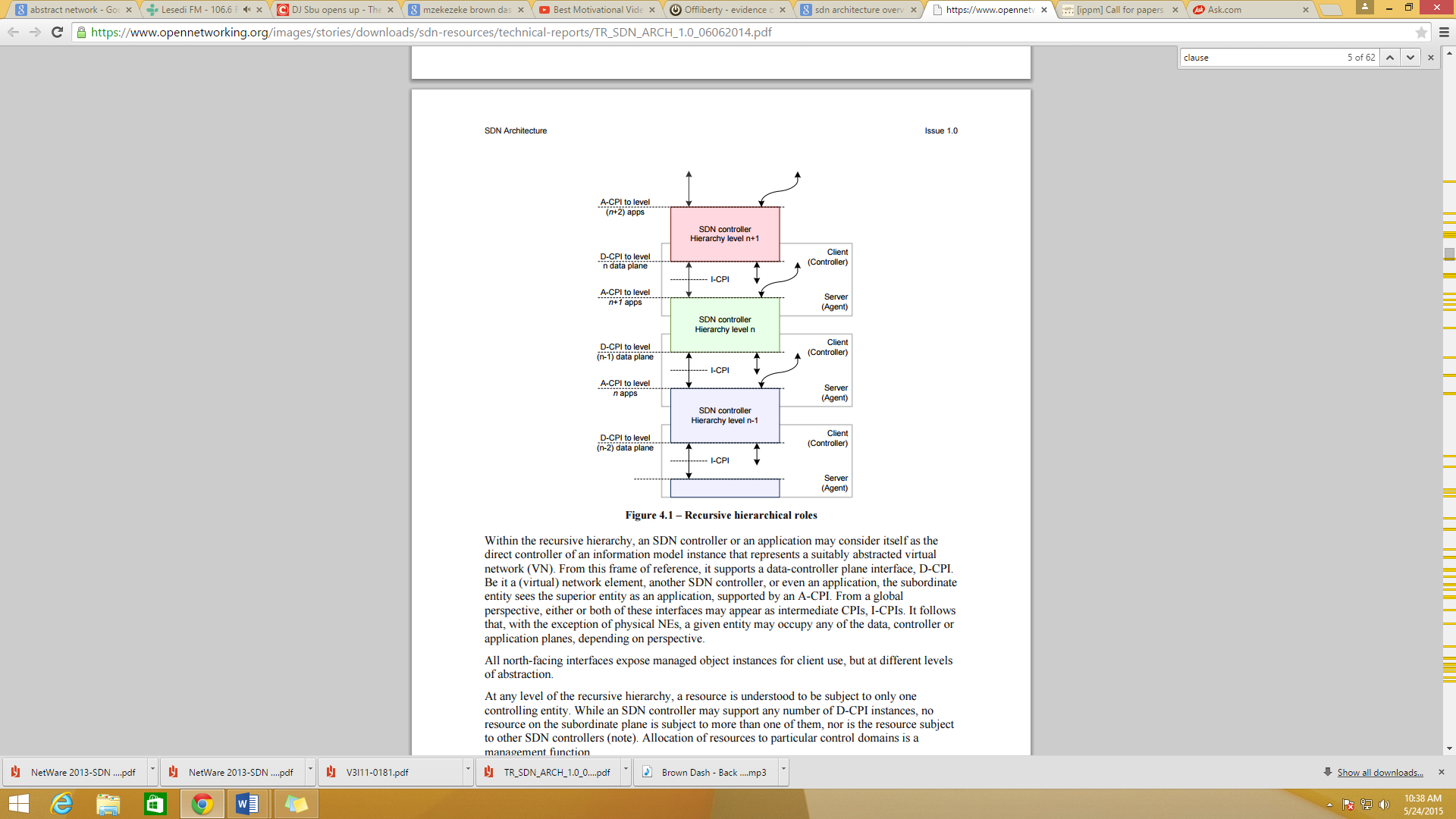


Figure 2.1 – Recursive hierarchical roles

Within the recursive hierarchy, an SDN controller or an application may consider itself as the direct controller of an information model instance that represents a suitably abstracted virtual network (VN). From this frame of reference, it supports a data-controller plane interface, D-CPI. Be it a (virtual) network element, another SDN controller, or even an application, the subordinate entity sees the superior entity as an application, supported by an A-CPI. From a global perspective, either or both of these interfaces may appear as intermediate CPIs, I-CPIs. It follows that, with the exception of physical NEs, a given entity may occupy any of the data, controller or application planes, depending on perspective.

All north-facing interfaces expose managed object instances for client use, but at different levels of abstraction. At any level of the recursive hierarchy, a resource is understood to be subject to only one controlling entity. While an SDN controller may support any number of D-CPI instances, no resource on the subordinate plane is subject to more than one of them, nor is the resource subject to other SDN controllers (note). Allocation of resources to particular control domains is a management function.

Some applications may require all-or-nothing semantics, that is, transactional integrity (the ACID property: atomicity, consistency, isolation, durability) (note). Expansion of global and abstract operations invoked by such an application implies transactional semantics at each lower level of abstraction, continuing all the way down into the hardware. Further, failed transactions must not leave behind stranded resources. Each level of hierarchy is recursively responsible for orchestrating the transactional semantics of its subordinate entities.

2.2 Data plane

The data plane incorporates the resources that deal directly with customer traffic, along with the necessary supporting resources to ensure proper virtualization, connectivity, security, availability, and quality. Figure 4.2 expands the NE resources view of figure 3.3 accordingly. The NE resources block comprises data sources, data sinks and forwarding and/or traffic processing engines, as well as a virtualizer whose function is to abstract the resources to the SDN controller and enforce policy. This expansion of detail also introduces a master resource data base (RDB), the conceptual repository of all resource information known to the network element.

