A Software Architecture for Highly Configurable Software-Defined Networks

Ichiro Satoh
National Institute of Informatics, 2-1-2 Hitotsubashi Chiyoda-ku Tokyo, Japan

Keywords: Software-Defined Network, Software Deployment, Configurable Protocol, First-Class Object.

Abstract: We propose a novel architecture for software-defined networking (SDN) that allows for dynamic addition and modification of functionalities. Existing SDN architectures separate data transfer from software changes, making it difficult to modify the latter in response to the former. In this paper, we treat the software that defines communication protocols as first-class objects, indistinguishable from data, enabling their transfer to other computers. Our contribution to software engineering is to introduce a dynamic and unified mechanism for software components involved in network processing, which enables flexible and diverse customization. To demonstrate the utility of our proposed approach, we implemented a prototype on a Java Virtual Machine (VM) and designed and implemented several practical protocols for both data transmission and component deployment.

1 INTRODUCTION

Network requirements are diverse and constantly evolving. Real-world networks, such as sensor networks, experience changes not only in their applications but also in their requirements, which are influenced by shifts in the real world. A network may be expected to meet multiple, potentially conflicting requirements simultaneously. These requirements often necessitate modifications not only to the applications but also to the network processing itself.

Software-Defined Networking (SDN) traditionally enables communication protocols—previously defined by hardware—to be implemented in software. When protocols are realized in software, it becomes possible to customize network processing by updating or modifying this software. However, in existing SDN architectures, network processing software is often designed at a low level, similar to firmware, making runtime customization difficult. Changes to network processing through software modifications during operation—such as updating firmware—are typically not permitted. Moreover, the deployment of networking software at nodes is assumed to rely on mechanisms outside of SDN, limiting the flexibility of software distribution. Additionally, current SDN architectures assume software-based switching across all communication nodes, which prevents fine-grained customization at the level of individual sessions or packets.

This paper aims to propose a new software architecture that enables a more flexible SDN by introducing three key goals: First, allowing SDN customization during operation; Second, integrating the distribution of protocol-defining software as a native function of SDN; And third, enabling the finest granularity of customization, down to individual packets or nodes. To achieve the first goal, our architecture enhances protocol-level isolation to support the coexistence of multiple protocols, even within the same network layer, while minimizing the impact of swapping software components that implement them. For the second, it leverages SDN's inherent flexibility by embedding protocol distribution capabilities into the data communication process itself, enabling dynamic protocol changes. For the third, it supports fine-grained deployment of protocol-defining software to appropriate nodes, allowing coexistence with other protocols. However, predicting which nodes will require specific protocol software is not always feasible; therefore, we propose on-the-fly deployment of such software to relevant nodes during communication. To realize these goals, we propose a novel software architecture for SDN based on a cross-layer approach, as conventional SDN architectures are insufficient. For the first objective, we adapt the concept of the microkernel from operating systems to distinguish between dynamic and static software compo-

Satoh and I.
A Software Architecture for Highly Configurable Software-Defined Networks.
DOI: 10.5220/0013559500003964
In *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*, pages 319-326
ISBN: 978-989-758-757-3; ISSN: 2184-2833

nents, maximizing the former while minimizing the latter. Regarding the second goal, protocol-defining software is treated as a first-class object, enabling it to be transmitted between nodes just like regular data. For the third, protocol software can be distributed to participating nodes during ongoing communication. Our contribution to software engineering lies in introducing a dynamic and unified mechanism for managing software involved in network processing, thereby enabling highly flexible and diverse customization. To validate our proposal, we implemented a prototype of the architecture on the Java Virtual Machine (JVM) and developed several practical protocols for both data transmission and component deployment.

In this paper, we provide an overview of its structure. In Section 2, we delineate the fundamental concepts and design of our approach. In Section 3, we present the runtime systems employed in our implementation. Section 4 focuses on the deployable protocols used in our approach. Section 5 presents the initial experiences we encountered while developing and testing our approach. We commence with a discussion of related work in Section 6. Finally, in Section 7, we offer our conclusions based on our findings.

