Chapter 3

SDN components and OpenFlow

Yanbiao Li, Dafang Zhang*, Javid Taheri**, and Keqin Li****

Today's Internet suffers from ever-increasing challenges in scalability, mobility, and security, which calls for deep innovations on network protocols and infrastructures. However, the distributed controlling mechanism, especially the bundle of control plane and the data plane within network devices, sharply restricts such evolutions. In response, the software-defined networking (SDN), an emerging networking paradigm, proposes to decouple the control and data planes, producing logically centralized controllers, simple yet efficient forwarding devices, and potential abilities in functionalities programming. This chapter presents a short yet comprehensive overview of SDN components and the OpenFlow protocol on basis of both classic and latest literatures. The topics range from fundamental building blocks, layered architectures, novel controlling mechanisms, and design principles and efforts of OpenFlow switches.

3.1 Overview of SDN's architecture and main components

In Internet Protocol (IP) networks, implementing transport and control protocols within networking devices indeed contributes to its great success in early days. However, its flexibility in management and scalability to emerging applications suffer from more and more challenges nowadays. What makes the situation worse is that the vertically integration becomes one of the biggest obstacles to fast evolutions and incessant innovations on both protocols and infrastructures. To this point, SDN [1] has been proposed, with a new architecture that decouples the control plane and the data plane of the network. Ideally, the underlying infrastructure could work as simple as an automate that processes received packets with pre-defined actions, according to polices installed by the logically centralized controller. Such a separation of control protocols from forwarding devices not only enable technologies in both sides evolve

*Computer Science and Electrical Engineering, Hunan University, China
**Department of Mathematics and Computer Science, Karlstads University, Sweden
***Department of Computer Science, State University of New York, USA

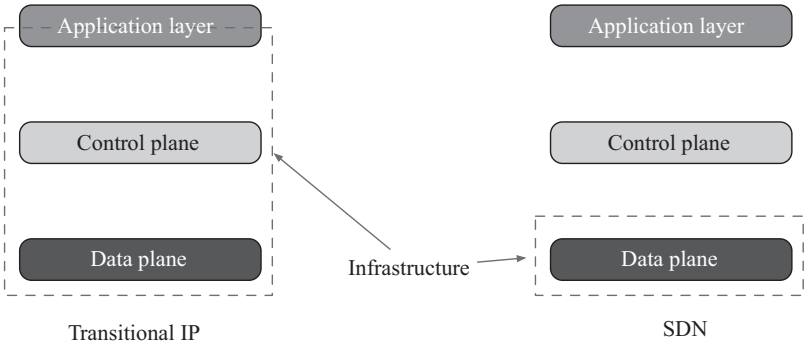

Figure 3.1 Comparison of layered architectures between IP and SDN

independently and much faster, but also simplifies the management and configuration of the whole network.

3.1.1 Comparison of IP and SDN in architectures

From the view of infrastructures, the network can be logically divided into three layers: (1) the data plane that processes network packets directly, (2) the control plane that controls the behaviour of the data plane and expresses the upper layer's requests of installing polices and applying resources, and (3) the application layer, which is composed of all applications that manages the infrastructure and that provides special network services on basis of the infrastructure. In traditional IP networks, the control plane and the data plane are tightly coupled within the same infrastructure, working as a whole middle box. Besides, some network applications, such as the Firewall, the Load Balancer, the Network Intrusion Detection System, etc., reside in the box as well.

While, as shown in Figure 3.1, SDN introduces a very different architecture. First of all, the control and data planes are completely decoupled, leaving the data plane in the network infrastructure only. By this means, networking devices are only required to play a very simple and pure role: the packet forwarding element. This will sharply simplify the design and implementation of devices, boosting technology evolution and product iteration as a result. Second, being outside the box, the control plane gains more power and flexibility. As a smart 'brain', the logically centralized controller manages all networking devices at the same time in a global view, which could balance network traffics in a fine-grained manner, improve resource utilizations globally, and provide more efficient management with desired intelligences. Last but not the least, decoupling control logics from the infrastructure also opens up the chance of implementing all network applications in software, producing more flexibility and scalability. Furthermore, with the help of potentially enabled high-level virtualization, the network becomes highly programmable. It's even possible to produce a network service by packaging a series of basic functionality elements, as simple as programming a software from modules. This is one of the simplest perspectives to understand essential differences between traditional IP and SDN.

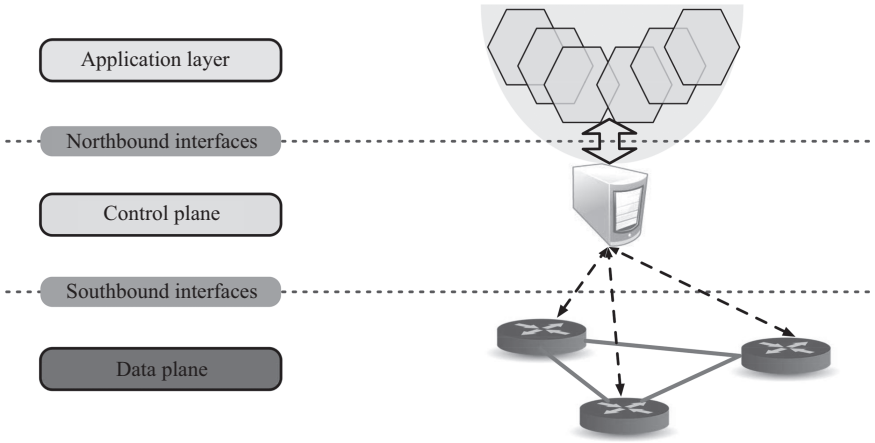

Figure 3.2 Overview of SDN's functionality layers and system architecture

3.1.2 SDN's main components

As for SDN, Figure 3.2 demonstrates its architecture more specifically. In addition to three functionality layers, there are two bridge layers, the southbound interface and the northbound interface respectively, connecting them one by one. The southbound interface layer defines the protocol associated with a series of programming interfaces for the communication between the data and the control planes. For instance, it should define the manner by which the data plane could be configured and re-configured by the control plane, the number and format of mandatory and optional arguments used in installing high level policies into the data plane, the right way and time of data plane's requesting higher level assistances, only to name a few.