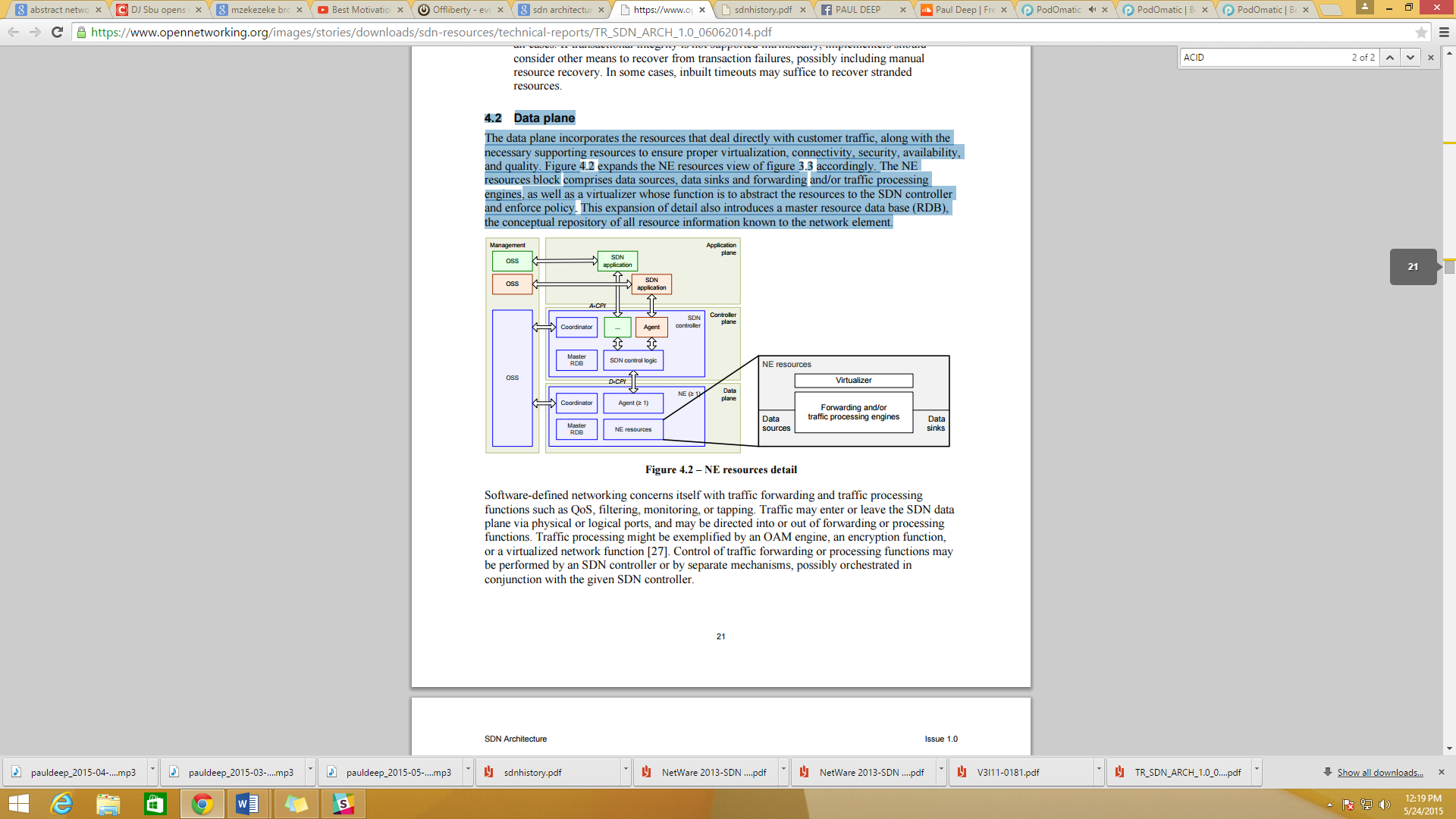


Figure 2.2 – NE resources detail

Software-defined networking concerns itself with traffic forwarding and traffic processing functions such as QoS, filtering, monitoring, or tapping. Traffic may enter or leave the SDN data plane via physical or logical ports, and may be directed into or out of forwarding or processing functions. Traffic processing might be exemplified by an OAM engine, an encryption function, or a virtualized network function. Control of traffic forwarding or processing functions may be performed by an SDN controller or by separate mechanisms, possibly orchestrated in conjunction with the given SDN controller.

The data plane implements forwarding decisions made in the controller plane. In principle, it does not make autonomous forwarding decisions. However, the controller plane may configure the data plane to respond autonomously to events such as network failures or to support functions delivered by, for example, LLDP, STP, BFD, or ICMP.

The interface between data and controller planes (D-CPI) includes functions such as

* Programmatic control of all functions exposed by the RDB
* Capabilities advertisement
* Event notification

2.3 Controller plane

Although control is exercised to varying degrees in other planes (note), the SDN controller plane is modeled as the home of one or more SDN controllers. This clause describes the functional components of an SDN controller and its relation to other controllers and other administrative domains. As will subsequently emerge, not all responsibilities of the SDN controller can be allocated to specific functional components; the architecture sees no value in proliferating blocks beyond the current level.