2 APPROACH

This paper proposes a software architecture that enables dynamic modification of applications and protocols. The architecture treats software components that define applications and communication protocols as first-class objects, facilitating software adaptability and a dynamic deployment mechanism. As a result, it allows the transfer of data and software without distinction. In real-world networks, the following requirements exist, and the architecture can deploy and execute software that defines applications and communication protocols while meeting these requirements:

- Resource Management: the architecture dynamically allocates and manages network resources such as bandwidth, processing power, and storage to meet the diverse requirements of different applications.
- Quality of Service (QoS): it should provide varying levels of service quality to different applications, ensuring that critical applications receive the necessary network resources to meet their requirements.
- Network Virtualization: it uses virtualization techniques to create multiple virtual networks that run different applications and provide different services, each with its own set of network resources and requirements.
- Application-Specific Network Functions: it should be able to deploy application-specific network functions, such as firewalls, load balancers, and traffic monitors, to provide custom network services to different applications.

This architecture supports the dynamic deployment of both application and networking software, and the deployment software can also be deployed in real-world networks.

2.1 Architecture

To satisfy the above requirements, the architecture introduces the following approaches: First, the difficulty in dynamically modifying network processing software during operation in existing SDN can be greatly mitigated by introducing the concept of a microkernel from operating systems, where processes related to communication are defined by software components operating outside the microkernel, thus significantly expanding the scope for customization. However, unlike operating systems, communication involves the concept of protocol layers, and our architecture differs from microkernel architectures in operating systems in the sense that our replaceable software components themselves are structured hierarchically. Nonetheless, since layering itself tends to rigidify the architecture, we enhance flexibility by realizing protocol layering through containment relationships among software components. These containment relationships can also be defined by the software components themselves.

Additionally, while existing SDN relies on non-SDN mechanisms when deploying software that defines communication protocols, our approach does not distinguish between data and software by treating the software defining communication protocols as first-class objects, like data. This approach allows for software distribution without differentiating it from data transmission, thereby enabling dynamic software deployment and modifications utilizing SDN capabilities. Regarding the issue that existing SDN cannot accommodate fine-grained customization at the level of communication sessions or packets, the proposed solution ensures that software defining relevant protocols can be distributed to nodes involved in communication while data is being transmitted.

2.2 Layered Protocols as Hierarchical Software Components

As most network protocols are organized in a hierarchy of layers, this architecture introduces network protocols arranged in a hierarchy of layers, where each layer provides an interface and extends services from the layer below. However, the structure of layering itself can sometimes hinder the flexible modification of protocols. Therefore, the architecture does not define layering explicitly; instead, protocol layering is realized through containment relationships among software components. The containment structure of components allows network protocols for components or data to be organized hierarchically. That is, each network protocol stack is implemented as a component hierarchy, and the deployment of a component within a component hierarchy is introduced as a basic mechanism for accessing services provided by the underlying layer. Our component-based protocols in the bottom layer are responsible for establishing point-to-point channels for transmitting data or components between neighboring computers. The middle layer is for routing protocols for transmitting data or components beyond directly connected nodes, and the architecture enables routing protocols for component migration to be performed by components.

2.3 Deployable Software Components as First-Class Objects

This architecture treats software components, including their containing components, as first-class objects, enabling their transfer between nodes without distinguishing them from data. Additionally, components are introduced as autonomous programs that can move between different computers under their own control. Similar to mobile agents, when a software component migrates to another computer, not only the component's code but also its state is transferred to the destination. However, unlike mobile agents, this architecture is characterized by two novel concepts: **component hierarchy** and **inter-component migration**. The former implies that one component can be contained within another component as shown in Fig. 1. That is, components are organized in a tree structure. The latter means that each component can migrate to other components as a whole, along with all its inner components, as long as the destination component accepts it. Components manage their inner components and provide their services and resources. Network protocols for component migration are also implemented within components.

