A Thesis for the Degree of Ph.D. in Engineering

Smart Community Edge Platform Providing Stream Content Analysis, Service Migration, and Service Chaining

February 2021

Graduate School of Science and Technology
Keio University

Wickramaarachchi A. Shanaka P. Abeysiriwardhana

Abstract

A smart community utilizes information technology to interconnect and manage community infrastructures. These networks consist of many Internet of Things(IoT) devices that provide different services to the end-users. In conventional networks, these sensor data send to cloud services for processing and management. However, cloud-based data processing introduces latency to the services. Fog computing techniques have been introduced to support these services at the network edge reducing the network latency. Smart community networks should support latency-sensitive services such as smart grid systems at the edge. In addition, Smart community services require service migration and service chaining to manage and distribute multiple services. For example, the current smart community edge(SCE) supports smart energy management services where data anonymization and data aggregation services should be chained, and the services should be migrated depending on the network's location and network traffic.

SCE services can leverage generic hardware devices and network virtualization technologies to deploy the services without proprietary middleware devices. The current network virtualization methods mainly consider only core network applications. In contrast, smart community services operate on application layer data and process in-transit data to capture sensor data at the edge. Therefore, data extraction edge nodes that support sensor data processing are required to support these smart community services. A service-oriented container-based solution that processes data streams from sensors using conventional hardware will improve the applicability, compatibility, and latency of smart community services.

To this end, a software-based edge node, namely, the SCE platform, was proposed to support smart community services. SCE supports data-tapping applications, especially for IoT devices, and has a stream processing feature with a comparatively shorter processing delay. This tapping and processing function on in-transit data was named stream content analysis(SCA). SCA captures in-transit data through zero copy stream reconstruction and string matching process. Afterward, SCE proposes a distributed rule application method to manage multiple services and distribute matched data to the services. SCE supports services through Docker containers to provide remote deployment, service migration, and service isolation. The real world SCE platform implementation allows SCE services to operate on 10Gbps links and apply 100 accumulated rules while maintaining less than 1ms latency using commodity hardware devices.

To support SCE service migration, SCE proposes a consistently guaranteed migration method to support service migration to distribute the services depending on the nodes' availability. The proposed migration technique is designed to guarantee network consistency while migrating between nodes.

Compared to existing container migration methods, the proposed migration reduces the migration data transfer through container layers and migrating only the streams affected by the migration application through SCA. The proposed container migration methods reduced the network downtime by more than 10% compared to conventional methods for containers with image sizes larger than 400MBs. Furthermore, SCE services require chaining to distribute sensor data efficiently to the edge nodes to apply multiple network services for a given traffic flow. To this end, SCE introduces a service function chaining-based request distribution method that utilizes proactive data collection and heuristics to analyze the network traffic and to select optimal SCE nodes. The SCE request distribution method reduces the end-to-end service latency by 10% compared to the available algorithms. The SCE platform provides commodity hardware-based SCA, distributed rule change application, service migration, and service chaining to support SCE services.

Acknowledgments

At various stages throughout the Ph.D. process, I had interesting discussions, support, and valuable feedback from my supervisor Prof. Hiroaki Nishi. I would like to express my sincere gratitude to him for the valuable guidance and support throughout these years.

I would like to give my heartfelt gratitude to Mrs. Yuko Nishi for the endless support during this study. Moreover, I would like to acknowledge all West Laboratory members, especially Janaka Wijekoon, Rajitha Tennakoon, Yuchi Nakamura, Tatsuki Miura, and Ryo Morishima, for their generous support and help during the time I spent in West Laboratory. I want to acknowledge my friend Kasun Prasanga and Maheshi Ruwanthika for the generous support.

A very special thanks go to my dearest parents and my dearest sister and brother in law, for the courage and strength you guys gave me makes the man who I am today.

Finally, I acknowledge the following people and institutions for the precious contribution to this thesis's success.

- For the valuable comments, guidelines, and, above all, advice to make the dissertation a success.

- MEXT (Ministry of Education, Culture, Sports, Science, and Technology), for financial support throughout the period at Graduate School of Science and Technology, Keio University, Japan.

- KLL research grant and Keio University Doctorate Student Grant-in-Aid program for support during the thesis study.

Wickramaarachchi A. Shanaka P. Abeysiriwardhana

January, 2021

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivation

Information technology integrates communication, processing, and computing technologies to provide services such as healthcare, education. UNIVAC 1, the first commercial computer [1], was developed by John Eckert and John W. Mauchly in 1951. For the next few decades, enterprises developed mainframe computers to store and process a large amount of information. The internet was started in the 1960s as a data transfer solution between large-sized immobile computers[2]. The size of computers reduced around the 1980s to the size of minicomputers. The ARPANET and Defense Data Network officially standardized internet communication with TCP/IP protocol in 1983, and this day is considered the birthday of The internet[2]. The connected devices on the internet have become smaller and smaller to the current trend of IoT, Big data, and Smart cities.

Processing, computing, and communication of a large number of connected devices required considerable processing power. Before the internet, mainframe computers were initially used in enterprises to compute information in the 1950s[2]. Internet services required datacenters to process the information on connected devices. The term cloud computing was introduced around 2006 to identify the new paradigm in which people increasingly access software, computer power, and files over the web instead of on their desktops[3]. In the cloud computing paradigm, user traffic usually sends to data centers for processing. Commonly, the Representational state transfer Application Programming Interface[4] (Rest API) is used to communicate where users request data through a well-defined interface. The client-server architecture was used by these services to execute end-user requests in the cloud. Cloud computing also uses virtualization technologies to deploy the services on bare metal servers. The data processing services are usually packaged to virtual machines and executed within the data center. The virtualization technologies improve the scalability, efficiency, and agility of these networks.

Cloud services usually process application layer data of the communication. However, network appliances process network-layer, transport layer, and data-link layer data using services such as a firewall. These network appliances are also moving from proprietary hardware devices to virtualized devices with the development of virtualization technologies. ETSI introduced network function virtualization(NFV) in 2012 in a conference on software-defined networking and OpenFlow[5]. NFV provides a framework to manage virtualized network services. The framework includes virtualized network functions(VNFs), network function

virtualization infrastructure(NFVI), and network function virtualization management and orchestration(MANO). These three components allow service providers to deploy, manage, and orchestrate network services. However, NFV is designed toward network-layer applications, while cloud computing is designed toward application layer services.