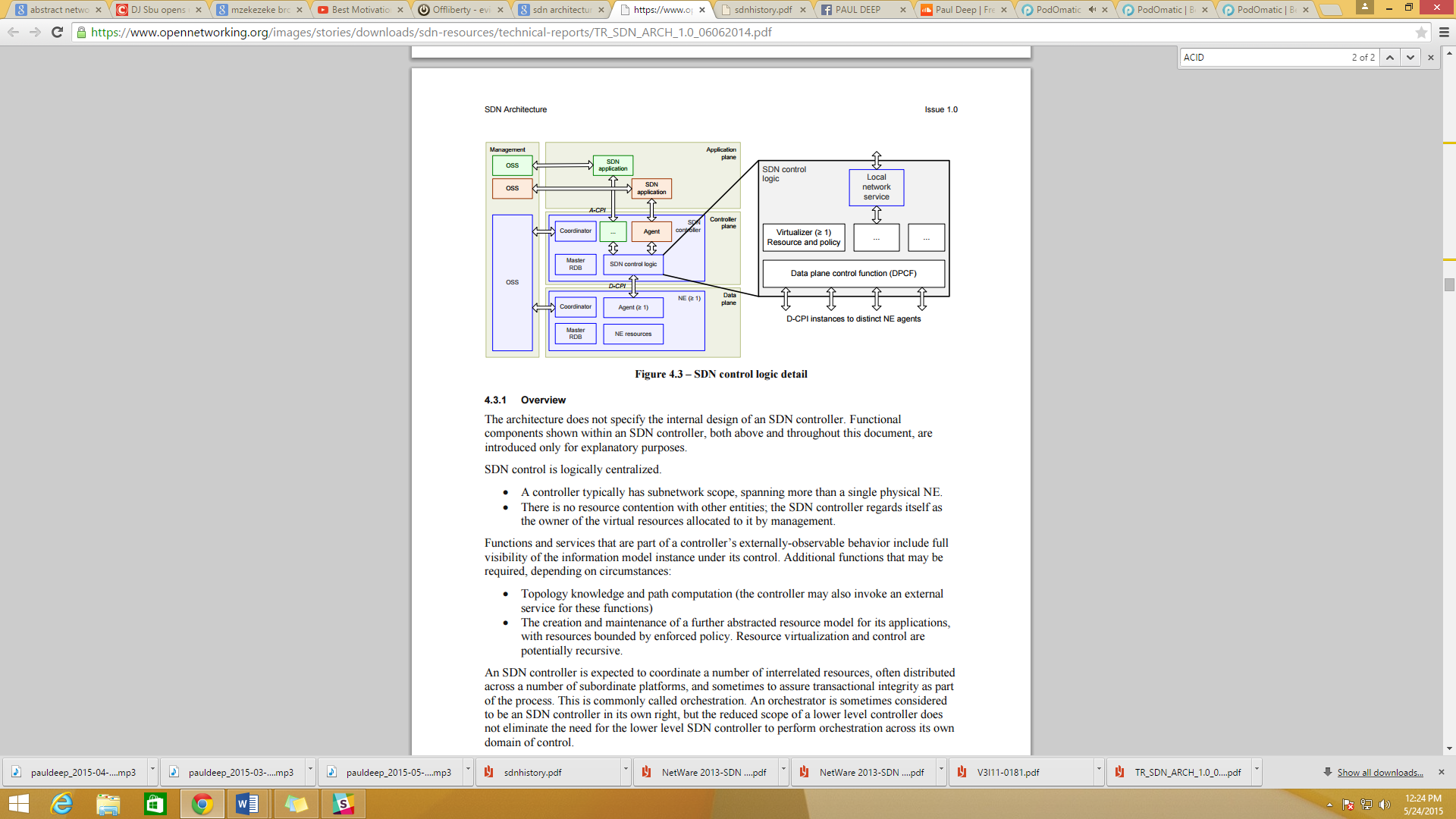


Figure 2.3 – SDN control logic detail

2.3.1 Overview

The architecture does not specify the internal design of an SDN controller. Functional components shown within an SDN controller, both above and throughout this document, are introduced only for explanatory purposes.

SDN control is logically centralized.

* A controller typically has subnetwork scope, spanning more than a single physical NE.
* There is no resource contention with other entities; the SDN controller regards itself as the owner of the virtual resources allocated to it by management.

2.3.2 SDN controller

The SDN architecture does not specify the internal design or implementation of an SDN controller. It could be a single monolithic process; it could be a confederation of identical processes arranged to share load or protect one another from failures; it could be a set of distinct functional components in a collaborative arrangement; it could subscribe to external services for some of its functions, for example path computation. Any combination of these alternatives is allowed: the SDN controller is viewed as a black box, defined by its externally-observable behavior. Controller components are free to execute on arbitrary compute platforms, including compute resources local to a physical NE. They may also execute on distributed and possibly migratory resources such as on virtual machines (VMs) in data centers.

2.3.3 SDN controller functional components

Having just stated that the SDN controller is a black box, it is nevertheless useful to conceptualize a minimum set of functional components within the SDN controller (figure 4.3), namely data plane control function (DPCF), coordinator, virtualizer, and agent. Subject to the logical centralization requirement, an SDN controller may include arbitrary additional functions. A resource data base (RDB) models the current information model instance and the necessary supporting capabilities. Clause 5 discusses the RDB and its partitions in detail.

**OpenFlow**

OpenFlow is a programmable network protocol designed to manage and direct traffic among routers and switches from various vendors. It separates the programming of routers and switches from underlying hardware

**The Problem Open Flow Solves**

OpenFlow has emerged from the need to address these critical deficiencies in networking today in order to help research bloom. OpenFlow exploits the existence of lookup tables in modern Ethernet switches and routers. These flow-tables run at line-rate to implement firewalls, NAT, QoS or to collect statistics, and vary between different vendors. However the OpenFlow team has identified a common set of functions that are supported by most switches and routers. By identifying this common set of functions, a standard way of manipulating flow-tables can be deployed for all network devices regardless of their vendor-specific implementation. OpenFlow provides this standard way of manipulating flow-tables, allowing a flow-based network traffic partition. This way, network traffic can be organized into various different flows which can be grouped and isolated in order to be routed, processed or controlled in any way desired.

OpenFlow can find great use in campus networks where isolating research and production traffic is a crucial operation. Flows can be created and maintained by a centralized entity called the controller. The controller can be extended in order to perform additional tasks such as routing and network access decisions. By removing network functionality from devices scattered along the network and centralizing it completely or locally, one can more easily control and therefore change it. The only requirements in order to implement this modification are switches that can support OpenFlow and a centralized controller process which contains the network logic. This way, the control and data plane are no longer co-located in one single network device, but separated and dynamically linked to one another. The separation of control and data plane functions and the adoption of a centrally controlled network model are concepts that have been discussed and approached by researchers before. Efforts like ForCES and SoftRouter have proposed architectures for enabling the decoupling of the control and data plane functionality of network devices, aiming in providing more efficient packet forwarding and greater flexibility in control functions. OpenFlow shares much common ground with these architectures, however inserting the concept of flows and leveraging the existence of flow tables in commercial switches today.