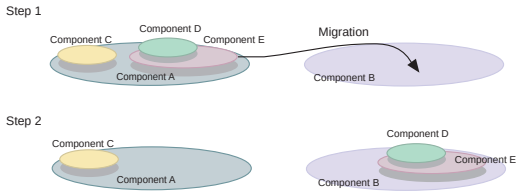

Figure 1: Hierarchical components and their migration.

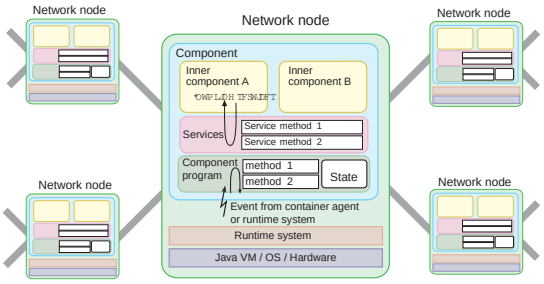

Figure 2: Structure of component.

3 COMPONENT-BASED PROTOCOLS FOR DEPLOYING COMPONENTS

In this section, we present the realization of the proposed architecture along with an implementation example of a protocol based on this architecture. The architecture consists not only of software components that define communication protocols, but also of a runtime system that manages the execution of these components. In the prototype implementation, both the software components and the runtime system operate on the Java Virtual Machine (JVM). As mentioned earlier, this study introduces the concept of a microkernel from operating systems in order to minimize the unchangeable parts of the software on each node and maximize the parts that can be modified. To achieve this, the runtime system defines only the minimal necessary functionality, and everything else is defined by components. Regarding communication, the prototype implementation uses TCP and UDP provided by the OS, but all other behaviors are defined and handled by components. To enable fine-grained and flexible protocol modifications, the system allows for preserving the program state (heap area) of a component even after replication or redeployment to another node. Although retaining execution state and resuming component operations from that state incurs overhead, it enables communication to be resumed at the new location, including communication settings and intermediate results.

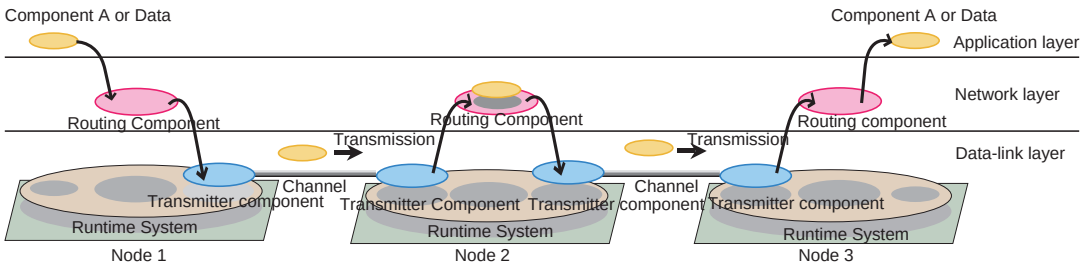

Figure 3: Architecture of component-based protocols for transmitting data or components.

3.1 Runtime System

The runtime system manages the execution of components—such as starting and stopping them—as well as the containment relationships among components. Additionally, the runtime system provides functionality for converting components into transferable data during redeployment, and for reconstructing them back into components afterward. Since components in the prototype are implemented as Java programs, Java's serialization features are used for this purpose. There is no distinction between software components and data, so components and their contained subcomponents can be transferred and executed on other nodes. The runtime system also maintains and manages the containment relationships among components. It allows one service to be provided by one or more components. Therefore, more than one different network protocol can be supported by service components that provide their own protocols for their inner components. Components can autonomously select an appropriate service component with suitable protocols to meet their requirements and reposition themselves into the selected components.