Unlike the southbound interface that has clear basic responsibilities and many widely accepted proposals, the northbound interface is relatively unclear. It's still an open issue to clarify some common interfaces and standards. Learning from the development of the southbound interface, it must arise as the SDN evolves that being expected to describe some issues and solutions, manners and arguments for the communication between network applications and the controller. In the literature, there are already many discussions about northbound interfaces. Obviously, an initial and minimal standard is important for the development of SDN, a common consensus has been made out that it's too early to confine the specifications of the controller with a single abstraction right now. Although there are different application programing interfaces (APIs) provided by different implementations of the controller [2–9], we can summarize and conceive some key points here. First, it should be implemented within a software system to keep desirable flexibility. Besides, to explore all potential benefits from SDN, it should be abstracted to break the network applications' dependency to specific implementations. Last but not the least, it should support virtualization naturally, which reflects the basic motivation of SDN.

From the perspective of system design, the SDN's data plane is implemented as a series of software or hardware switches, which take the only responsibility of forwarding packets according to pre-installed polices. On the other hand, the network operating system (NOS) running on one or more commodity devices plays the role as the logically centralized controller. Through southbound interfaces, the controller initializes all switches at the beginning with some pre-defined rules, collects their statuses, controls their behaviours by updating rules, and handlers their requests when undefined events happen. While northbound interfaces can be treated as system APIs of the NOS, which is used by network applications to apply for resources, to define and enforce polices and to provide services. As those APIs may partially vary in different SDN controllers, the implementation of SDN applications still rely on the specification of the SDN controller.

Accordingly, in an classic SDN architecture, there are three main components: the controllers, the forwarding devices and the communication protocols between them. In next sections, they are discussed in detail. First, Section 3.2 introduces OpenFlow, the most popular and the most widely deployed southbound standard for SDN as of this writing. Then, Sections 3.3 and 3.4 review and analysis research topics as well as industrial attractions towards SDN controllers and forwarding devices respectively. At last, Section 3.5 concludes the whole chapter and discusses a series of open issues and future directions towards SDN's main components.

3.2 OpenFlow

As SDN's southbound interface proposals, there are already a number of protocols proposed towards different use cases [1,10–12]. ForCES [10] proposes an approach to flexible network management without changing the network architecture. OpFlex [11] distributes part of management elements to forwarding devices to add a little bit intelligence to the data plane. Protocol oblivious forwarding (POF) [12] aims at enabling the SDN forwarding plane be protocol-oblivious by a generic flow instruction set. Among them, OpenFlow, short for OpenFlow switch protocol, is no doubt the most widely accepted and deployed open southbound standard for SDN.

3.2.1 Fundamental abstraction and basic concepts

The fundamental abstraction of OpenFlow is to define the general packet forwarding process, how to install forwarding polices, how to track the forwarding process timely and how to dynamically control the process. Before stepping into details, a series of basic concepts are introduced below in groups according to the latest (as of this writing) OpenFlow specification [13].

3.2.1.1 Packet, flow and matching

A **Packet** is a series of consequent bytes comprising a header, a payload and optionally a trailer, in that order, which are treated as a basic unit to forward. Inside a packet, all control information is embedded as the **Packet Header**, which is used by forwarding

devices to identify this packet and to make decisions on how to process it. Usually, parsing the packet header into fields, each of which is composed of one or more consequent bytes and expresses a piece of special information, is the first step of processing an incoming packet.

And **Flow** is a series of packets that follow the same pattern. A **Flow Table** contains a list of flow entries, where a **Flow Entry** is a rule that defines which pattern of packets applies to this rule and how to process those packets. Besides, each flow entry has a priority for the matching precedence and some counters for tracking packets. On this basis, **Matching** is defined as the process of checking whether an incoming packet follows the pattern defined in some flow entry. All parts of a flow entry that could be used to determines whether a packet matches it are called **Match Fields**.

3.2.1.2 Action and forwarding

An **Action** is an operation that acts on a packet. An action may forward the packet to a port, modify the packet (such as decrementing the time-to-live (TTL) field) or change its state (such as associating it with a queue). Both **List of Actions** and **Set of Actions** present a number of actions that must be executed in order. There is a minor difference. Actions in a set can occur only once, while that in a list can be duplicated whose effects could be cumulated. An **instruction** may contain a set of actions to add to the action set towards the processing packet, or contains a list of actions to apply immediately to this packet. Each entry in a flow table may be associated with a set of instructions that describe the detail OpenFlow processing in response to a matching of packet. Besides, an **Action Bucket** denotes a set of actions that will be selected as a bundle for the processing packet. While a **Group** is a list of action buckets and some means of selecting one or more from them to apply on a per-packet basis.

Forwarding is the process of deciding the output port(s) of an incoming packet and transferring it accordingly. Such a process could be divided into consequent steps, each of which includes matching the packet against a specified flow table, finding out the most matching entry and then applying associated instructions. The set of linked flow tables that may be used in forwarding make up the **Forwarding Pipeline**. While **Pipeline Fields** denote a set of values attached to the processing packet along the pipeline. The aggregation of all components involved in packet processing is called **Datapath**. It always includes the pipeline of flow tables, the group table and the ports.

3.2.1.3 Communication

A network connection carrying OpenFlow messages between a switch and a controller is called **OpenFlow Connection**. It may be implemented using various network transport protocols. Then, the basic unit sent over OpenFlow connection is defined as an **Message**. A message may be a request, a reply, a control command or a status event. An **OpenFlow Channel**, namely the interface used by the controller to manage a switch, always have a main connection and optionally a number of auxiliary connections. If an OpenFlow switch is managed by many controllers, each of them will setup an OpenFlow channel. The aggregation of those channels (one per controller) is called **Control Channel**.