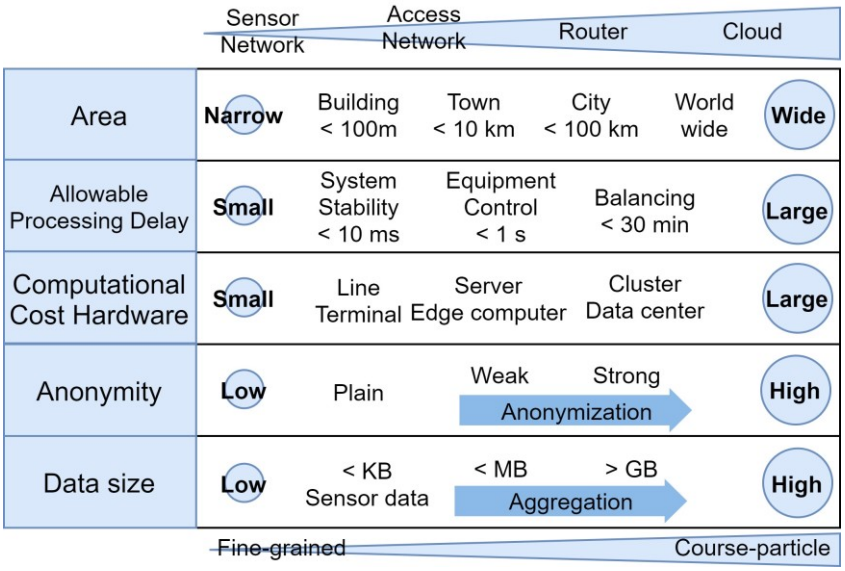


Figure 1-1 Application service hierarchy

Edge and fog computing was introduced as an added computing layer between the cloud and terminal devices[9]. Fog computing is a paradigm of distributed computing. In contrast, cloud computing tends to be more centralized. Centralized systems are easier to manage and deploy while they introduce longer access time to users. On the other hand, a distributed system provides resilience, better performance, and flexibility while causing high deployment and maintenance costs due to the system's geographical distribution. Therefore, fog computing has the decentralized system's advantages, while edge nodes' deployment and management become problematic in networks. Cloud computing has more extensive computation capability compared to edge computing nodes. However, cloud computing data centers are placed further away from the end-users. Fog computing is about processing real-time data closer to the network edge, while cloud computing runs end-user applications, as shown in Figure 1-1. Therefore, the fog layer should capture in-transit IoT data to provide real-time smart community services. A fog computing platform should be able to capture in transit IoT data from the line. In contrast, cloud services usually process the end request; therefore, it could operate in a standard client-server architecture. In contrast, fog computing nodes carry out stream processing to process in-transit information.

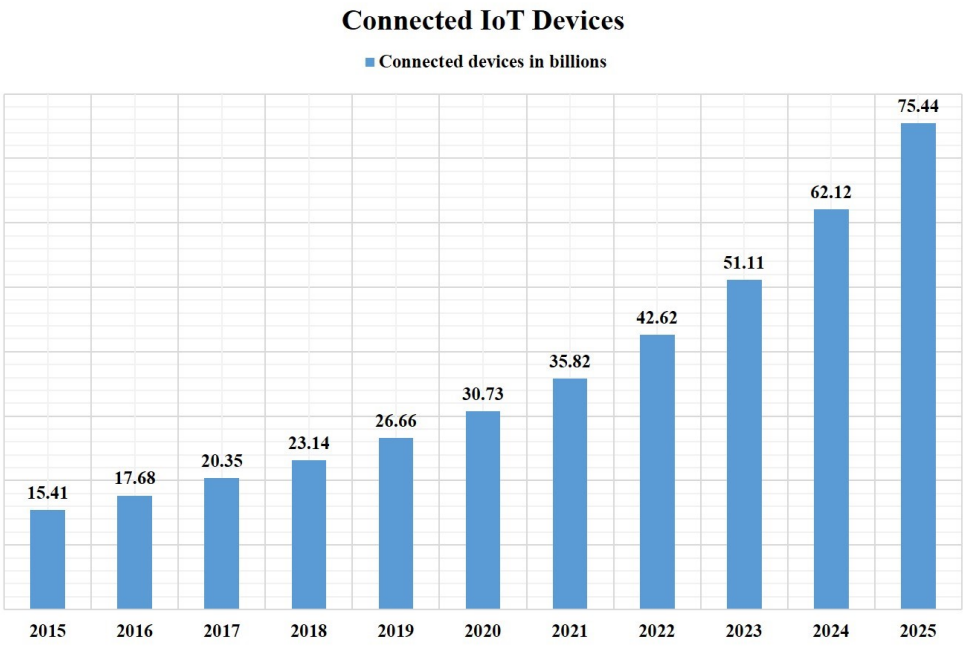


Figure 1-2 Connected IoT devices

Fog and edge-based services could be used to process the information of a rapidly increasing number of connected IoT devices[6], as shown in Figure 1-2. According to current trends, this will increase to 75 billion devices in 2025[6]. However, these connected devices allow us to develop complex systems to monitor and manage our community. However, with the increase of IoT sensors, it is required to process these data closer to the network edge to reduce network latency and traffic[7]. The smart community concept uses these devices to efficiently manage the citizens' infrastructure, ICT, energy, and lifestyle. According to the Japan smart community alliance, "smart community is a community where various next-generation technologies and advanced social systems are effectively integrated and utilized, including the efficient use of energy, utilization of heat and unused energy sources, improvement of local transportation systems and transformation of the everyday lives of citizens" [8]. This requires smart community services and users to process data efficiently. However, the application services and the computing location depend on the type of capabilities required by the different services, as shown in Figure 1-1. For example, a sensor network operates in a narrow area with limited computation and security capabilities. In contrast, cloud services operate in a broader area with unlimited computation capability. The fog layer operates in the middle to provide services such as anonymization for these weak terminal devices before the traffic transit into the cloud. Smart community services could leverage commodity hardware devices between the cloud and the terminal devices to support these services.

The smart community services can be differentiated according to the application hierarchy, as shown in Figure 1-3. Services such as healthcare monitoring, remote control, and grid control should be processed at the edge to reduce the delay of the services. Smart community networks should process in-transit traffic of the sensors to support smart community services. Smart community services should capture the in-transit data and carry out device identification through stream reconstruction of network traffic. For example, a smart energy management service that anonymizes and aggregate sensor data should capture the device identifier and data values by analyzing the network's RestAPI communications. Therefore, a platform that supports smart community services should have the ability to analyze the terminal devices' application layer data.