**The OpenFlow Components**

The main components of a controller-based OpenFlow network are:

• OpenFlow enabled switches

• Server(s) running the controller process

• Database containing the network view, a «map» of the entire topology of the network.

The OpenFlow switch, consists of a flow table containing flow entries, used to perform packet lookup and forwarding and a secure channel to the controller, through which OpenFlow messages are exchanged between the switch and the controller.

By maintaining a flow table the switch is able to make forwarding decisions for incoming packets by a simple lookup on its flow-table entries. OpenFlow switches perform an exact match check on specific fields of the incoming packets. For every incoming packet, the switch goes through its flow-table to find a matching entry. If such entry exists, the switch then forwards the packet based on the action associated with this particular flow entry.

Every flow entry in the flow-table contains:

1. **header fields to match against packets :** These fields are a ten-tuple that identifies the flow

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ingress Port | Ether Source | Ether Dst | Ether Type | VLAN Id | IP Src | IP Dst | IP Proto | Src Port | Dst Port |

The above table is used by OpenFlow to match packets against flow entries

2. **Counters to update for matching packet**: These counters are used for statistics purposes, in order to keep track of the number of packets and bytes for each flow and the time that has elapsed since the flow initiation.

3. **Actions to apply to matching packets**: The action specifies the way in which the packets of a flow will be processed. An action can be one of the following: 1) forward the packet to a given port or ports, after optionally rewriting some header fields, 2) drop the packet 3) forward the packet to the controller.

**4.2 OpenFlow use for network virtualization**

**4.2.1 Types of controllers available**

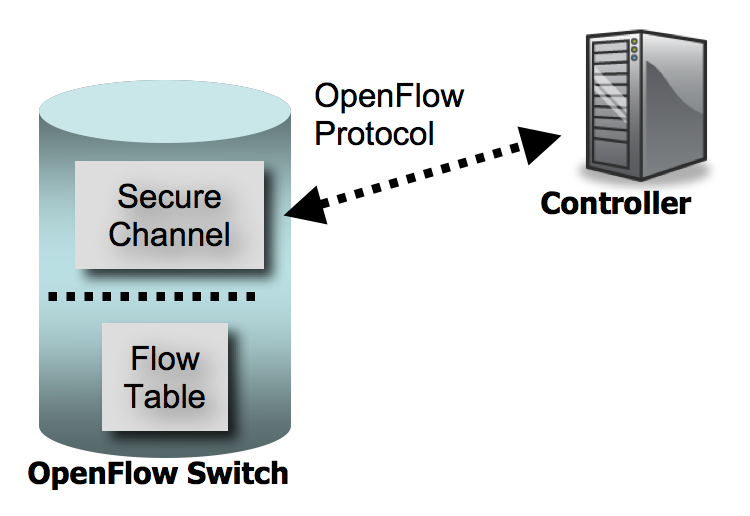
|  |  |
| --- | --- |
| **Controller** | **Main characteristic** |
| NOX | Open-source OpenFlow controller that provide a simpliﬁed platform for writing network control software in C++ or Python |
| Beacon | Beacon is a fast, cross-platform, modular, Java-based controller that supports both event-based and threaded operation. |
| Trema | Full-Stack OpenFlow Framework for Ruby/C |
| Maestro | A scalable control platform written in Java which supports OpenFlow switches |
| SNAC | SNAC is an OpenFlow controller, which uses a web-based policy manager to manage the network. |

The controller is a core, central entity, gathering the control plane functionality of the OpenFlow network. Currently there are several controller implementations available. However the most widely used and deployed is the NOX controller [14] an open-source OpenFlow controller. The controller provides an interface for creating, modifying and controlling the switch’s flow-tables. It runs typically on a network-attached server and could either be one for the entire set of OpenFlow switches on the network, one for each switch or one for each set of switches. Therefore the control functionality of the network can be completely or locally centralized according to how the delegation of switch management to controllers is performed. The requirement, however, is that if there are more than one controller processes, they should have the same view of the network topology at any given time. The network view includes the switch-level topology; the locations of users, hosts, middleboxes, and other network elements and services. Moreover it includes all bindings between names and addresses. In NOX the controller creates the network view by observing traffic related to services such as LLDP, DNS and DHCP.

It should be clear by now that, although the term „forwarding‟ is used, this does not refer to L2 forwarding. This is because the examined fields include L3 information. Similarly, an OpenFlow switch does not perform L3 routing. There is no longest-prefix-match, or any other complicated calculation that takes place on the switch. In fact, the protocol does not define how the forwarding decisions for specific header fields (i.e. the actions) are made. The decisions are made by the programmable controller and are simply installed in the switches‟ flow tables. OpenFlow switches address the flow tables and match the incoming packets‟ header fields to pre-calculated forwarding decisions, and they simply follow these decisions.

The secure channel is the interface that connects each OpenFlow switch to a controller. Through this interface the controller exchanges messages with the switches in order to configure and manage them.

**OpenFlow architecture**



The decisions are made by the programmable controller and are simply installed in the switches‟ flow tables. OpenFlow switches address the flow tables and match the incoming packets‟ header fields to pre-calculated forwarding decisions, and they simply follow these decisions.

The secure channel is the interface that connects each OpenFlow switch to a controller. Through this interface the controller exchanges messages with the switches in order to configure and manage them.

OpenFlow provides a protocol for communication between the controller process and the OpenFlow switches. There are three types of messages supported by the OpenFlow protocol. The controller-to- switch, the asynchronous and the symmetric messages. We will briefly describe these three types of messages. For further study, the OpenFlow specification [1] provides an excellent source of information.

The controller-to-switch messages are initiated by the controller and may not always require a response from the switch. Through these messages the controller configures the switch, manages the

Switch’s flow table and acquires information about the flow table state or the capabilities supported by the switch at any given time.