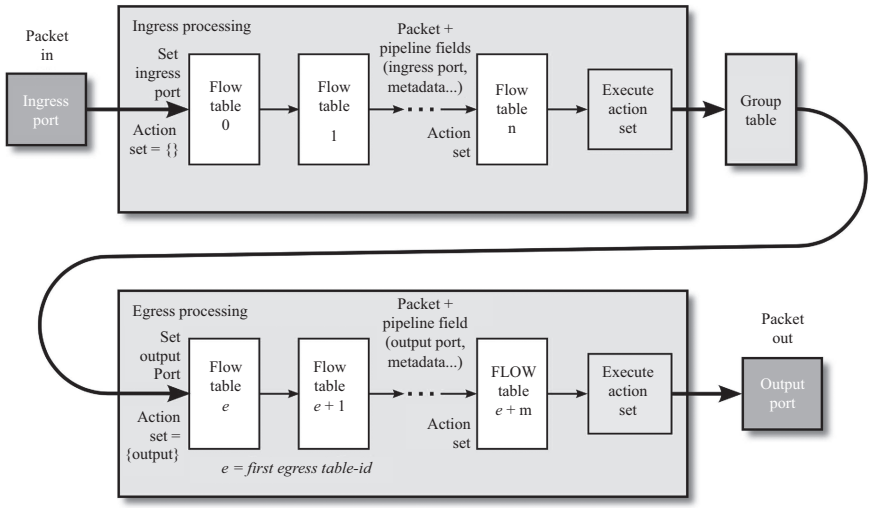

Figure 3.3 A simplified view of forwarding pipeline in OpenFlow (directly borrowed from the OpenFlow specification [13])

3.2.2 OpenFlow tables and the forwarding pipeline

This subsection describes the components of flow tables and group tables, along with the mechanics of matching and action handling.

As introduced above, an OpenFlow table contains one or more flow entries, which tells what packets could be matched and how to process them when matched. More specifically, an OpenFlow flow entry has three main components: (1) *match fields* that consists of ingress port, parts of packet headers and even metadata retrieved from previous steps, (2) *priority* that presents the matching precedence of this entry, and (3) *instructions* that may modify the action set associated with the processing packet or the forwarding process. Besides, a flow entry also has other fields for management, such as *timeouts* that denotes the time before it is being expired, *flags* that could be used to alter the way it is managed, and *cookie* that may be used by the controller to filter flow entries affected by flow statistics, flow modification and flow deletion requests. An OpenFlow table entry is uniquely identified by its match fields and priority. The flow entry wildcarding all fields (all fields omitted) and having a priority equal to 0 is called the table-miss entry, which will take effect when no other entries can match the processing packet.

During the forwarding process, all flow tables are traversed by the packet following a pipeline manner. Accordingly, they are numbered by the order they can be traversed, starting from 0. While, as Figure 3.3 depicts, pipeline processing happens in two stages, ingress processing and egress processing, respectively, which are separated by the first egress table. In another word, all tables with a lower number than that of the first egress table must be ingress tables and others works as egress tables.

Pipeline processing will start at the first ingress table (i.e. the table 0), other ingress tables may or may not be traversed depending on the outcome of the match in it. If the outcome of ingress processing is to forward the packet to some port, the corresponding egress processing under the context of that port will be performed then. It's noteworthy that egress tables are not mandatory. However, once a valid egress table is configured as the first egress table, packets must be performed on it, while other egress tables may be traversed according to the result of matching in it.

For the matching in one flow table, some header fields extracted from the processing packet, as well as some metadata transferred from previous steps, are compared to match fields of each table entry, to find out a matched entry with the highest priority. Then, the instructions associated with it are executed. The *goto-table* instruction is usually configured to direct packets from one flow table to another one whose table number is larger (i.e. the pipeline processing can only go forward). The pipeline processing will stop whenever the matched entry has not a *goto-table* instruction. Then, all actions associated with the processing packet will be applied one by one. While how to process a packet without any matching? The *table-miss* entry is configured for this purpose that defines whether miss-matched packets should be dropped, passed to other tables or sent to connected controllers.

3.2.3 OpenFlow channels and the communication mechanism

This subsection introduces types and components of OpenFlow channels, as well as underlying communication mechanisms.

As introduced earlier, an OpenFlow channel is defined, from the view of switches, as the interface connecting a switch to a controller that configures and manages it. Meanwhile, it's possible that multiple controllers mange the same switch at the same time. In this case, all channels, each of which connects the switch to one of those controllers, make up a **Control Channel**.

3.2.3.1 Control messages

OpenFlow protocol defines three types of messages exchanged between the switch and the controller: *controller-to-switch*, *asynchronous*, and *symmetric*. The essential difference among them is who is responsible to initiating and sending out the message.

As the name suggests, a *controller-to-switch* message is initiated and sent out by the controller. Those messages could be divided into two sub-groups further. One is to query status data from the switch, which therefore expects a response. For example, the controller may query the identity and basic capabilities or some running information of a switch via the **Features** requests and **Read-State** requests respectively. The other is to express control commands to the switch, which may or may not require a response. The most two popular messages in this group are **Modify-State** and **Packet-out**. Modify-State messages are primarily used to modify flow/group tables and to set switch port properties. While Packet-out messages indicate the switch to forward the specified packet along the pipeline, or to send it out on specified port(s). This type of messages must contain a full packet or the identity that could be used to locate a packet stored locally. Besides, a list of actions to be applied are mandatory as well.

An empty list means 'to drop this packet'. Besides, there is an interesting message of this type named **Barrier** that does nothing on the switch, but ensuring the execution order of other messages.