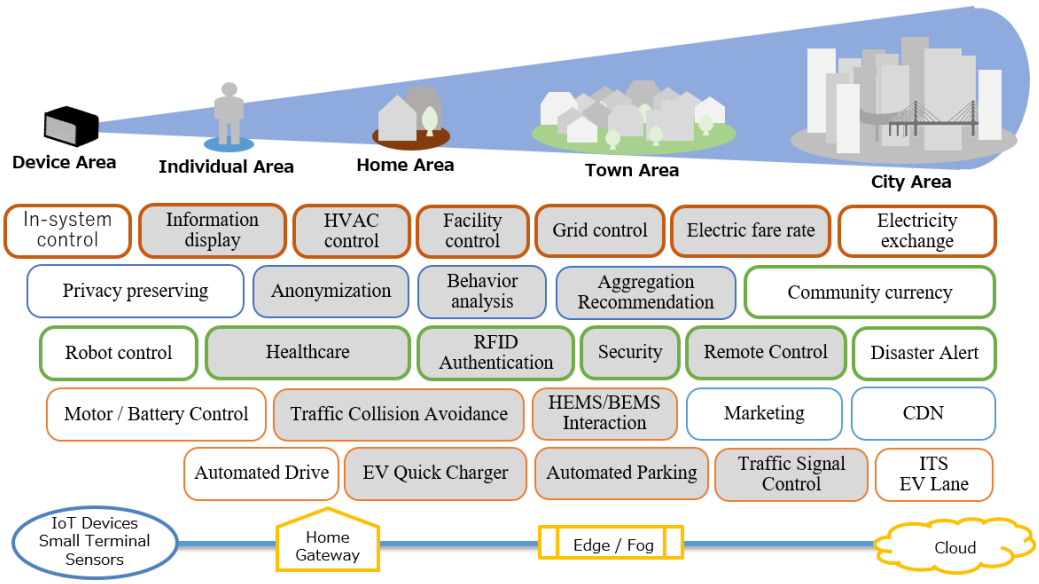


Figure 1-3 Smart community services hierarchy

Smart community networks can leverage these methods to provide network transparency, add-on services without updating or interacting with the terminal devices. For example, the smart community can provide privacy encapsulation for the network traffic by analyzing and watermarking in-transit data without updating the terminal devices' computation capabilities. However, smart community networks should process the traffic at 1-10Gbps bandwidth to meet Japanese households and enterprises' typical requirements, such as 10G-EPON, XGS-PON, and NG-PON2[9], [10], some of the popular optical passive optional networks for last-mile connections. Therefore, smart community networks are required to analyze the network traffic at 1-10Gbps bandwidth to support the services.

Smart community services require services to migrate from cloud to edge and edge to cloud depending on the network load. For example, the smart energy

management service requires the service containers to migrate from cloud to edge or one-N replicate to support the increasing network workloads, as shown in Figure 1-4. Smart community service could be migrated closer to the terminal devices or replicate into multiple locations to improve the latency and throughput of the service. Therefore, service migration is an essential requirement for these services. The migration process should also prevent the data loss of any in-transit data under migration as loss of sensor data could cause control problems such as a smart energy management system. Therefore, consistently guaranteed service migration is required for smart community services.

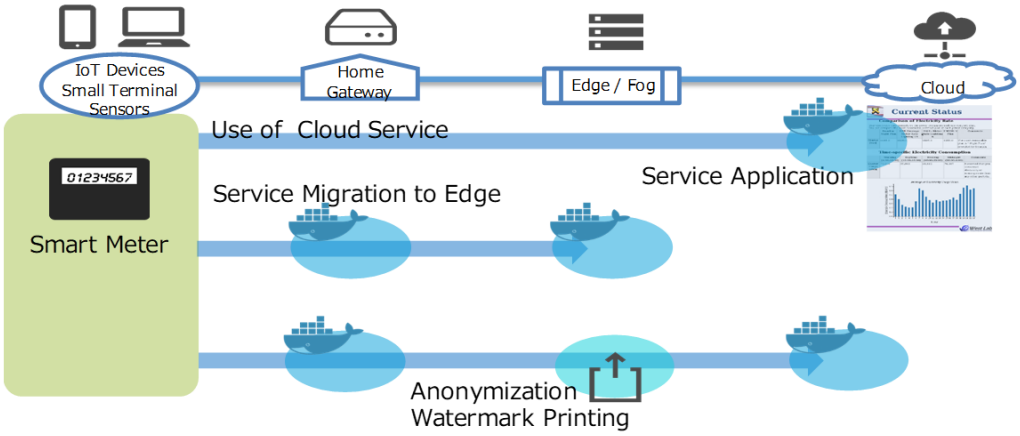


Figure 1-4 Typical smart community application

Smart community services commonly apply multiple services to the same network flow in a chain to provide a complete set of services. For example, the smart meter service first anonymizes the data, and then the data is sent to the aggregate node to record the data of multiple smart meters for processing and display, as shown in Figure 1-4. Therefore, smart community networks should support service chaining to distribute sensor traffic to execute these services in the required order. This requires smart community services to transit the correct sensor through nodes that support these services while reducing the end-to-end service delay of the network. Therefore, smart community networks should analyze in-transit network content, provide service migration, and service chaining to support SCE services.

1.2 Research Directions

SCE, a platform that provides SCA, service migration, service chaining, was proposed to this end. SCE platform proposes SCA to capture sensor data in the network. SCA is the process of stream reconstruction, application layer decoding, and string matching to identify and segregate network streams containing sensor data. SCA leverages the capabilities of packet forwarding and string matching

of data plane development kit (DPDK) and Hyperscan technologies to achieve 1-10Gpbs throughput. SCE platform uses DPDK to manipulate header and data sections of the packet to carry out SCA, while current software-based packet forwarding research [11]-[16] only manipulates the header. The SCA provides an interface for smart community services to access the sensor data. Furthermore, the SCE platform proposes a distributed rule application change method using a multi-layer architecture. SCE platform uses multiple layers to apply service rules to network flows to segregate network streams compared to packets. Compared to NFV [17], [18] research where virtual switches are used to segregate the packets, the SCE platform proposes a distributed rule application that allows smart community services to migrate through the smart community network without affecting the network flows. The SCE platform uses a service management layer(SML) to facilitate the distributed rule change process without affecting the SCA in the network forwarding layer.

In addition to the distributed rule change, the SCE platform uses the SML to facilitate multiple services. The management layer allows smart community services to run multiple applications in isolated software environments. Furthermore, the applications can directly capture data from the IoT terminal devices without being aware of the network control plane. The SML allows services to remote deployment and migration. SCE platform utilizes Docker containers to isolate the application services, allowing application migration without affecting the network flows. SCE platform proposed a consistency guarantee of the migration(CGM) to resolve the above issues. SCE platform CGM provides stateful application migration with data buffering to capture and store the network data within the migration downtime.