The asynchronous messages are sent without solicitation from the switch to the controller and denote a change in the switch or network state. This change is also called an event. One of the most significant events is the packet-in event which occurs whenever a packet that does not have a matching

Flow entry reaches a switch. When this happens, a packet-in message is sent to the controller, containing the packet or a fraction of the packet, in order for the controller to examine it and determine which kind of flow should be established for it. Other events include flow entry expiration, port status change or other error events.

Finally, the third category of OpenFlow messages are the symmetriconous messages which are sent without solicitation in either direction. Those can be used to assist or diagnose problems in the controller-switch connection. OpenFlow Network Virtualization

A number of representative virtualization techniques and architectures have been presented in previous chapters. An interesting conclusion deriving from this short survey is that most of the available network virtualization solutions today do not intend to provide a straight-forward definition of a virtual network. The definition is certainly implied but is not the starting point for proposing a certain technique or architecture. It could be said that all existing solutions accede to the loose concept of network virtualization being the partitioning of the physical network infrastructures to logical networks and proceed in implementing it.

In the case of VLANs and VPNs the virtual network boils down to a set of endpoints identified through their MAC/IP addresses or services (essentially TCP ports) operating on them. In the context of Planet Lab and VINI virtual networks are overlays on top of the existing Internet infrastructure. GENI and FEDERICA have a more abstract and holistic approach, being technology agnostic and providing full virtualization of nodes, network devices and links.

The definition of a virtual network is a crucial step that will enable us to further propose an OpenFlow-based architecture.

5.1 Towards the definition of a virtual network

The attempt to come up with a straight-forward, concrete definition of a virtual network that would allow us to further develop an OpenFlow-based architecture invoked a number of interesting issues. Many possible approaches for defining a virtual network were examined. We will briefly mention the alternative solutions that were considered along the way and the reasons why they lacked to provide an adequate means of defining an Openflow virtual network.

35

An approach that comes naturally to mind is to think of a virtual network in terms of the endpoints, links and network devices it comprises of. Similarly to PlanetLab's or FEDERICA's approach, a virtual network can be defined as a slice which consists of a set of virtual links, endpoints and network devices. End-users can opt in, requesting a set of resources which they can utilize in their preferred way, while being assured both high levels of isolation and interconnection with other existing virtual or real networks. This definition of a virtual network is resource-based, meaning that a virtual network can be adequately defined by the set of resources it consists of. Although this kind of approach is rather straightforward and focused on the hands-on usage of virtual networks, it is very high-level and does not make use of Openflow's capabilities. As described in the previous chapter, Openflow enables the logical partitioning of network traffic in flows. Therefore, in order to leverage this unique capability, we should no longer think in terms of virtual links, end points and devices but rather in terms of traffic that goes through this set of resources. This way we can take advantage of the Openflow capability of describing traffic in terms of flows.

The next question that comes to mind is how many and which of Openflow‟s ten-tuple fields can be used to describe a virtual network. Following the same approach as VLANs for example, a set of MAC/IP addresses or services (TCP ports) can mapped to one virtual network. However this approach is not flexible enough, since it should be possible for an endpoint with a certain MAC address to belong to more than one virtual networks. And VLANs fail to cover this case. Additionally, the IP address approach assumes IP connectivity.

In order to allow generality and produce a more abstract, hence more flexible, definition of a virtual network, it is essential to impose as few restrictions as possible. Instead of using MAC/IP addresses or services to define a virtual network membership, we can assume that everything that can be found on a packet header, i.e. IP, MAC, port info or any possible combination of this information, can be used to define virtual network membership.

Instead of defining a virtual network through all possible information found on an individual packet, we can group packets in flows while preserving a certain level of generality. For example, a flow could be uniquely identified by an IP source and an IP destination address, but this statement makes the assumption that the packet is an IP packet. Therefore we need to use more fields from the packet's

36

header to be able to avoid assumptions while adding flexibility to our definition. MAC source/destination addresses are possible candidates. Following this concept, we conclude in a set of fields that could provide the building blocks for creating customized virtual network memberships.

Following this idea, virtual networks can be defined not by the purpose that they serve, but by the kind of traffic that they carry. While these two concepts sound similar, they are fundamentally disparate: Mapping a virtual network to human-recognizable ideas, such as the service it provides or the hosts that use it, is bound to provide a constrained definition. The goal of this work is to provide a realization of virtual networks that are defined solely by the traffic that flows within them. It is a way of looking at virtualization from a network point of view, instead of the users‟ point of view. In this sense, a virtual network can be defined as a sum of traffic flows. This definition will become clearer as we revisit it in paragraph 5.3.

Openflow is a protocol that provides this kind of flow-based network abstraction. As mentioned in the previous chapter, a flow can be identified by a 10-tuple which is part of the Openflow header [1]. By using flows, we can achieve the following:

• One endpoint can belong to one or more virtual networks. • The definition of a virtual network becomes general hence virtual network membership can be customized according to the special requirements of the network.

In a production network there needs to be a way to map a large amount of flows to virtual networks. This should not significantly affect the network's performance or create an increased workload for the network devices.

This chapter will go through the steps followed during the design phase, highlighting these decisions that were of critical importance to the architecture while presenting the alternative options that were available at any time. Further on, a set of terms and definitions that will be used in the context of the architecture will be presented. Finally, an overview of the final architecture will be briefly described and the high-level functionality of the various components comprising it will be discussed in detail.

37

**5.2 Design Steps**

This section describes the evolution of our design. Each paragraph describes an extension to the approach of the previous paragraph, in an attempt to deal with shortcomings of every approach. These are considerations that had to be taken into account, and are provided as an indication of the challenges involved with an abstract design. They are not necessarily included in the actual architecture proposal, which is described in section 5.3.

The first steps in our design stage were made with network virtualization for university campus networks in mind. In this context, a use case scenario of the proposed architecture would be to provide several virtual slices of the physical campus topology, each of them allocated to groups of researchers for experimental purposes. The target would be to allow research and experimentation on the available infrastructure, without affecting the operation of the campus production network. The benefits of deploying network virtualization in these environments are several, the most considerable of them being the ability to utilize large experimental topologies at no extra cost for equipment, or the need to establish distinct experimentation environments.