On the contrary, *asynchronous* messages are initiated on and sent out from the switch. The most important message of this type is **Packet-in**. It is usually sent to all connected controllers along with a miss-matched packet, when a *table-miss* entry towards the **CONTROLLER** reserved port is configured. Besides, the switch will also initiatively report local status changes to controllers. For example, **Port-status** messages inform the controller of any changes on the specified port, such as being brought down by users. **Role-status** messages inform the controller of the change of its role, while **Controller-status** messages are triggered when the status of the channel itself has been changed.

Being much simpler than above two types of messages, most *Symmetric* messages could be sent without solicitation in either direction and are usually used to exchange lightweight information for special purposes. For instance, **Hello** messages are triggered when connection are established, **Error** messages are used to report connection problems to the other side, while **Echo** messages that require responses are very useful in verifying the connection and sometimes measuring its latency or bandwidth. Note that there is a special symmetric message named **Experimenter**, which provides a standard way of exchanging information between switches and controllers. This would be very useful in extending the OpenFlow protocol.

3.2.3.2 Communication mechanisms

An OpenFlow controller always manages multiple switches, via OpenFlow channels connecting it from each of them. Meanwhile, an OpenFlow switch could also establish multiple OpenFlow channels towards different controllers that shares the management on it, for reliability purpose. Note that the controller and the switch connected by an OpenFlow channel may or may not reside in the same network. While OpenFlow protocol itself provides neither error detection and recovery mechanisms nor fragmentation and flow control mechanisms to ensure reliable delivery. Therefore, an OpenFlow channel is always established over transport layer security (TLS) or plain transmission control protocol (TCP) and is identified in the switch by an unique **Connection uniform resource identifier (URI)** in the format of *protocol:name-or-address* or *protocol:name-or-address:port*. If there is no port specified, port 6653 is taken as the default.

The connection is always set up by the switch through a pre-configured URI. But it's also allowed to set up the connection from the controller. In this case, the switch must be able to and be ready to accept TLS or TCP connections. Once a connection is established, it works in the same manner no matter where it's initiated. To ensure both sides work under the same version of OpenFlow protocol, they must negotiate on the version number when the connection is firstly established, by exchanging the highest version they can support through *hello* messages. Then, the negotiated version number is set as the smaller of the one was sent and the one is received. A more complicated case is when bitmap is enabled in the negotiation, where the negotiated version number should be set as the one indicated by the highest bit of the interaction of the bitmap was

sent and the bitmap is received. When the negotiated version of OpenFlow protocol is not supported in either side, the connection will be terminated immediately.

Once a connection is successfully established the version of OpenFlow protocol is negotiated, the employed transport protocol will take over on its maintenance. And all connections of a switch are maintained separately, protecting each of them being affected by the failures or interruptions on other connections. On receiving error messages, a controller or a switch can terminate the connection. Besides, whenever a connection is terminated unexpectedly, its originator is responsible to re-create it. But, in some cases such as the negotiated version of protocol is not supported, there should be no attempt to automatically reconnect.

SDN's core idea is decoupling the control and data planes, letting the logically centralised controller mange distributed switches to forward packets. But how will an OpenFlow switch work if all its connections to controllers are lost? The OpenFlow protocol also provides the answer. There are two modes of operations in that case. In the *fail secure mode*, the switch will work normally expect dropping mis-matched packets instead of forwarding them to controllers. While in the *fail standalone mode*, the switch, usually a hybrid switch, will work as a legacy Ethernet switch or router. Which one will take effect depends on the configuration.

3.3 SDN controllers

In SDN, the controller is the key component to enable highly elastic network management over networking infrastructures. It provides abstractions for connecting and communicating with forwarding devices, accessing underlying resources, generating and maintaining device configurations, and forwarding polices, to name only a few.

3.3.1 System architectural overview

From the perspective of the system architecture, SDN controllers can be divided into two main groups: centralized controller and distributed controllers.

As shown in Figure 3.4(a), a centralized controller is a single entity that manages all forwarding devices of the network. NOX [2] is the firstly proposed SDN controller that supports the OpenFlow protocol. It, especially its Python version (POX), plays an important role for prototyping SDN applications. Besides, it's the technical and architectural basis of many emerging controllers, such as NOX-MT [3] that improves NOX's performance by utilising the computing power of multi-core systems. To satisfy the ever-increasing requirements of throughput, especially for enterprise class networks and data centres, most centralized controllers [3,4] are proposed as highly concurrent systems, exploring the parallelism of multi-core platforms to boost the performance. As a popular instance, Beacon [4] has been widely adopted in both research experiments and industrial deployment (like Amazon), for its high performance, scalability, and stability. Its success owns to its modular and cross-platform architecture, as well as its easy-to-use programming model and stable user interfaces.

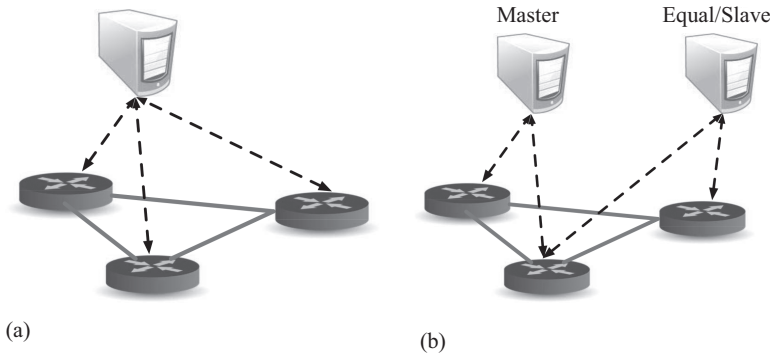

Figure 3.4 System architectures of SDN controllers: (a) centralized architecture and (b) distributed architecture

Centralized controllers do contributed to SDN's deployment, development and application innovations in early days. However, they may have scaling limitations, which prevents them being adopted to manage a large number of data plane elements. First, the resources in one single entity is limited. Second, in a large-scale network, no matter where to deploy the controller there must be some forwarding devices suffering from long latencies, for configuration and real-time management. Last but not the least, the centralized controller also represents a single point of failure and the bottleneck of the security protection.