In comparison to current research [19] [20], techniques such as network storage devices improve the migration downtime. CGM uses container layer separation and buffered data separation through distributed rule applications to reduce data transfer and data buffering between the source and destination nodes. The SCE management layer communicates with destination nodes to identify transit traffic between the source and destination nodes. Afterward, the proposed method uses temporary buffers to store the in-transit data until the container restores at the destination node. Furthermore, the CGM can be used to provide one-many migration(O2NCGM) to support service replication. The CGM performance was compared with conventional methods through the hardware tests to evaluate CGM's effectiveness.

Furthermore, the SCE platform proposes a service function path(SFP) selection process to distribute traffic through SCE service chains optimally. The current path selection algorithms, such as the optimal path selection algorithm[21], use reactive data collection to identify the optimal SFP. However, this method consumes considerable network traffic and calculation time waiting for the nodes to advertise their loads. To resolve this, SCE proposed an optimized SFP selection method that periodically collects the load data to approximate the

system load within the advertisement period. The proposed SCE platform SFP selection process uses calculation cost and network delay to calculate the SFP cost. Then SFP calculation process calculates the cost of assigning all SFs to a single node to identify whether the single node assignment yields a better end-to-end delay. This method reduces the network traffic introduced by the node advertisements while increasing the overall SCE nodes' efficiency. The proposed optimized path selection algorithm is evaluated using a Cloudsim simulation environment compared to optimal and nearly-optimal SFP methods.

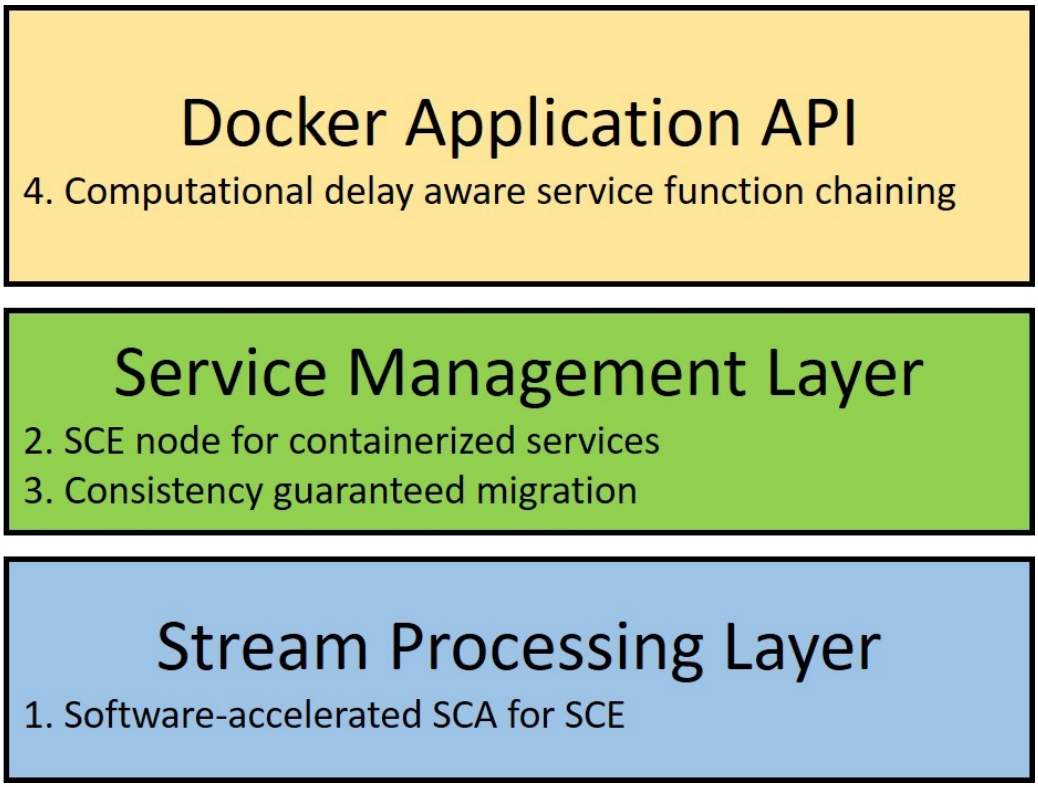


Figure 1-5 SCE platform layers

SCE platforms layered architecture provides the following capabilities and supports the requirements of smart community services mentioned above, as shown in Figure 1-5.

1. Software-accelerated SCA
2. SCE node for containerized services
3. Consistency guaranteed migration
4. Computational delay aware service function chaining

The place the dissertation resides in the current context is shown in Figure 1-6. The SCE node implementation is related to the network infrastructure layer where physical and virtual network devices operate to support the services. The SCE management layer and live container migration reside within the application management and orchestration, where consistency guarantee migration supports application orchestration. Finally, SFP selection resides in the application layer, where application request distribution is managed.

