Self-Defined Protocols for Ubiquitous Networks

Ichiro Satoh
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
E-mail: ichiro@nii.ac.jp

***Abstract*—The paper presents a novel approach to the dynamic deployment of both networking and application software in ubiquitous networks. The proposed approach introduces the concept of first-class objects, enabling the dynamic construction and configuration of software components. This leads to the capability of dynamically deploying software components for defining both networking and application functions at nodes by utilizing network protocols specified within software components. The dynamic deployment improves the reliability and resilience of the network by actively managing network resources, configuring network components, and monitoring network conditions. Furthermore, the approach offers the added advantage of allowing the use of non-standard protocols, as components for defining protocols can be dynamically deployed at nodes using other component-based protocols.**

***Index Terms*—Software deployment, configurable protocol, first-class object**

I. Introduction

Ubiquitous networks serve multiple purposes and exhibit inherent dynamism, as they simultaneously accommodate various applications and frequently undergo topology changes when nodes are added or removed. User demand for network services in ubiquitous networks can shift rapidly, necessitating that the network dynamically adapt to new requirements and quality of service levels. To support multiple applications, ubiquitous networks must tailor themselves to the unique requirements of individual applications. Such networks need to be made more reliable and resilient through dynamic configuration and deployment of software that can automatically detect and recover from failures or errors. They must also enhance their performance through dynamic reconfiguration and optimization of network resources.

Therefore, ubiquitous networks require the capacity to dynamically allocate and manage their network resources, such as bandwidth, processing power, and storage, to accommodate changing network conditions and requirements. They should be capable of dynamically configuring and reconfiguring network components, including routers, switches, and servers, to boost network efficiency and performance. Modern ubiquitous networks tend to be software-defined, as many of their functions are determined and operated at the software level rather than the hardware level. They necessitate the ability to dynamically deploy and manage software components for defining network protocols and functions, in order to bolster network reliability and resilience. Furthermore, they should be able to monitor network conditions, detect faults or errors, and allow the network to dynamically recover from failures or enhance performance.

In this paper, we present a novel approach to the dynamic deployment of both networking and application software without differentiation between the two. The approach capitalizes on the concept of first-class objects in programming languages, which are entities that can be dynamically constructed, passed as parameters, returned from functions, and assigned to variables [6]. The proposed approach encompasses not only the dynamic deployment of networking and application software but also the dynamic deployment of relocation software. The proposed solution comprises software components for defining both applications and networking, which can process or transmit other components.

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

II. 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.

SDN adopts software-based controllers or APIs to communicate with the underlying hardware infrastructure. Dynamic reconfiguration and deployment of networking software is crucial in SDN, and there are practical implementations, such as OpenFlow [10] and NETCONF [9]. 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 ubiquitous and heterogenous networks.

The present study proposes a framework for programmable networks, drawing upon the concept of active networking [14]. 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 a framework 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.

The proposed framework for programmable networks introduces the use of hierarchical software components, a unique approach to configuring SDNs in addition to ubiquitous 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 SDNs networks through sub-policy construction [5]. Finally, it is noteworthy that the proposed framework adopts mobile agent technologies, instead of code migration techniques, to facilitate programmable networks. Previous studies have applied mobile agents in active networks [3]; however, these did not support hierarchical components, unlike the proposed framework.

III. Approach

The framework considers software components as important objects, allowing adaptability of software and dynamic deployment mechanism. To support multiple applications and dynamic environments and requirements, ubiquitous networks support the following functions:

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

The framework supports dynamic deployment of both application and networking software and the deployment software can also be deployed in ubiquitous networks.

A. Software Components as First-Class Objects

This framework treats software components as first-class objects, allowing them to move and transfer code and state between computers. Also, software components are autonomous programs that can travel between different computers under their own control. Like mobile agents, when a software component migrates to another computer, not only the code of the component but also its state can be transferred to the destination. Unlike mobile agents, the framework is characterized by two novel concepts: **component hierarchy** and **inter-component migration**. The former means that one component can be contained within another component. That is, components are organized in a tree structure. The latter means that each component can migrate to other components as a whole, with all its inner components, as long as the destination component accepts it. A container component manages its inner components and offers its services and resources. Network protocols for component migration are implemented within components.