In contrast, distributed controllers could be more scalable to meet potential requirements of both small and large-scale networks. As shown in Figure 3.4(b), a distributed controller consists of a set of physically distributed elements, which therefore could be more resilient to different kinds of logical and physical failures. However, since any controller node within a distributed controller must maintain at least one connection to a forwarding device, to balance the load among all controller nodes is important. In view of this, some proposals [8,9] focus on balancing the load among distributed controllers. As an example, ElastiCon [8] proposes a series of novel mechanisms to monitor the load on each controller node, to optimize the load distribution according to the analysis of global status, and to migrate forwarding devices from highly loaded controller nodes to lightly loaded ones. But its distribution decisions are always made upon a pre-specified threshold, which cannot be guaranteed optimal as the network grows.

Another issue of distributed controllers is the consistency semantics. Most existing controllers, such as DIStributed SDN COntroller (DISCO) [5], all have low consistency. More specifically, within those controllers, different nodes may learn different values of the same property sometime, because data updates cannot spread to all nodes immediately. Currently, only a few proposals such as Onix [6], and SMaRt-Light [7] provide relatively strong consistency, which at least ensures all nodes read the latest value of some property after a write operation. But the cost is the performance.

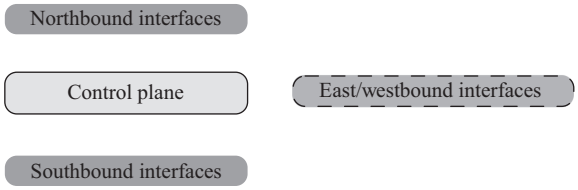

Figure 3.5 Overview of SDN controllers' components

3.3.2 System implementation overview

No matter what architecture the controller follows, there are some common components to implement. As shown in Figure 3.5, all controller systems consist of three mandatory components: northbound interfaces, the core control platform, and southbound interfaces. While for distributed controllers, there is another important component called east/westbound interfaces, which is used to exchange management information among all controller nodes within the same distributed controller system.

The core control system is made up by a series of service functions shared by network applications in building their systems, such as the topology discovery mechanism, notification streams, device management strategies, trust models and security mechanisms, and so on. Take security mechanisms as an example, they are critical components to provide basic isolation and security protection. For instance, rules generated by high priority services should not be overwritten with rules created by applications with a lower priority.

As mentioned above, there is no common standard for SDN's northbound APIs. In another word, how to implement the controller's northbound interfaces can vary completely. As a matter of fact, existing controllers implement a broad variety of northbound APIs according to application requirements and environment features, such as ad-hoc APIs, multi-level programming interfaces, file systems, among other more specialized APIs such as network virtualization platform (NVP) northbound API (NBAPI) [6]. Besides, there is another emerging type of northbound interfaces that focuses on building network applications from a series of basic functionality units, through specialized programming languages, such as Frenetic [14].

Southbound APIs of SDN controllers are implemented as a layer of device drivers, which provides unified interfaces to the upper layers, for deploying network applications onto existing or new devices (physical or virtual). By this means, a mix of physical devices, virtual devices (e.g. Open vSwitch (OVS) [15]) and a variety of device interfaces (e.g. OpenFlow, Open vSwitch database (OVSDB), NetConf, and simple network management protocol (SNMP)) can co-exist on the data pane. Although most controllers adopt OpenFlow as the southbound protocol, a few of them, such as OpenDaylight [16] and Onix [6], provide a range of southbound APIs and/or protocol plugins.

In a SDN controller, northbound and southbound interfaces are primarily used to communicate with network applications and forwarding devices, respectively. They work as bridges to entities in other layers. From this view, east/westbound interfaces

are very different. They work between controller nodes within the same distributed controller system. General components of east/westbound interfaces may include, but not limited to, mechanisms of exchanging data between nodes, monitoring their status, and algorithms for ensuring data consistency. It's important to have some standards in constructing east/westbound interfaces. There are many research efforts contributing to this objective, such as Onix data import/export functions [6]. What are the differences between eastbound and westbound interfaces? The 'SDN compass' [17] makes a clear distinction, where westbound interfaces are treated as SDN-to-SDN protocols and controller APIs, while eastbound interfaces are used to communicate with legacy control planes.

3.3.3 Rule placement and optimization

From the perspective of the forwarding devices, the most frequent and important task of the controller is to install and update forwarding rules. Since a controller (or a controller node of a distributed controller) may manage two or more forwarding devices, how to distribute rules generated by high-level applications over the network becomes an issue. Improper solutions may not only raise traffic between the controller and the device, but also lead to highly frequent *table-miss* operations in the OpenFlow switch.

To split the set of all generated rules and to distribute them over the network efficiently, many approaches have been proposed with different optimization models, such as minimizing the total number of rules needed throughout the network [18,19]. For instances, the One Big Switch [19] abstracts all managed switches as a single one and proposes a flexible mechanism to split and place rules. Besides, an emerging proposal [18] presents a novel dependency graph to analysis the relationship between rules, where the node indicate a rule, while the edge connecting two nodes represents the dependency between corresponding rules. Then, the rule placement problem can be transformed into classic graph problems, which could then be solved via corresponding algorithms. On the other hand, the more rules the device can hold, the more packets will get matched within the device, and the less traffic will be produced between the device and controllers.

3.4 OpenFlow switches

Like all other Internet architectures, SDN's forwarding devices are the fundamental networking infrastructures. As OpenFlow is the first and the most popular southbound standard of SDN, this section only discusses OpenFlow switches, which communicate with SDN controllers following the OpenFlow protocol.