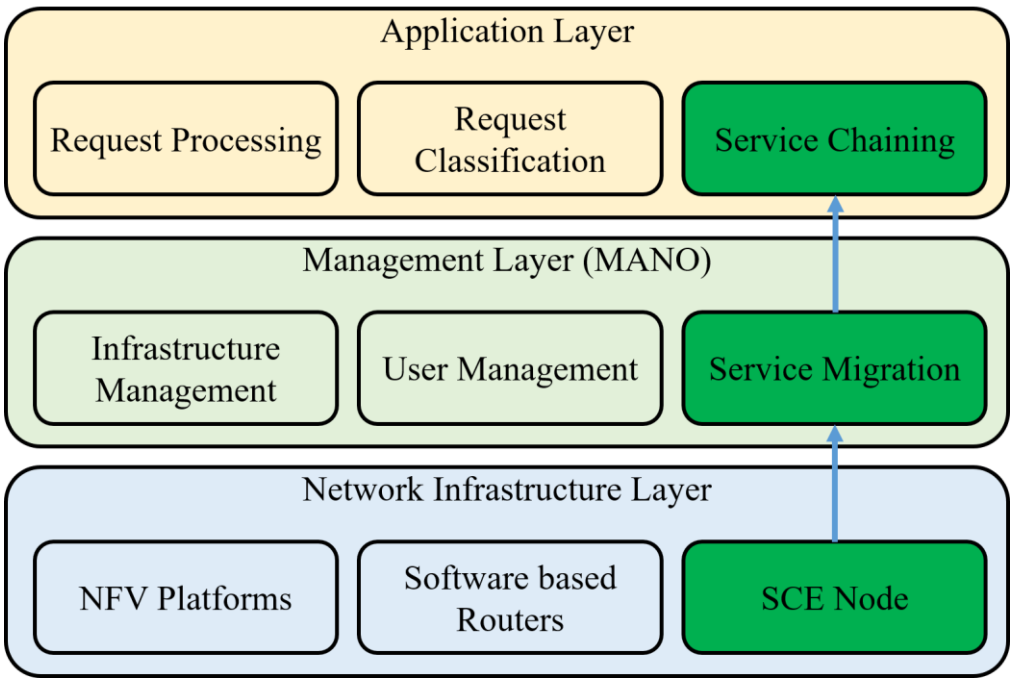


Figure 1-6 The place the dissertation resides within the current context

1.3 Dissertation structure

The dissertation structure is illustrated in Figure 1-7. Table 1-1 denotes a brief description of each chapter. As denoted in Figure 1-7, Chapter 2 explains the background studies associated with this dissertation. It provides a detailed explanation of the smart community, smart community services, service migration, and service function chaining. Moreover, the chapter briefly explains the issues with currently available methods and how the proposed methods will resolve those issues using the SCE.

Chapter 3 introduces the software-accelerating for SCA through DPDK and Hyperscan technology with the evaluation results. How the software-accelerated SCA is used to create SCE is explained in chapter 4. The chapter explains the components of the SCE node, the distributed rule change method, and its' core implementation details. In addition, it explains how the SCE application process is carried out within the multi-layer architecture. The chapter is concluded with the evaluation results of the SCE node comparison of the f-stack library.

Chapter 5 extends the ability of SCE by implementing CGM for services. This chapter explains the architecture of Docker-based live migration. Explain the implementation process of CGM through buffers. The evaluation results of CGM in Intel NUC computers are provided at the end of the chapter. Chapter 6 further extends the SCE platform by introducing the SFP selection method for service chaining to smart community networks. The proposed SFP selection algorithm is evaluated compared to currently available algorithms to identify its ability to execute user requests efficiently.

Finally, Chapter 7 summarize and concludes the dissertation.