3.2 Deployable Protocols

Software components for defining protocols are capable of being repositioned and operational, which allows for extensive customization of network processing. Each component is capable of containing one or more components, with the outer components providing services to the inner components as shown in Fig. 3. Since software components and data are not distinguished, components are capable of transferring their contained components by serializing their program code and internal state, either within the same computer or to components on a different computer. Each component can be transmitted or duplicated to other nodes. Components are hierarchically organized as a protocol stack, where each component at a lower layer can be seen as a service provider for components at an upper layer. In addition to data, when one component is repositioned within another, the former component is treated as contained by the latter. The containing component can provide services such as transmission and transformation to its contained components. All components in this architecture are programmable entities. Protocols need to be defined as abstract classes in the Java language, but these basic protocols can be easily extended to define advanced protocols. This architecture demonstrates that the hierarchical model of communication protocols can be replicated.

Transmission for Point-to-Point Channels

A *transmitter* component is provided to dynamically deploy software components between neighboring nodes. The transmitter component is deployed on runtime systems at the source and destination nodes and establishes point-to-point channels for data and component transmission. This component can automatically exchange its internal components with the counterpart *transmitter* component at the current or remote node. Currently, the transmission of serialized components is assumed to utilize TCP for performance reasons, but a TCP-equivalent communication protocol can also be realized by components. To deploy a component, it is serialized with its state and code before being sent to the destination *transmitter* component. The receiving component reconstructs the component upon receipt. The *transmitter* component can be customized with security extensions such as an authentication mechanism and encryption. Dynamic deployment of *transmitter* components is also possible.

Routing Protocol for Component Transmission

Components can be deployed to one or more destinations under their own control, but determining the itinerary is challenging. Two approaches for determining and managing component itineraries are introduced. The first approach is based on packet routing mechanisms and is implemented as *forwarder* components that redirect components or data to new destinations. Each forwarder component holds a network structure table and redirects received components or

data to its destination or a closer forwarder component. This process is repeated until the component reaches its final destination. The second approach uses *navigator* components to convey the destinations of their inner components. The *navigator* component migrates itself with all its inner components to the next destination, propagates events to the inner components, and continues to the next destination. A security mechanism for *encrypted* component transmission is also provided.

Forwarding-Based Transmission: The forwarder mechanism is a packet routing approach commonly used in network systems. It consists of *forwarder* components that redirect data components to new destinations. *Forwarder* components are located at intermediate nodes in a network and store a table that describes the network structure. Upon receiving a data component, the *forwarder* checks its destination, and if it's not the final destination, it redirects it to the final destination or to another forwarder closer to the destination as shown in Fig. 4. *Forwarder* components can propagate certain events to visiting components, instructing them to perform actions before forwarding them to their final destinations. The process is repeated until the data component reaches its final destination.

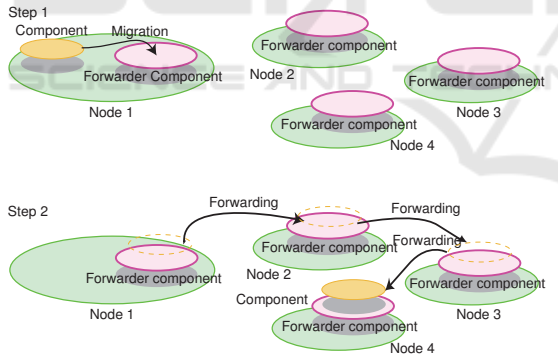

Figure 4: Routing components for forwarding to the next hosts.

Navigating-Based Transmission: Navigators are service components that convey the destinations of inner components to these components, as shown in Fig. 5. The *navigator* components can move themselves and their inner components to the next location, determined statically, algorithmically, or based on the inner components' previous computations and the environment. The *navigator* components can propagate events to their inner components upon arrival, then continue their itinerary. Security mechanisms are implemented in the navigators, which can *encrypt/decrypt* components using their own encryption/decryption mechanisms.

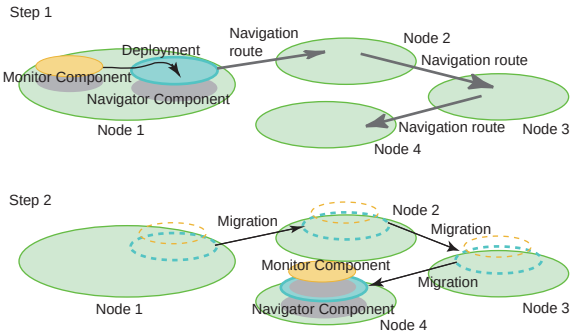

Figure 5: Navigator component with its inner components for traveling among nodes.