B. Layered Protocols for Deploying Software Components

Since most network protocols are arranged in a hierarchy of layers, this framework introduces network protocols arranged in a hierarchy of layers, where each layer provides an interface and extends services from the layer below. This framework introduces the notion of higher-order functions in the sense that a component can contained by the most one component and a container component is responsible for processing its containing components. The containment structure of components enables network protocols for components or data to be organized hierarchically. That is, each a network protocol stack is implemented as a component hierarchy as shown in Fig. 2, and the deployment of a component in 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 correspond to data-link layered protocols. They are responsible for establishing point-to-point channels for transmitting data or deploying components between neighboring computers. The middle layer is for routing protocols for transmitting data or components beyond directly connected nodes, and the framework enables routing protocols for component migration to be performed by components.

IV. Runtime System

This section presents runtime systems for component protocol definition. Our runtime systems operate on the Java Virtual Machine, utilizing Java objects as components. Adopting a micro-kernel architecture, it consists of two parts: a minimal runtime system offering common, environment-independent functions, and higher-level components providing additional functionality such as component migration over a network, dependent on the environment.

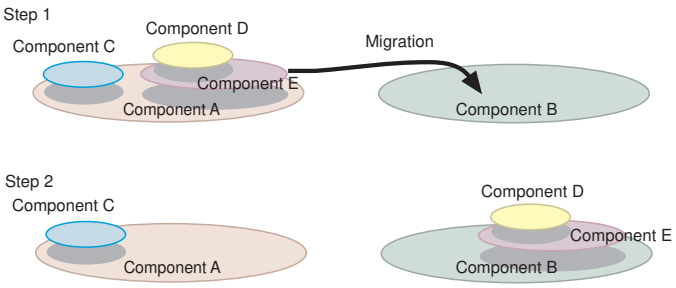

Fig. 1. Hierarchical structure of components and their migration.

Component execution management: The runtime system manages the active threads of each component through the Java thread mechanism. The life-cycle state of components running in the system is monitored. Changes to the life-cycle state, such as creation, termination, or migration, result in core system events that trigger designated methods within the component and its parent components. For instance, upon deployment of a component and its subcomponents to a different node, the runtime system initiates callback methods to halt active threads prior to transferring. Upon arrival at the destination, the runtime system at the destination assigns new active threads to the transferred component and subcomponents, then invokes corresponding callback methods to resume activities with the new active threads.

Component hierarchy management: Each runtime system serves as the root node in a component hierarchy, represented as a tree structure with each node comprising a component and its attributes. Component migration within the hierarchy is accomplished via structural transformations of the tree. Upon component creation, the runtime system assigns a unique identifier. To designate components in a hierarchy, the system utilizes URL-based notation consisting of a path notation constructed from component identifiers separated by slashes, reflecting the hierarchical arrangement, combined with the network address and port number assigned to the runtime system at the node.

Serialization and security management: The runtime system features component marshaling (serialization) and unmarshaling capabilities. The current implementation utilizes Java object serialization for component state marshaling and operates based on weak mobility principles [7]. The runtime system uses Java security to verify the validity of marshaled components and supports duplication of code, state, and subcomponents of a component.

A. Component

A component consists of three parts: body program, context objects, and inner components (as depicted in Fig. 3), which are represented as Java objects. These parts are given as Java objects.[1] The body program is an instance of a subclass of the abstract Java class `Component`. This class defines crucial callback methods that are invoked during component life-cycle changes, such as creation, suspension, marshaling, unmarshaling, destruction, etc. It also provides a migration command, `go(Component URL destination)`, for component migration within a component hierarchy. The migration of a component to a specified destination component occurs when the component executes this command. Inner components do not have access to methods defined in the container component. Instead, the container component offers service methods through a context object, a subclass of the `Context` class, that can be indirectly accessed by the inner components to obtain information about and interact with the environment, including the container component, sibling components, and the underlying computer system.

[1]The framework itself is independent of any programming languages, but the Java language is machine-neutral, and provides a stable serialization mechanism and rich libraries. Therefore, the current implementation assumes that all components are defined in the Java language.

V. Component-based Protocols for Deploying Components

In this framework, network processing for not only data but also components can be implemented as software components, which are designed to be run on the core system. Each component can provide its own service to its inner components. We call the former *service* component.

- Components are hierarchically organized as a protocol stack in the sense that each component in a lower layer can be viewed as a service provider for components in an upper layer. The movement of a component to a service component corresponds to the process of applying the network service of the service component to the moving component.
- Each runtime system permits one service to be provided by one or more components. Therefore, more than one different network protocol can be supported by the service components that provide their own protocols for their inner components. Components can dynamically select a suitable service component whose protocols can satisfy their requirements and migrate them into the selected components.
- Since service components for performing protocols are still mobile, the protocols can be dynamically deployed at nodes by migrating the components to the nodes. Service components and other components can be migrated to other components at the current or remote nodes in a unified manner.
- When each service component migrates to another service component at the current or different node, it can carry its inner components.