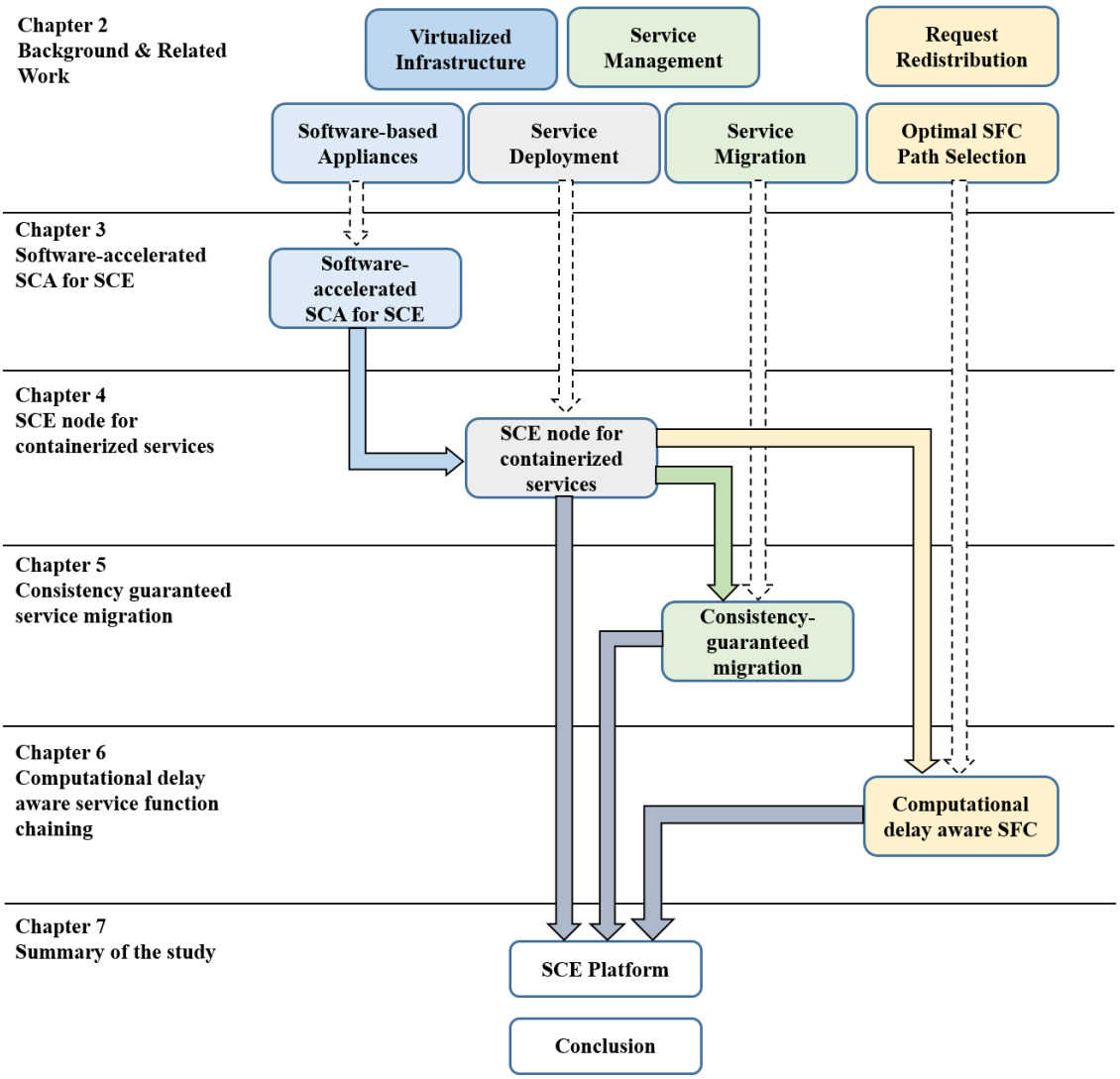


Figure 1-7 Dissertation structure

Table 1-1 Chapter description

Chapter 2	Purpose	Background study and survey of related work.
Chapter 3	Purpose	Implement and evaluate software-accelerated stream processing of SPL.
	Objectives	1) Find available methods to improve the performance of the SCA. 2) Handle 1-10Gbps throughput in SCA.
	Proposed Methods	Use DPDK, and Hyperscan libraries to improve the performance of SCA by using zero copy stream processing.
	Achievement	1) Implemented and evaluated SPL using DPDK and Hyperscan Technology. 2) Achieve 1-10Gbps throughput in multi-core server. The implementation and results are published.
Chapter 4	Purpose	Implement and evaluate SCE node to support containerized services.
	Objectives	1) Provide multi-service support for SCE service. 2) Provide distributed rule change to separate application traffic. 3) Handle 1-10Gbps throughput for multiple services at sub millisecond latency.
	Proposed Methods	Use modular multilayer architecture to support distributed rule change method that use of SPL and SML. Use Docker containers to containerize applications.
	Achievement	1) Implement multi-service support, and distributed rule change through SML. 2) Implement application API that support applications through Docker containers. 3) Achieve 1-10Gbps throughput for multi-service SCE platform while minimizing SCE latency to 0.8ms. The implementation and results are published.
Chapter 5	Purpose	Implement and evaluate CGM to support data consistency in Docker migration.
	Objectives	1) Provide network data consistency to Docker applications. 2) Reduce downtime of applications.
	Proposed Methods	Use container layer separation and application based data separation to buffer data at SML to guarantee data consistency of applications.
	Achievement	1) Implement and evaluate CGM in hardware platform.

Chapter 6	Purpose	Implement a SFP selection process to reduce end-to-end delay of SFCs.
	Objectives	1) Reduce end-to-end delay of SFC execution.
	Proposed Methods	Use computational delay aware SFC to gather node data to identify an optimal SFPs.
	Achievement	1) Implement SFP selection process that improves the end-to-end delay. 2) Evaluate the results compared to other available algorithms. Results are published and presented.
Chapter 7	Purpose	Summarize the details of SCE platform and conclude the dissertation.

Chapter 2 Background study and related work

2.1 Smart Community

Smart cities and smart communities are gaining momentum because of the technological advancements in smart electronic devices and sensors [22]. According to the smart communities guidebook [23] by the State University of San Diego, a smart community is defined as "a geographical area ranging in size from a neighborhood to a multi-county region, whose residents, organizations, and governing institutions are using information technology to transform their region in significant ways. Cooperation among government, industry, educators, and the citizenry, instead of individual groups acting in isolation, are preferred. The technological enhancements undertaken as part of this effort should result in fundamental rather than incremental changes." Smart communities are expected to realize a considerable increase in the number of electronic devices and sensors [24], also known as trillion sensors [25], for achieving cooperation through data exchange. These connected devices manage energy, information, communication technology(ICT), infrastructure, and citizens' lifestyles. As an example, smart community energy management services uses sensors attached to smart houses. Similarly, transport services would use sensors attached to vehicles to monitor road activity. Therefore, sensors become an integrated component of the smart community to monitor and manage the ecosystem.

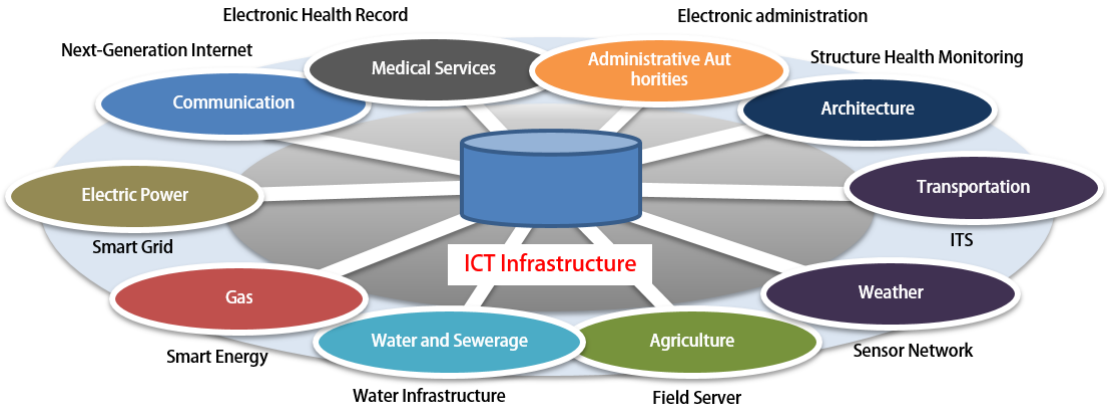


Figure 2-1 Concept of smart community

The smart community ICT infrastructure uses these data to provide different

services such as medical services, smart grid, smart energy management, and water infrastructure services, as shown in Figure 2-1. For example, the smart community utilizes smart energy management and smart grid service to collect live data to improve the electricity network's efficiency and balance. Smart community services become an integral part of the daily life of the citizens of smart communities and smart cities. Therefore, the smart community should efficiently manage and utilize these services to improve citizens' living standards while efficiently managing and utilizing the infrastructure and resources.

2.2 SCE services

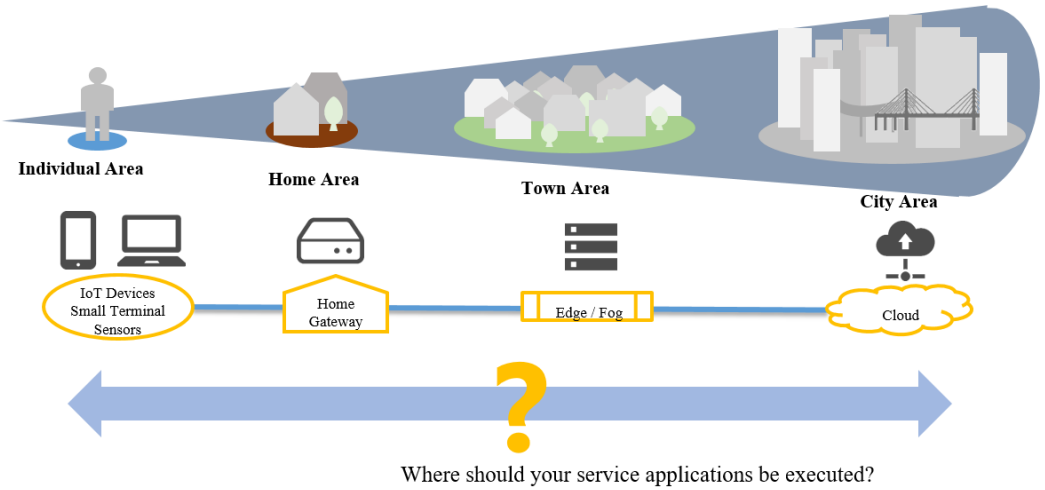


Figure 2-2 Execution location problem at smart community networks

The smart community and smart city ICT infrastructure and its related infrastructure require a significant amount of data processing and network transactions to smart community services. Therefore, the location of the execution of these services becomes a question, as shown in Figure 2-2. Cloud-based data processing has been extensively researched for the provisioning of cloud-based application services[26]-[28]. However, the increase in smart community sensors has incentivized the shift of computing resources from the cloud to the edge/fog for increasing network efficiency. As with a typical delay-sensitive application, a delay of less than 10 ms is permitted on the demand-side resource management for ancillary services using a smart grid [29]. According to Lema *et al.*[30], typical remote control services require less than 10 ms processing delay. Edge and fog computing bring computing resources closer to the network edge. Open Fog [31]-[34] is a category of service deployed closer to the terminal devices for improving the efficiency of the network infrastructure in next-generation networks. The edge and fog layer will act as