**5.2.1 Flow establishment**

**5.2.1.1 Preconfigured flows**

With the above scenario in mind, our initial thoughts involved the configuration of virtual slices on a centralized supervising entity, and the establishment of matching flows that instantiate the configured slices. In this first approach, the administrator configures the virtual networks by providing a description for each network to the controller. The description is a topological description of the virtual network, which involves the definition of the network devices and their interfaces that constitute the slice.

Depending on the capabilities of the controlled protocol, the slice configuration can optionally include additional information, used to customize the level of abstraction of the slice description. For example, in the case of OpenFlow, the administrators can provide fine-grained descriptions by explicitly specifying host identifiers (MAC/IP addresses), type of traffic (port), etc. However, if no additional information is provided, the slice will still be abstractly defined as a subset of the physical topology, with no host, traffic, or even protocol restrictions.

38

Upon the configuration of a slice, flow entries that match the given configuration are automatically set up on the appropriate network devices, forming the slice. This way, the flows are pre-configured, and are ready to be used whenever a host tries to use the slice from a designated location. This could refer to any host connected to a designated location or specific hosts connected on designated locations, depending on the granularity of the configuration as described in the previous paragraph.

While this simple approach provides some level of abstraction, this is far from the desirable generality. Pre-configured flow entries might be useful for virtual networks with very specific requirements, hosts and general usages (e.g. applications or type of traffic), but this kind of configuration is too static and does not favor scalability. Moreover, in order to abide by the principle of abstraction, the definition of a virtual network would ideally be independent of the network devices that comprise it, and a virtual network will not bound to a specific overlay of the physical topology, but the used paths can shift dynamically.

5.2.1.2 Dynamic flows with host identification

An alternative solution that is a step closer to the desired abstraction involved the dynamic establishment of flows upon user request. In this case, when a user would like to use a virtual slice, they would first contact this supervising entity, which, after authentication, would provide the user with a list of configured virtual slices of the network. The virtual slices listed there, would depend on the user‟s credentials, and/or the physical point of connection to the network. The user would then choose to join one of the available virtual networks, and after authentication the host would be a part of the selected slice. After that point, both the underlying production network and the other available virtual slices would be transparent to this host.

A simple analogy of the aforementioned scenario is the process of authentication to wireless LANs. In WiFi, the physical medium is the air. In a given environment, several groups of users utilize this medium by configuring and operating separate networks on top of it. A WiFi user retrieves a list of advertised beacons and chooses to use one of them. After authentication, the host‟s network stack is unaware of other networks on the physical medium. Of course this is irrelevant to virtualization, since network isolation is achieved on the physical layer. However this describes the desirable functionality. In our case, the list of available networks is provided by a supervising entity that maintains their

39

configurations, and just like WLANs are isolated and used by different groups, the virtual slices are ignorant to each other‟s existence although they share the same medium; the campus network.

This approach still raises several limitations that we wanted to avoid. One significant restriction is that each host can only operate on a single OVN at a time. This is something that contradicts the desirable generality of our objective. Virtualization becomes most efficient when it is realized in multiple forms. Network virtualization that forces a host to operate on a single slice automatically negates the benefits of server virtualization on that host. In other words, one machine would be unable to host several virtual servers that would operate on separate network slices. This kind of loss of generality should be avoided.

Moreover, in the above scenario virtual network membership identification essentially takes place on the application layer, and is based on user credentials. In this way users are aware of the slices, making virtualization an opaque function. Thus, while this design introduces a centralized virtual network management entity, it does not seem to abolish several undesirable, restricting characteristics.

The objective of our work has been to reach a definition of virtual networks which would be as abstract - and thus as customizable - as possible. A nice feature enforcing this kind of abstraction would be to completely conceal the concept of virtualization from the end users as well as the network devices. We wanted the notion of virtual slices to be manifested only on the control plane, which is in this case represented by our supervising entity, the controller.

5.2.2 Path Calculation

The above considerations also raised the problem of defining a method of selecting a physical path in order to connect two points of a slice. Even for this simple approach, the connection locations to a virtual network are only half of the required information that defines the network. In order to describe it in its entirety, we actually need to describe how these locations are connected, by setting up the flow entries along available physical paths.

It is thus obvious that there is need for an entity that, given the network topology and a set of routing requirements, can calculate a path which connects two points of the network. So the question that rises is where should this path calculating entity fit into our architecture? The selection of a

40

physical path is a separate and complex process. Moreover, considering the variable purposes of network slices, the path selection might be required to be based on different algorithms for different networks. These considerations led us to separate the path finding functionality from the rest of the architectural mechanisms.

The notion of an offline path-finding entity has been discussed a lot in the research community during the last decade. This trend has been followed by researchers working on routing protocols in general, and is not strictly related to virtualization. RFC 4655 [40] discusses a network architecture based on an offline Path Computation Entity (PCE). While the purpose of the PCE in that document is to run constrained based routing algorithms for MPLS networks, the benefits provided by a path finder as a separate finding entity are the same.

5.3 Terms and definitions

Having discussed several initial alternative approaches, we will proceed to the proposed architecture. However, first, certain terms that will be used throughout this paper should be defined:

• We will refer to any machine that sends or receives data through an OVN as an Endpoint.

• An OF switch is a switch that supports the Openflow protocol ('type 0' switch as defined in the Openflow whitepaper [1])

• The term Access Point will frequently be used for describing an ingress/egress port for the network. An access point is represented by a [switchID, switchPort] pair.

• A path is a physical route between two access points.

• The term OF 10-tuple will refer to the 10 header fields found on every packet that enters the OF network, as described in the previous chapter.

• Flow entries are forwarding rules stored in an OF switch's flow table. These rules provide a forwarding decision for an incoming packet based on its OF 10-tuple.