3.4.1 The detailed working flow

Figure 3.6 demonstrates the complete flowchart of a packet going through the OpenFlow switch. As depicted, when receiving a packet, an OpenFlow switch may perform

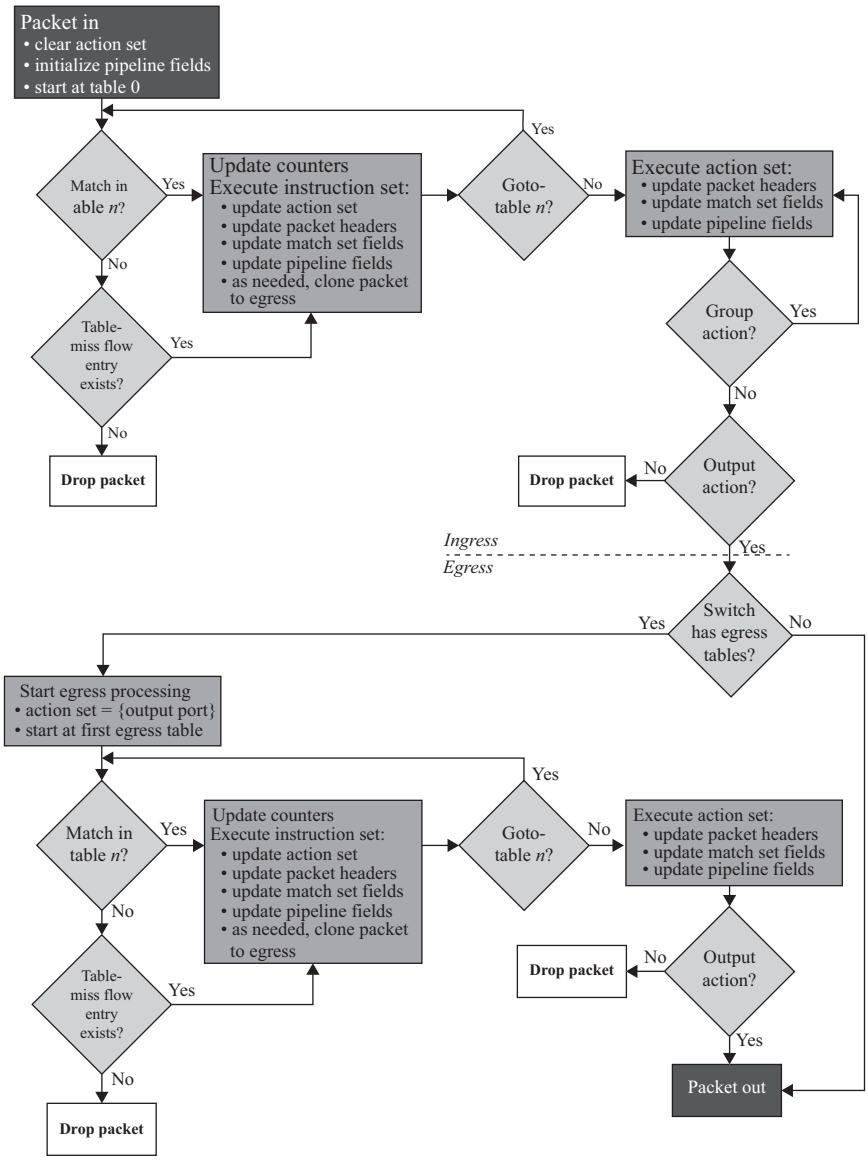

Figure 3.6 Detailed working flow of the OpenFlow switch (directly borrowed from the OpenFlow specification [13])

a series of functions in two similar pipelines: the ingress and egress pipelines, of which the latter is optional. Within each pipeline, a sequence of table lookups on different flow tables will be performed. To match a packet against a flow table, its header fields are extracted, as well as some pipeline fields. Which header fields should be used

in the matching depend on the packet type and the version of OpenFlow protocols. Generally, the fields extracted for matching include various protocol header fields, such as Ethernet source address or IPv4 destination address. Besides, the ingress port, the metadata carrying some information between two sequential tables, and some other pipeline fields that represent the status of the pipeline, may also be involved in the matching process.

A packet matches a flow entry means all match fields of this entry are carefully checked and tell matchings at last. For any match field, there are three possible cases where the processing packet can be determined to match the flow entry being compared. The first and the simplest case is when this field of the entry being compared is an omitted field that can match any value of the processing packet at this field. The second and the most common case is when this field of the entry being compared is present without any mask and its value is just equal to that of the processing packet at this field. The last but the most complicated case is when this field of the entry being compared is present with a bitmask and values of all active bits, determined by the bitmask, are equal to that of the processing packet at this field correspondingly.

It's noteworthy that a packet can match two or more entries in one flow table. In this case, the entry with the highest priority will be selected as the matched entry, the instructions and the counters associated with which will be executed and updated respectively. When a packet cannot match any regular entries, this is a *table-miss*. As a rule recommended by the OpenFlow protocol, every flow table must configure a *table-miss* entry that omits all fields so that it can match any incoming packet and has lowest priority (i.e. 0). Accordingly, the *table-miss* is only used to define how to process mis-matched packets. As a matter of fact, there possible instructions could be configured with the *table-miss* entry according current versions of the OpenFlow protocol: dropping the processing packet, forwarding it to a subsequent table, or sending it to controllers.

3.4.2 Design and optimization of table lookups

In the working flow of the OpenFlow switch, table lookup is the basic and most important operation. The design and implementation of table lookup could be divided into two related parts: the structure design of flow tables and the design and optimization of lookup algorithms.