an additional processing layer between terminal devices and the cloud. In addition, decentralization and flexibility are the main advantages of edge/fog computing.

Consequently, the fog layer will improve the service latency and distribution in networks [7]. In addition, edge/fog computing conserves network bandwidth, reduces operating cost, enhances security, improves reliability, and boosts agility. Edge/fog has usability in systems such as smart grid management. Smart cities must manage electricity demand by real-time electrical consumption data. This kind of data can be effectively captured at the edge to operate the smart grid efficiently. Therefore, SCE should support services that are susceptible to network latency closer. Data processing at the edge, data aggregation, and caching can reduce such services' network delay. SCE services can also provide add-on services such as data watermarking and anonymization at the edge to support weak IoT terminals that send clear text private information, as shown in Figure 2-3.

The SCE services should be distributed on available nodes. This provides location flexibility to the services. However, the application services should handle the migration and sensor data to support such service migrations through the network. Furthermore, smart community services commonly operate in chains to provide multiple add on services to the terminal devices without adding software or hardware upgrades to end terminals. In such cases, the SCE should support service chaining to process these requests. In addition to the application, location flexibility requires the end terminal traffic to travel through the available SCE nodes to efficiently support service chains.

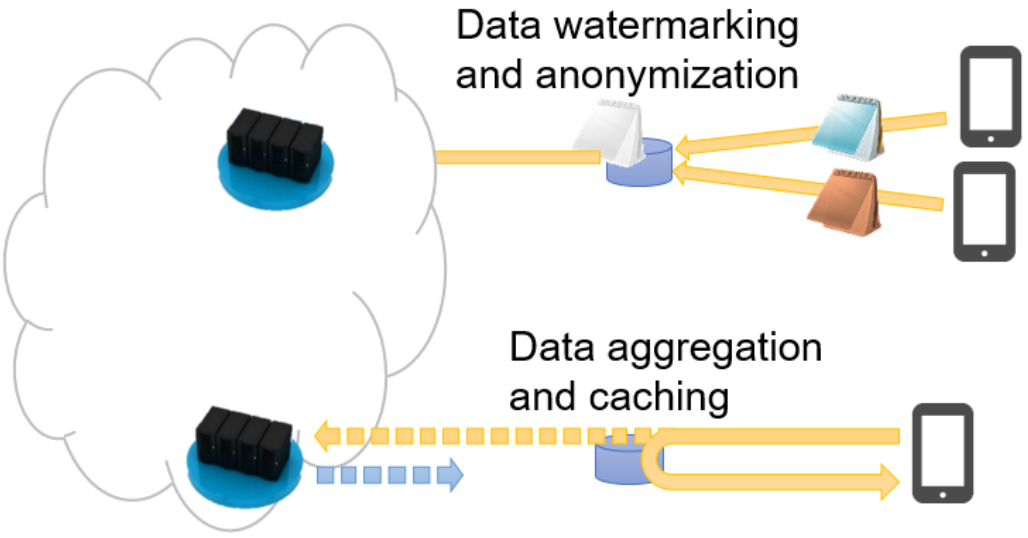


Figure 2-3 Sample applications for SCE

2.3 Related work

2.3.1 Software-accelerated SCA

Conventional routers and switches were designed to forward packets without intelligent payload analysis and inspection capabilities. The changes in network architecture, the introduction of electronic equipment and sensors, and big data require network equipment to be versatile enough to operate in smart community networks efficiently. Nishi laboratory initially proposed a simulation of a content-based router called Service-oriented Router[35] that could analyze the data streams travel through the network. SCE uses the SoR to implement its' Stream Processing Layer(SPL) to process smart community sensor data. Advanced packet processing techniques were required to handle a 1/10Gbps line rate to analyze SCE traffic. Network manufacturers use application-specific integrated circuits (ASICs) to handle the required bandwidth and computational power requirements [36], [37]. However, this is not a cost-effective method to manage cutting-edge networks such as smart community networks because the SPL should work with conventional hardware systems to ensure easy implementation and adaptability. In Japan, customer premises internet bandwidth can range from 1Mbps to 1Gbps [38], and enterprise core level usually handles bandwidths of 10Gbps. Therefore, customer premises or building-based SPL deployment would require 1Gbps bandwidth, and enterprise core deployment to require at least 10Gbps bandwidth.

Generally, packet processing applications in conventional servers are deployed in Linux using the Libpcap library, implementations of deep packet inspectors [39], and packet filters [40]. There are several examples of these implementations; however, there are limitations in performance when leveraging the Libpcap library due to its use of interrupt-driven NIC drivers. Furthermore, the implementation of multi-threading requires thread memory handling.

As a solution, the studies [41] discussed using packet processing offloading to a general-purpose graphics processing unit (GPGPU). Even though The GPGPU based SoR implementation [41] achieved 1Gbps throughput, it required a 100ms stream buffer wait time. Additionally, it is known that high-performance GPGPU availability is limited in the case of conventional servers. Moreover, GPGPU has an overhead for copying data from kernel memory to GPGPU memory. Therefore, the use of GPGPU is not a feasible method for SPL.

Additionally, compared to GPGPUs using Intel DPDK, [42] intel CPUs have the following advantages. The intel CPU cores have better performance compared to GPGPU kernel/processing elements. Additionally, one stream can be forwarded to a dedicated CPU to improve the cache-hit rate, thus improving the performance. Furthermore, the CPU cores can operate independently without memory transactions, whereas GPGPUs need memory transactions to transfer the data, and CPUs also have

a higher memory bandwidth compared to GPGPUs, which is limited by the PCI memory bandwidth.