41

• The term flow refers to traffic that matches a specific OF 10-tuple. This definition is traffic- oriented in a sense that it only reveals information about the values of the header-fields of the packets that constitute the flow and not the path that they follow through the network. In this sense, a combination of flow entries on several OF switches instantiates a flow, binding it to a specific path.

• A sum of flows can thus adequately define a subset of the total traffic that goes through the physical network. Our approach to a virtual network is to define it in an abstract way using the traffic that goes through it. Therefore an Openflow Virtual Network (OVN), can be defined as a sum of flows. This provides the desirable abstract definition, which is based solely on the type of traffic that goes through the virtual network. Inductively, the definition of the OVN does not imply anything about the paths that the OVN traffic follows along the network, or the endpoints that use the OVN. These could and should be able to change dynamically in order to adapt to network changes or failures without affecting the OVN definition.

5.4 Additional components for the proposed architecture

The proposed architecture comprises of all the components needed for an Openflow-based network as described in the previous section, as well as some additional entities which provide OVN abstraction. These entities are the following:

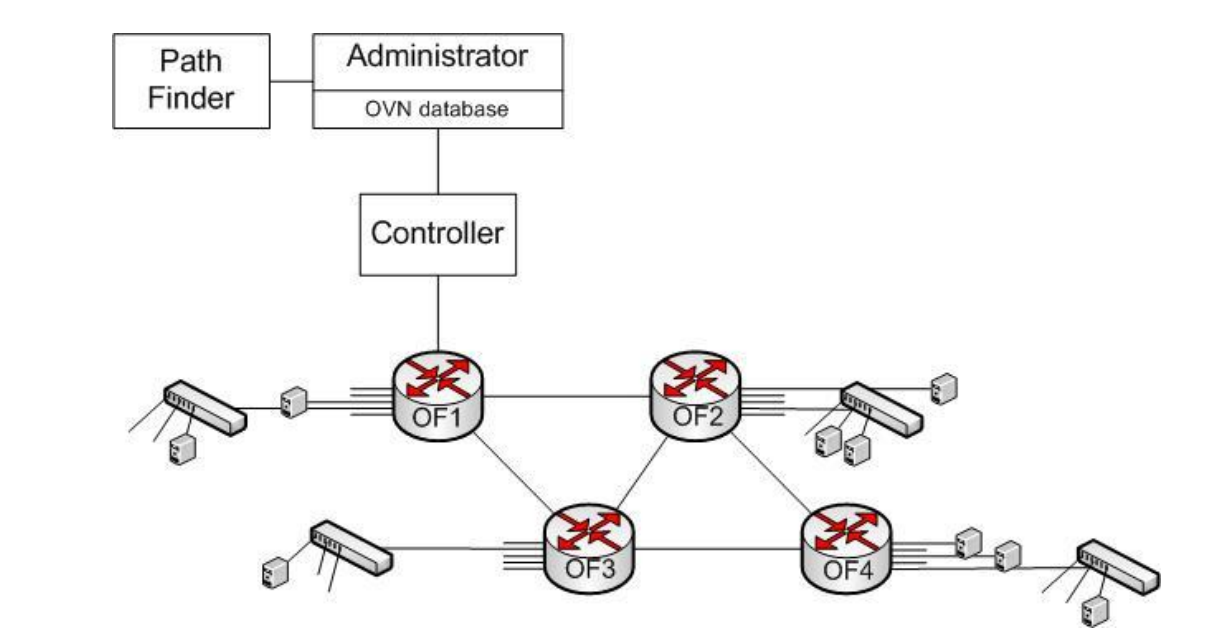
• Administrator: The administator is the entity that provides OVN abstraction to the network. It maintains a database of OVN entries called the OVN database. Furthermore it provides an interface for adding, deleting and modifying OVNs on that database. Being the only entity which has a global view of existing OVNs, the administrator is also responsible for OVN identification of a packet that is introduced to the network. The administrator could be seen as a process that runs on the same machine as the controller or a separate machine in the network. There could be one or more administrator processes as long as they have access to the same or to exact copies of the OVN database.

• Path-Finder: An entity that is responsible for making the routing decisions within an OVN. This process takes 2 access points (an ingress port, and a destination port) as input and, knowing the network view at a particular time, calculates the path that a packet should follow

42

through an OVN. This entity could also run separately or on the same machine as the controller, and there can also be one or more path-finder entities in the network.

The figure below describes the basic architecture of an OVN enabled Openflow network



**5.5 Components in detail**

**5.5.1 Administrator**

As mentioned in the previous paragraph, the administrator is the entity that provides OVN abstraction within the network. It is the only entity that can understand and interpret this abstraction. The administrator maintains and has access to the OVN database which is the core information tank of our architecture. The OVN database contains one OVN entry for each of the existing OVNs. The format of the OVN entires is shown in the following figure:

Name Id Properties Services Datapaths Auth

**Professors 1 [...]**

**Http 10.10.0.100 nl:1 port:2**

**All Yes**

**MyRouting 2 [...]**

**Dns 192.168.1.10 nl:2 port:4**

**All Yes**

**BobSlice 3 […]**

**dp2, dp4, dp5, dp9, dp10**

**No**

**43**

**Figure 6: The OVN database**

**The various fields contained in an OVN entry are explained below:**

**• Name: Name of the OVN, chosen by the OVN owner.**

**• Id: Unique identifier for the OVN, incremented for every added ovn.**

**• Properties: An array of properties of OVN traffic according to the OF 10-tuple definitions, based on which the administrator can decide if a packet belongs to this OVN. Each property can be assigned more than one value, as well as wildcard values. These properties are described in the next paragraph.**