3.3 Component-Based Protocol Distribution for Deploying Components

Service components that support protocols are deployed in the system using one of three mechanisms: autonomous migration to nodes where the protocols are needed, passive deployment with components for the protocols installed at nodes, or active deployment through a designated component that selects and installs the required protocols at nodes. Service components for defining protocols autonomously migrate to nodes at which the protocols may be needed and remain at the nodes in a decentralized manner. Components for protocols are passively deployed at nodes that may require them by using forwarder components prior to utilizing the protocols as distributors of protocols. Moving components can carry service components for defining protocols inside themselves and deploy the components at nodes that the component traverses.

4 EARLY EXPERIMENT

The current implementation runs on Java VM ver. 17 or later and employs the common Java object serialization package for marshaling and unmarshaling the states of components. The mechanism for marshaling and unmarshaling components is similar to that used for marshaling and unmarshaling mobile agents (Satoh, 2010). Transmitter components establish communication channels through TCP, HTTP, and SMTP, while forwarder and navigator components traverse multiple nodes based on their static routing tables and SNMP agents.

The current implementation of the architecture was not optimized for performance. However, its basic performance was evaluated on a distributed system consisting of 8 PCs (Intel Core i5 2.4GHz) connected by 1Gbps Ethernet. The per-hop latency (in microseconds) and the throughput (in components per second) were measured. The target components were minimal-sized components consisting of common callback methods invoked during changes in their life-cycle state. The size of each component was approximately 4 Kbytes (zip-compressed). The time for migrating the component within a hierarchy and between two nodes was also measured for reference. Component migration within a component hierarchy took less than 1 millisecond, including the time for checking entry permission. Migration between nodes was performed using simple TCP-based transmitter components and had a per-hop latency of 10 milliseconds and a throughput of 75 components per second. The latency included marshaling, compression, establishing a TCP connection, transmission, acknowledgment, decompression, security verification, and creating Java namespace components. The forwarder protocol had a per-hop latency of 13 milliseconds and a throughput of 60 components per second, while the navigator protocol had a per-hop latency of 11 milliseconds and a throughput of 62 components per second. The forwarder component determined the next hop based on its routing table, while the navigator migrated itself and its inner components to the next node by incorporating itself into a transmitter component.

The above results, obtained without performance optimization, are not conclusive. The forwarder protocol performed better than the navigator protocol because the latter also migrated the protocol itself. When multiple components were migrated onto a network, congestion on each computer was sometimes unbalanced due to asynchronous execution. The overhead of the component-based protocols in terms of latency was significant compared to high-speed communication protocols. However, the protocols were sufficient for high-level prototypes of application-specific protocols, such as data transmission in sensor networks and monitoring of sensor nodes. The throughput was limited by the security mechanisms of the Java VM and runtime system, not by the protocols.

4.1 Application

We present an application of the proposed architecture here. The application is employed to gather data from each sensor node in a sensor network and to process the collected data using application-specific data processing programs. The sensor network comprises one or a small number of intermediate nodes and a large number of sensor nodes. The intermediate node collects data from the sensor nodes, processes it, and then forwards it to the cloud. To minimize data communication, edge computing is utilized to perform some data processing at the sensor nodes.

4.2 Data Collection

Two approaches exist for collecting data in the sensor network. The first approach entails sending a request message from the intermediate node to each sensor node to collect measurement data. In this approach, the intermediate node collects data from numerous sensor nodes by sending a request message and receiving reply messages containing measurement data from the sensor nodes. The second approach involves sending a message from the intermediate node to one sensor node to query its measurement data. Each sensor node subsequently sends messages containing its measurement data to its neighboring nodes and ultimately returns the message to the intermediate node with the results of multiple sensor node measurements.