According to the OpenFlow protocol, the essential problem under table lookup is multi-filed rule matching, which shares the model with that of packet classification. But the number of fields and the scale of tables are much larger. If every match filed of a flow entry can be transformed into a prefix (namely the mask has consequent 1s), a hierarchical tree, based on the backtrack detecting theory, could be used to store all flow entries, enabling efficient lookups to find out the most matching entry. Deploying multiple copies of some rules onto some nodes can sharply reduce the time of backtracks, boosting the matching speed as a result [20]. Besides, multi-dimensional leaf-pushing technologies [21] can lead to further improvements on performance. On the other hand, as multi-field rules and the bundle of extracted packet header fields can be seen as super-rectangles and points in a multi-dimensional space, multi-filed

rule matching can be transformed into a point locating problem. An efficient solution is to divide the space into lower dimensional spaces and then to solve simpler and similar problems recursively. For example, HiCuts [22] proposes to construct a decision tree to split the rule space, while HyperCuts [23] optimizes spatial and temporal efficiency by the multi-dimensional splitting mechanism and smart algorithms to migrate common rules. EffiCuts [24] presents a series of heuristic algorithms to achieve further memory compression. From the perspective of set processing, multi-filed rule matching can be solved by calculating the cross-products on the results of matching on rules with less fields [25]. The speed is fast, but memory consumptions will increase sharply as the number of fields increases, while HyperSplit [26] optimizes the splitting of rule projections to reduce memory consumption and utilizes the binary search to ensure processing speed.

Most existing approaches for TCP/IP packet classification suffer from the scalability issue that their comprehensive performance decreases as the number of fields increases, impeding their use in OpenFlow switches. One exception is the tuple-space-search (TSS) algorithm [27] that divides all flow entries into several groups according to the mask, ensuring that all entries in the same group share the same mask. Accordingly, the matching against any group is exact matching, which can be efficiently solved by hashing. Therefore, TSS has been adopted in the industrial level OpenFlow switches [15].

3.4.3 Switch designs and implementations

There are many types of OpenFlow switches available in the market or open source project communities. Typically, they vary in aspects, such as flow table size, performance, interpretation and adherence to the protocol specification, and architecture (e.g. hardware, software, or even heterogeneous implementations). This subsection will introduces some classic and main-stream switches grouped by the architecture.

3.4.3.1 Hardware switches

Thanks to its simple yet efficient processing logic, ternary content-addressable memory (TCAM) becomes a common choice of storing flow entries for fast lookup at early days. However, the TCAM is usually very small (can store 4k to 32k entries), costly and energy inefficient. All these drawbacks restrict its use in today's situation. That's why the open network foundation (ONF) forwarding abstraction working group works on table type patterns. In this area, most efforts focus on reducing the number of flow entries deployed onto TCAMs by novel compression techniques. Such as the Espresso heuristic algorithm [28] that can save up to 4k flow table entries by compressing wildcards of OpenFlow-based inter-domain routing tables. To keep updates consistent and rule tables away from space exhaustion, Shadow MACs [29] is proposed to employ opaque values to encode fine-grained paths as labels, which can be easily and cost-effectively implemented by simple hardware tables instead of expensive TCAM tables. Another trend of solutions is to combine many other hardware platforms with TCAMs, such as field-programmable gate array (FPGA), graphics processing units (GPUs), etc., in some specialized network processors.

3.4.3.2 Software switches

A software switch is a software programme that runs on the operating system to pull packets from the network interface cards, determine how to process them or where to forward them, and then send them out as expected. Though being a little bit slower than hardware implementations, software switches play an increasingly important role in SDN due to their scalability and flexibility, which are key factors to spread SDN's use in large, real-world networks. Open vSwitch [15] is such an software implementation of a multi-layer, open source virtual switch for all major hypervisor platforms. It is designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols. Apart from operating as a software-based network switch running within the virtual machine hypervisors, it can work as the control stack for dedicated switching hardware; as a result, it has been ported to multiple virtualization platforms, switching chipsets, and networking hardware accelerators. Switch Light is a thin switching software platform for merchant silicon-based physical switches and virtual switches within hypervisors. It provides consistent data plane programming abstractions across merchant silicon-based physical switches and hypervisor vSwitches. Switch Light OS is developed by the Big Switch company to closely integrate with whitebox hardware, where OpenFlow-like functions work well on the current generation of switching silicon for data centres.

3.4.3.3 Industrial efforts

Microchip companies like Intel are already shipping processors with flexible SDN capabilities to the market such as the proposed data plane development kit (DPDK) that allows high-level programming of how data packets shall be processed directly within network interface cards (NICs). It has been shown of value in supporting high-performance SDN software switches. On the other hand, hardware-programmable technologies such as FPGA are widely used to reduce time and costs of hardware-based feature implementations. For example, NetFPGA has been a pioneering technology used to implement OpenFlow 1.0 switches [1]. Recent developments have shown that state-of-the-art System-on-chip platforms, such as the Xilinx Zynq ZC706 board, can also be used to implement OpenFlow devices yielding 88 Gbps throughput for 1k flow entries, supporting dynamic updates as well [30].

Besides, in order to improve the performance of software switches, off-loading some parts of the switch components onto specified hardwares become a trend according to recent industrial efforts. There are two representatives made contributions to this area. Netronome's Agilio software is dedicated to off-loading and accelerating server-based networking. Agilio software and the Agilio family of intelligent server adapters (ISAs) aim at optimizing Open vSwitch as a drop-in accelerator. Its use cases include computing nodes for IaaS or SaaS, network functions virtualization, and non-virtualized service nodes, among others. Netronome Agilio ISAs provide a framework to transparent off-load of OVS. With this solution, the OVS software still runs on the server, but the OVS datapath are synchronized down to the Agilio ISA via hooks in the Linux kernel. The Agilio software is derived from the OVS codebase

and preserves all compatible interfaces. More specifically, it includes an exact match flow tracker that tracks each flow (or microflow) passing through the system. Such a system can achieve five to ten times improvement in performance. Another solution is provided by Mellanox. Mellanox's Accelerated Switching and Packet Processing (ASAP2) solution combines the performance and efficiency of server/storage networking hardware along with the flexibility of virtual switching software. There are two main ASAP2 deployment models: ASAP2 Direct and ASAP2 Flex. ASAP2 Direct enables off-loading packet processing operations of OVS to the ConnectX-4 eSwitch forwarding plane, while keeping intact the SDN control plane. While in ASAP2 Flex, some of the CPU intensive packet processing operations are off-loaded to the Mellanox ConnectX-4 NIC hardware, including virtual extensible local area network (VXLAN) encapsulation/decapsulation and packet flow classification. Evaluations demonstrates that the performance of ASAP2 Direct is three to ten times higher than DPDK-accelerated OVS.