**• Services: List of services running on the OVN, host of the service, switchID and port where the service resides.**

**• Auth: indicates whether access to the ovn requires authentication.**

**• Datapaths: Indicates which OF switches are available for this OVN. This parameter is optional. An empty field indicates that the whole physical topology is at the path-finder's disposal, and the OVN can consequently expand anywhere on the network. A non-empty value requires a path-finder that supports route calculation based on given topology restrictions. Although this field introduces path restrictions for an OVN, the fact that it is optional does not affect our OVN definition which does not associate an OVN to the paths it uses. An implementation that supports a restriction-based path finder and makes use of this field, binds an OVN to a subset of the openflow network topology. The datapath restrictions can be either loose or strict. In the case of loose restrictions, when the path-finder cannot provide a path using the preferable set of datapaths, it looks for an alternative using the whole topology. If the datapath restrictions are strict and no path can be calculated using the designated datapaths, the path-finder notifies the administrator that there is no path available.**

**The properties array is the most significant element of an OVN entry. OVN identification for a new packet is based on this array. When a packet that is received by an OF switch matches an existing flow in its table (be it either a production network flow or an OVN specific), it will be forwarded**

**44**

**accordingly. If however the packet cannot be matched to any flow, it will be forwarded to the controller. The controller will in turn consult the administrator process which will have to deduce which OVN the packet is intended for. In order to do so, it will need to match information found on the packet's OF 10-tuple to the properties that the owner of the OVN has defined for the OVN. These OVN properties are held in the OVN database and they correspond to 10 fields defined in an OF 10-tuple. The only difference is that the original ingress port field is now replaced by an access point. Each field can be assigned multiple values, for example many source MAC addresses, or many possible destination ports. The sum of all different values for every field provides a description of all the traffic that belongs to this OVN. It is the responsibility of the OVN owner to define the OVN traffic as strictly as possible. The more information the administrator is provided with, the finer the granularity of the flow entries corresponding to the OVN.**

**Connected Ports [switch, port]**

**Ether Src**

**Ether Dst**

**Ether Type**

**Vlan ID IP Src IP Dst IP Proto Src Port**

**Dst Port**

**[1,3] [1,4] [3,2] [4,1]**

**Any Any 0x0800 - Any Any - - 4444**

**Figure 7: The Properties array of an OVN entry.**

**The elements of the properties array are explained in detail below:**

**• Connected ports: A list of access points (pairs of <switchID, port>), indicating which ports of the physical topology have been enabled for the OVN.**

**• Ether type: The hexadecimal code for the ethernet protocol used in this OVN. If multiple OVNs are matched based on this, the tiebraker is one of the next properties.**

**• Ether src/dst: Ethernet addresses registered in this OVN. If a packet is originated/destined from/to one of these addresses, it can belong to this OVN. If multiple OVNs are matched based on this, the tiebraker is one of the next properties.**

**• VLAN id: reserved – could be used as ultimate tiebraker.**

**• IP src/dst: IP addresses registered in this OVN. If a packet is originated/ destined from/to one of these addresses, it belongs to this OVN. If multiple OVNs are matched based on this, the**

**45**

**tiebraker is one of the next properties.**

**• IP proto: The hexadecimal code for the ethernet protocol used in this OVN.**

**• Src/dst port : Ports on which OVN-specific applications operate.**

**5.5.2 Path-finder**

**The path-finder is the entity responsible for making routing decisions within OVNs. The path-finder provides a path in the form of a list of consecutive access points. The administrator process acquires this list, translates every two consecutive access points from the list into a flow entry for a specific switch and instructs the controller to invoke the flow entry creation in the respective switches. The process through which a path is calculated is left at the discretion of the implementation of the path- finder. The routing decision can be based on one or more of the following criteria:**

**• Conflicts (mandatory): Any newly created flow entry should not conflict with the pre-existing flow entries on a switch.**

**• Hops (optional): Select the path with the minimum number of hops**

**• Link costs (optional): Select the path with the minimum total cost**

**• Flow table sizes (optional): Select the path in a way that the average flow table size is maintained at a minimum.**

**• Datapaths (optional): Calculate a path using a subset of the available OF switches, which is provided by the administrator upon a path request**

**• User defined factors (optional): The user can define their own metrics for example request that a specific switch is not included in the path.**

**5.6 Related Work**

**Rob Sherwood et al, have proposed a special purpose Openflow application called Flowvisor [32], which enables experiments by different researchers to run simultaneously on the same Openflow network. FlowVisor creates slices (OVN equivalents) of the same physical network which are**

**46**

**independent and isolated from one another. Flowvisor uses the same virtual network definition, where a set of flows define a virtual network. However the architecture differs. Each slice is controlled by a different controller and Flowvisor acts as a proxy, siting between the experimenters' controller and the OF switches in the network. Flowvisor acts as a transparent multiplexer/demultiplexer of controller- switch communications in such a way that controllers talking to Flowvisor believe that they communicate with their own dedicated switch, although another controller might be using another part of the same switch through the Flowvisor. Similarly, in the other direction, an Openflow switch thinks that it is communicating with a single controller, although the Flowvisor might be aggregating commands from several controllers, and dispatching them to different parts of the switch. Flowvisor makes sure that packets are forwarded according to the virtual network they belong to by modifying the Openflow messages received from the OF switches and controllers accordingly. Flowvisor allows for recursive delegation of virtual networks. This means that a slice can be further delegated to one or more experiments.**

**Based on cluster computing architectures where the resources of several distributed servers are combined to form one virtual server, Fang Hao et al., [45] propose an interesting approach for implementing network virtualization. The main purpose of this architecture is to assist the migration of services in data centers. Service migration refers to services moving between different Virtual Machines distributed along the network, in order to minimize user delays and improve performance. The work proposes a centralized architecture where different FEs (Forwarding Elements) distributed along the network, form a a virtual router, called VICTOR (Virtually Clustered Open Router). The sum of different FEs can be seen as a set of input and output ports on the virtual router. A VICTOR is controlled by a centralized entity called the CC (Centralized Controller) which gathers the control plane functionality of many VICTORS. Each VICTOR is allocated for a different virtual network. The advantage of this architecture is that there is a large pool of available ports on the virtual router in different locations since the FEs are distributed along the network. This way traffic can be routed in different ways according to parameters such as the location of the end hosts.**

**Having presented the main components that comprise our architecture and go through their functionality in detail, the following chapter will focus on the high-level implementation of these components covering issues such as the communication between them.**