Using Forwarder Components

The first approach collects data at sensor nodes using forwarder components through transmitter components in the intermediate node and the sensor nodes. The forwarder component receives a data processing component, creates duplicates of it, and transmits the duplicated components to the target sensor nodes for data processing at the nodes. After each sensor node receives a data processing component, it instructs the component to execute data processing programs and then returns the component with its processing results to the intermediate node. The forwarder component in the intermediate node aggregates the data or performs further processing.

Using Navigator Components

The second approach collects data at sensor nodes using navigator components between sensor nodes or between a sensor node and an intermediate node employing a transmitter component. The navigator component moves with the data processing component according to its own path and executes the data processing program at the arriving sensor node. After its execution, the navigator component forwards the data processing component with its results to the next sensor node and finally back to the intermediate node. This approach requires a larger amount of data

to be transferred, but with less frequent transfers, as the navigator component is transferred along with the data processing component.

Remarks

We compare the first and second approaches. The first approach necessitates a larger amount of data to be transferred but with less frequent transfers, as the navigator component is transferred between nodes along with the data processing component. The second approach can reduce the number of data communication instances and enable efficient data processing and collection.

5 RELATED WORK

The domain of dynamic deployment and configuration of software in networks has seen the implementation of various techniques for application software. However, the utilization of dynamic relocation for network processing software is not as prevalent. In this section, we will analyze existing literature with a specific emphasis on Software-Defined Networking (SDN) and active networking technologies.

One reason SDN gained attention is that many existing network devices had to be configured by directly operating the device itself, making them unsuitable for remote monitoring and operation. Another factor is the improvement in CPU (i.e., general-purpose processors) performance, which made it possible to handle network processing not with dedicated semiconductors (ASICs) but instead with general-purpose processors and software. Initially, most SDN research focused on remote network configuration under early frameworks such as OpenFlow (McKeown et al., 2008a).

SDN adopts software-based controllers or APIs to communicate with the underlying hardware infrastructure. Dynamic reconfiguration and deployment of networking software are crucial in SDN, and there are practical implementations, such as OpenFlow (McKeown et al., 2008b) and NETCONF (Enns et al., 2006). The OpenFlow protocol separates the control and data forwarding planes in software-defined networking (SDN) architecture. The control plane acts as a controller to manage network infrastructure and implement custom policies, while the data forwarding plane handles hardware forwarding. Communication between the two requires specific protocols. NETCONF is a management protocol for modifying network device configurations, but SDN reconfiguration can be limited. Active networking, which is programmable for custom services, has not been widely adopted due to security and performance issues. They are too heavy to support networks in the real world.

Although SDN allows for customizing network processing by distributing software that defines the network behavior to each node, in practice, software distribution is generally handled using non-SDN methods such as standard software deployment and management tools (e.g., Chef or Ansible). ONAP (Open Network Automation Platform) and OPNFV (Open Platform for NFV) enable comprehensive orchestration of NFV (Network Functions Virtualization) environments (Kirksey et al., 2017; The Linux Foundation, 2017). While the SDN controller can determine which nodes receive software, the actual distribution itself uses mechanisms other than SDN.

On the other hand, distributing the very software that constitutes SDN through SDN does enable centralized management, but there have been few attempts in this area. One example is P4Runtime. P4Runtime is a protocol that transfers data-plane behavior—such as packet processing logic described in the P4 language—from the controller to switches for execution (Bosshart et al., 2014; Community,). With P4Runtime, it is even possible to temporarily run the active pipeline in parallel with a new version of the pipeline (Jin et al., 2020). P4Runtime assumes that the communication nodes can execute programs written in P4, but the method proposed in this paper uses the Java Virtual Machine (JVM). If the code can run on the JVM, it does not matter which programming language is used. Also, while P4Runtime can only change the communication settings on a per-pipeline basis, our proposed method has the advantage that it can be changed on a per-packet basis.