3.5 Open issues in SDN

3.5.1 Resilient communication

For any Internet architecture, enabling resilient communication is a fundamental requirement. Accordingly, SDN is expected to achieve at least the same level of resilience as the legacy TCP/IP or other emerging architectures. However, its architecture with a logically centralized brain (i.e. the controller) is always questioned. Once such a brain is affected by kinds of faults or does not work due to some attacks, the data plane (i.e. switches) may step into a 'miss-control' state, where rules could not be updated and issues that need assistance could not be resolved timely. In this case, the whole system may become 'brainless'. Therefore, in addition to fault-tolerance in the data plane, the high availability and robustness of the (logically) centralized control plane should be carefully considered for resilient communication in SDN. In another word, there are more parts to deal with in SDN to achieve resilience, making this objective more challenging. Therefore, this topic calls for more and further research efforts in the near future to move SDN forward.

3.5.2 Scalability

For SDN, decoupling of the control and data planes contributes to its success, but also brings in more scalability concerns. Under some situations, i.e. processing a large number of tiny flows, many packets will be directed to the controller in short time periods, sharply increasing network load and make the controller a potential bottleneck. On the other hand, flow tables of switches are always configured by an outside entity, resulting extra latencies. These two issues could be ignored in small-scale networks. However, as the scale of the network becomes larger, the controller is expected to process millions of flows per second without compromising the quality of its service. Thus, in more real cases, above issues must be main obstacles to achieving the scalability purpose. Thus, improving the scalability is another hot topic now and in the future.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, *et al.* OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[2] N. Gude, T. Koponen, J. Pettit, *et al.* NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3): 105–110, 2008.

[3] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. *Hot-ICE*, 12:1–6, 2012.

[4] D. Erickson. The beacon OpenFlow controller. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined NETWORKING*, pages 13–18, 2013.

[5] K. Phemius, M. Bouet, and J. Leguay. Disco: Distributed multi-domain SDN controllers. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–4. IEEE, 2014.

[6] T. Koponen, M. Casado, N. Gude, *et al.* Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[7] F. Botelho, A. Bessani, F. M. Ramos, and P. Ferreira. On the design of practical fault-tolerant SDN controllers. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 73–78. IEEE, 2014.

[8] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed SDN controller. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 7–12. ACM, 2013.

[9] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng. Balanceflow: Controller load balancing for OpenFlow networks. In *IEEE International Conference on Cloud Computing and Intelligent Systems*, pages 780–785, 2012.

[10] A. Doria, J. H. Salim, R. Haas, *et al.* Forwarding and control element separation (forces) protocol specification. Technical report, 2010.

[11] M. Smith, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher. Opflex control protocol. *IETF*, 2014.

[12] H. Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, pages 127–132. ACM, 2013.

[13] OpenFlow specification. Version 1.5.1 (Wire Protocol 0x06). Open Networking Foundation. 2015.

[14] N. Foster, R. Harrison, M. J. Freedman, *et al.* Frenetic: A network programming language. In *ACM Sigplan Notices*, volume 46, pages 279–291. ACM, 2011.

[15] B. Pfaff, J. Pettit, T. Koponen, *et al.* The design and implementation of Open vSwitch. In *NSDI*, pages 117–130, 2015.

[16] J. Medved, R. Varga, A. Tkacik, and K. Gray. Opendaylight: Towards a model-driven SDN controller architecture. In *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[17] M. Jarschel, T. Zinner, T. Hoßfeld, P. Tran-Gia, and W. Kellerer. Interfaces, attributes, and use cases: A compass for SDN. *IEEE Communications Magazine*, 52(6):210–217, 2014.

[18] S. Zhang, F. Ivancic, A. G. C. Lumezanu, Y. Yuan, and S. Malik. An adaptable rule placement for software-defined networks. In *Proceedings of 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 88–99, Jun 2014.

[19] J. R. N. Kang, Z. Liu and D. Walker. Optimizing the one big switch "abstraction in software-defined networks, one big switch" abstraction in software-defined networks. *Proceedings of 9th ACM Conference on Emerging Networking Experiments and Technologies*, pages 13–24, 2013.

[20] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 229–240. ACM, 1998.

[21] J. Lee, H. Byun, J. H. Mun, and H. Lim. Utilizing 2-d leaf-pushing for packet classification. *Computer Communications*, volume 103, pages 116–129. Elsevier, 2017.

[22] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.

[23] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 213–224. ACM, 2003.

[24] B. Vamanan, G. Voskuilen, and T. Vijaykumar. Efficuts: Optimizing packet classification for memory and throughput. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 207–218. ACM, 2010.

[25] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. *Fast and Scalable Layer Four Switching*, volume 28. ACM, 1998.

[26] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In *INFOCOM 2009, IEEE*, pages 648–656. IEEE, 2009.

[27] F. Baboescu and G. Varghese. Scalable packet classification. *ACM SIGCOMM Computer Communication Review*, 31(4):199–210, 2001.

[28] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, 1987.

[29] K. Agarwal, C. Dixon, E. Rozner, and J. Carter. Shadow MACs: Scalable label-switching for commodity ethernet. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, pages 157–162. ACM, 2014.

[30] S. Zhou, W. Jiang, and V. Prasanna. A programmable and scalable OpenFlow switch using heterogeneous SOC platforms. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, pages 239–240. ACM, 2014.