All components in this framework can be programmable entities. Although protocols need to be defined as abstract classes in the Java language, we can easily define advanced protocols by extending these basic protocols.

A. Transmission for Point-to-Point Channels

A *transmitter* component is provided to deploy software components dynamically between neighboring nodes. The transmitter component is deployed on runtime systems at the

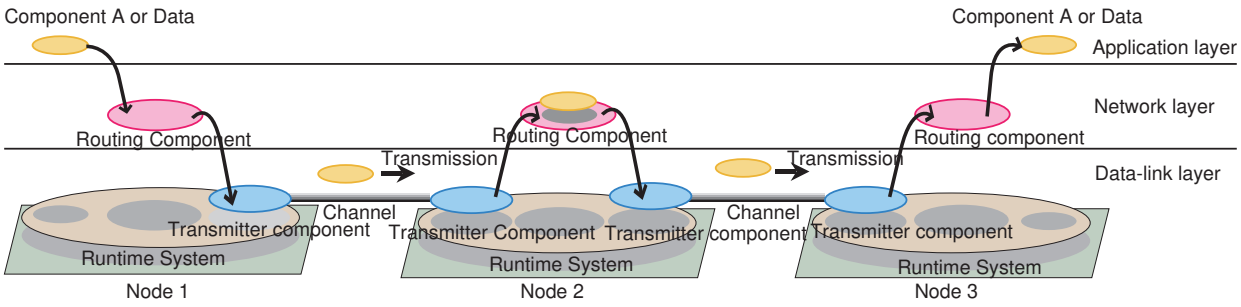

Fig. 2. Architecture of component-based protocols for transmitting data or components.

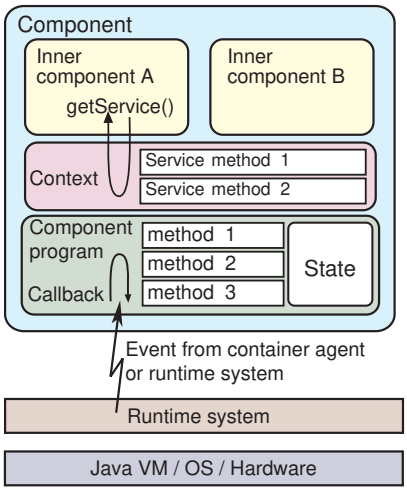

Fig. 3. Structure of component

source and destination nodes and establishes point-to-point channels for data and component transmission. The component can automatically exchange its internal components with the counterpart *transmitter* component at the current or remote node. Data transmission is achieved through data communication protocols such as TCP and UDP. To deploy a component, it is transmitted through the transmitter component, which serializes its state and code before sending it to the destination *transmitter* component. The destination 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.

B. Routing Protocol for Component Transmission

Components can travel to one or more destinations with their own control, but determining the itinerary is difficult. Two approaches for determining and managing component itinerary are introduced. The first approach is based on packet routing mechanisms and 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. The process is repeated until the component arrives at 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. The *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.

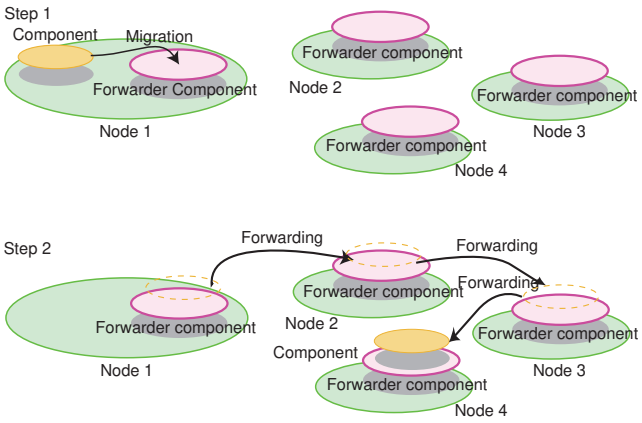

Fig. 4. Routing components for forwarding to the next hosts.

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.