The present study proposes a framework for programmable networks, drawing upon the concept of active networking (Tennenhouse et al., 1997). Despite being explored by several researchers, active networking, characterized by its programmability for custom services, has not gained widespread adoption due to security and performance challenges. The paper presents an architecture for programmable networks based on active networking, which has faced challenges in terms of security and performance. The framework utilizes mobile agent technologies and aims to use software-defined networking for edge computing and sensor networking. It overcomes limitations faced by previous proposals for improved connectivity by keeping components related to network processing small (less than 1 MB), enabling efficient adaptation of software networking.

Our architecture introduces the concept of hierarchical software components, a unique approach to

configuring SDNs in addition to real-world networks, which has rarely been explored. The authors highlight that there have been only a few attempts to incorporate hierarchical components in SDNs and edge computing configurations, with one example being a policy description proposed by Ferguson et al. to simplify large SDN networks through sub-policy construction (Ferguson et al., 2012). Finally, it is noteworthy that the proposed architecture adopts mobile agent technologies, instead of code migration techniques, to facilitate programmable networks. Existing works have applied mobile agents in active networks (Busse et al., 1999); however, these did not support hierarchical components, unlike the proposed architecture.

6 CONCLUSION

In this paper, we proposed a new software architecture for SDN, which makes several significant contributions. By incorporating the concept of a microkernel from operating systems, we differentiated between static and changeable software within the SDN architecture, minimizing the former and maximizing the latter to facilitate dynamic adaptations. By treating software that defines communication protocols as first-class objects, akin to data, we enabled seamless software delivery without distinguishing it from data communication, thus leveraging SDN for dynamic software distribution and modification. Additionally, we ensured that software defining protocols could be distributed to the nodes involved in data transmission during communication. These capabilities were preliminarily evaluated through its prototype implementation. Some readers may consider the current architecture to have low communication performance. However, in network infrastructures where flexible SDN is required, such as sensor networks, the priority often lies in flexible customization rather than high communication performance, and the lower performance is not typically a problem.

Lastly, future challenges are discussed. Although we have evaluated the performance of the system on a small scale, we intend to assess the performance of the system on a larger scale and in more realistic settings. The protocols implemented in this architecture are still few. We want to verify its versatility by implementing protocols tailored for specific networks, such as mobile ad-hoc networks, as well as application-level protocols such as service discovery and publish-subscribe models.

REFERENCES

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vannaccione, A., Walker, D., and Wotherspoon, L. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.

Busse, I., Covaci, S., and Leichsenring, A. (1999). Autonomy and Decentralization in Active Networks: A Case Study for Mobile Agents. In *Proceedings of Working Conference on Active Networks*, volume 1653 of *LNCS*, pages 165–179. Springer.

Community, P. P4runtime specification. https://p4.org/p4runtime/. Accessed: 2025-03-03.

Enns, R. et al. (2006). IETF NETCONF Network Configuration protocol. Technical report, RFC 4741 (Proposed Standard). Obsoleted by RFC 6241.

Ferguson, A. D., Guha, A., Liang, C., Fonseca, R., and Krishnamurthi, S. (2012). Hierarchical policies for software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks (HotSDN '12)*, pages 37–42. ACM Press.

Jin, X., Gao, J., Liu, A. A., Dong, Z., Xie, G., and Zhang, M. (2020). Dynamic control of programmable data planes with P4Runtime. *IEEE Transactions on Network and Service Management*, 17(4):2385–2398.

Kirksey, H. et al. (2017). OPNFV: An open platform to accelerate NFV. *IEEE Communications Standards Magazine*, 1(4):72–78.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008a). OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008b). Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.

Satoh, I. (2010). Mobile Agents. In *Handbook of Ambient Intelligence and Smart Environments*, pages 771–791. Springer.

Tennenhouse, D. L. et al. (1997). A Survey of Active Network Research. *IEEE Communication Magazine*, 35(1).

The Linux Foundation (2017). Open Network Automation Platform (ONAP): Overview and architecture white paper. https://www.onap.org/. Accessed: 2025-03-03.