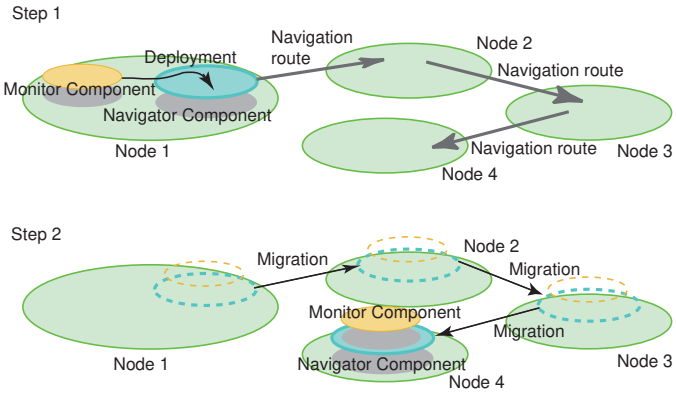

Fig. 5. Navigator component with its inner components for traveling among nodes.

C. 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.

VI. 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 of marshaling and unmarshaling mobile agents [11]. 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 framework 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 mechanism of the Java VM and runtime system, not by the protocols.

A. Application

We present an application of the proposed framework 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.

B. 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 node. 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 second method 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.

VII. Conclusion

This paper presents a framework for constructing software that enables the definition of networking in ubiquitous networks. The framework is characterized by the incorporation of first-class objects, eliminating the distinction between data and components for network definition. This facilitates the unified creation and management of ubiquitous networks through a single programming model. Software components for ubiquitous networks can define protocols for transmitting themselves or other components and data, and can be dynamically deployed or migrated to network hosts by other components. This component-based approach allows for the use of non-standard protocols, as the protocols can be dynamically deployed at source and destination nodes. The paper details a prototype implementation built on Java VM, including several practical protocols for deploying components and data transfer, and evaluates the basic performance of the implementation. It is worth noting that, compared to the micro-services method, the cost of dynamic deployment is significantly reduced due to the small software unit of this proposed method.

Lastly, future challenges are discussed. This paper demonstrates that basic communication protocols such as point-to-point channels and routing can be implemented within this framework; however, we plan to implement advanced or evolving protocols. Although we have evaluated the performance of the system on a small scale, we intend to assess the performance of the system on larger scales and in more realistic settings.

References

[1] J. Beal, M. Viroli, D. Pianini, and F. Damiani: "Self- Adaptation to Device Distribution in the Internet of Things," ACM Transactions on Autonomous and Adaptive Systems (TAAS), Vol.12, No.3, pp.1–12, ACM, 2017.

[2] A. Bieszczad, B. Pagurek, and T. White, "Mobile Agents for Network Management", IEEE Communications Surveys, Vol. 1, No. 1, Fourth Quarter 1998.

[3] I. Busse, S. Covaci, and A. Leichsenring: "Autonomy and Decentralization in Active Networks: A Case Study for Mobile Agents," Proceedings of Working Conference on Active Networks, pp.165–179, LNCS, Vol.1653, Springer, 1999.

[4] K.L. Calvert, W.K. Edwards, N. Feamster, R.E. Grinter, Y. Deng, and X. Zhou: "Instrumenting home networks," ACM SIGCOMM Computer Communication Review, Vol.41, No.1, pp.84–89, ACM, 2011.

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

[6] D. P. Friedman, M. Wand, and C. T. Haynes: "Essentials of Programming Languages", MIT Press, 1992.

[7] A. Fuggetta, G. P. Picco, and G. Vigna: Understanding Code Mobility, IEEE Transactions on Software Engineering, vol.24, no.5, 1998.

[8] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles: "Plan: A packet language for active networks," In Proc. Functional Programming (ICFP), 1998.

[9] R. Enns, et al.: "IETF NETCONF Network Configuration protocol," RFC 4741 (Proposed Standard), December 2006. Obsoleted by RFC 6241.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner: "Openflow: enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, Vol.38, No.2, pp.69–74, ACM, 2008.

[11] I. Satoh: "Mobile Agents," Handbook of Ambient Intelligence and Smart Environments, pp.771-791, Springer, 2010.

[12] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge: "Smart packets: Applying active networks to network management," ACM Tran. on Computer Systems, 2000.

[13] S. Y. Shin, S. Nejati, M. Sabetzadeh, L. C. Briand, C. Arora, and F. Zimmer: "Dynamic adaptation of software-defined networks for IoT systems: a search-based approach," In Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp.137–148, 2020.

[14] D. L. Tennenhouse et al.: "A Survey of Active Network Research", IEEE Communication Magazine, vol. 35, no. 1, 1997.

[15] D. Weyns, M. Usman, D. Hughes and N. Matthys: "Applying Architecture-Based Adaptation to Automate the Management of Internet-of-Things," In Proceedings of the 12th European Conference on Software Architecture (ECSA'18), pp.49–67, LNCS 11048, Springer, 2018.