Table 2-1 Summary of software-accelerated packet processing methods

Method	Advantages	Limitations
Libpcap	• Availability of applications such as packet filters	• Performance is limited due to interrupt driven drivers
GPGPU	• Improve the process offloading	• Overhead for copying data from kernel memory to GPGPU memory • CPU cores have better performance compared to GPGPU kernel elements
PF_RING ZC	• Provide buffers allocations in specific memory regions for multicore CPU direct access	• Limited multi-core support
DPDK	• Multicore support • Support libraries for packet processing • Provide highest degree of re-configurability	• Requires to use DPDK poll mode drivers

The SPL SCA requires packet processing libraries and hardware that support direct memory access and poll mode driver technologies. DPDK [43], netmap[15], and PF_RING ZC [44] are the major frameworks developed to overcome the issue with the Linux network stack. Netmap exposes the packet buffers to the application and allows system calls to transfer data. Software Router Click [13]and virtual switch VALE[45] show an increased performance using netmap. The PF_RING ZC is leveraging the use of zero packet copy, similar to Intel DPDK. PF_RING ZC buffers allocations in specific memory regions for multi-core CPU direct access. A network probe, nProbe, uses the PF_RING ZC ability to increase its performance. However, compared to these two frameworks, Intel DPDK offers multi-core support, supports libraries for packet processing, and has the highest degree of reconfigurability among the three frameworks [46], as given in Table 2-1. Even though Software Router Click [13]and virtual switch VALE[45] uses DPDK to forward the traffic through zero-copy buffers, they only forward packets without analyzing the payload. This limits their capability to analyze and support smart community services. Therefore, the SCE platform leverages the techniques of DPDK and integrate Hyperscan to analyze packet headers and payload to carry out SCA through zero-copy buffers of DPDK.

2.3.2 Software appliances for containerized services

SPL was initially proposed to capture sensor data through conventional routers. SPL supports a single service on a conventional server using SCA. SPL runs a single service without the use of virtualization technology or SCA content isolation. Therefore, SPL is unable to support multiple services in a single conventional server or an edge node. Other research on software-based network applications on conventional hardware is mainly designed for routing, deep packet inspection (DPI), and NFV. Software-based routers use conventional hardware such as commodity computing servers. Software-based routers have gained momentum in recent years [11]-[16]. Although hardware-based packet forwarding has better bandwidth than software-based systems, software-based routers' performance has improved because of the development of peripheral component interconnect technology and NIC designs [12]. The software-based routers were developed using solutions such as Intel DPDK [42], PF_RING [47], and Netmap [15]. DPDK and PF_RING use zero-copy packet processing to improve performance, while Netmap and other similar solutions focus on the modular processing of packets [48]. Studies on software-based routers provide an excellent platform for fast packet processing [11], [12]. However, these software-based routers do not carry out payload analysis and string matching of the sensor data.

The software-based DPI was developed using PF_RING technology [39], DPDK, and Hyperscan technologies [49], as given in Table 2-2. Deri *et al.* [39] proposed nDPI using PF_RING technology focused on high-throughput DPI. However, nDPI does not support additional services except the nDPI program, although it captures data from the end devices. Therefore, it is not possible to support smart community services using PF_RING-based nDPI implementation. Similarly, Luca *et al.* [49] proposed nDPI using DPDK technology to classify and block unwanted traffic. Their nDPI platform provides a method to classify the network flows; however, it does not support software isolation for the application processes using VM or container technologies. Therefore, applications can directly access other processes of a host machine. Consequently, application code inspection is necessary for DPDK-based nDPI solutions to guarantee the security and isolation of the nDPI program from multi-vendor applications [49]. Furthermore, These DPI methods only analyze the initial sections of the packets to identify the application flow. Therefore, their capability to continue capturing the in-transit traffic on the same flow is limited. These drawbacks limit the ability of nDPI in smart community environments. A service virtualization method is necessary to support edge service for multiple services at a single edge node. Furthermore, remote application deployment and migration without service isolation are not supported. These drawbacks limit the ability of nDPI to support multiple applications at the edge.

Virtualized DPI (vDPI) [50] uses DPDK technology and supports virtualized DPI applications using VMs. This implementation isolates the DPI instances through VMs and uses OpenVswitch [51] to share packets among the DPI nodes. Although this allows virtualization of DPI nodes, it is affected by the drawbacks of VM-based software isolation than container solutions. In addition, the use of a layer two switch to pipeline packets among vDPI applications causes all the vDPI instances to run stream reconstruction, which creates additional processing overhead by increasing network delay in the host machine. These drawbacks can be overcome by using SPL in place of the virtual switches. SCE platform proposes to runs SCA on the host machine using a distributed rule change method while sharing the captured content among service containers. Additionally, this allows for control over sensor data sharing, as the SPL can filter and share the sensor data among multi-vendor applications accordingly. Therefore, although these DPI solutions provide software-based platforms for multiple DPI applications, they cannot be optimized for virtualized smart community services.

Table 2-2 Summary of software-based DPI methods

Method	Usage	Limitations
Deri *et al.* [37] nDPI using PF_RING	• Focused on high-throughput DPI	• Does not support additional services except the nDPI program • Does not support software isolation for the application processes
Luca *et al.* [46] nDPI using DPDK	• Provides a method to classify the network flows	• Applications can directly access other processes of a host machine • Does not support software isolation for the application processes
Virtualized DPI (vDPI) [47]	• Use OpenVswitch [48] to support DPI applications using VMs	• The application rules applied at VM instance rather than using a distributed rule database at the packet forwarding layer

NFV was proposed along with the development of software-defined networking (SDN) to support virtual network functions (VNFs) and deliver network services as software processors [52], [53]. Major backbone router providers such as Cisco, Juniper, and NEC have proposed NFV platforms [37], [54], [55] using application-specific hardware. Although they achieve high-throughput packet forwarding, the hardware cost is high. Research on software-based NFV platforms [17], [18], [53] has been mainly with DPDK and SR-IOV [56] technologies. Intel SR-IOV allows hardware-based packet switching for VM-based VNFs. OpenVswitch [51] provides

software-based packet switching, functioning as a software switch to support core network services shown in Figure 2-4(1). The OpenVswitch transfers the network traffic through the services, acting as a virtualization switch without processing packet payload data. Similar software-based packet sharing solutions have been developed by sharing the huge page memory among VNFs [17], [18]. The huge page sharing causes network latency in the forwarding path because of the packet pipeline through each VNF instance in a core node. The raw packets are shared among VNFs as core network services such as network address translation, and switching [57] requires layer two and layer three information of all packets.

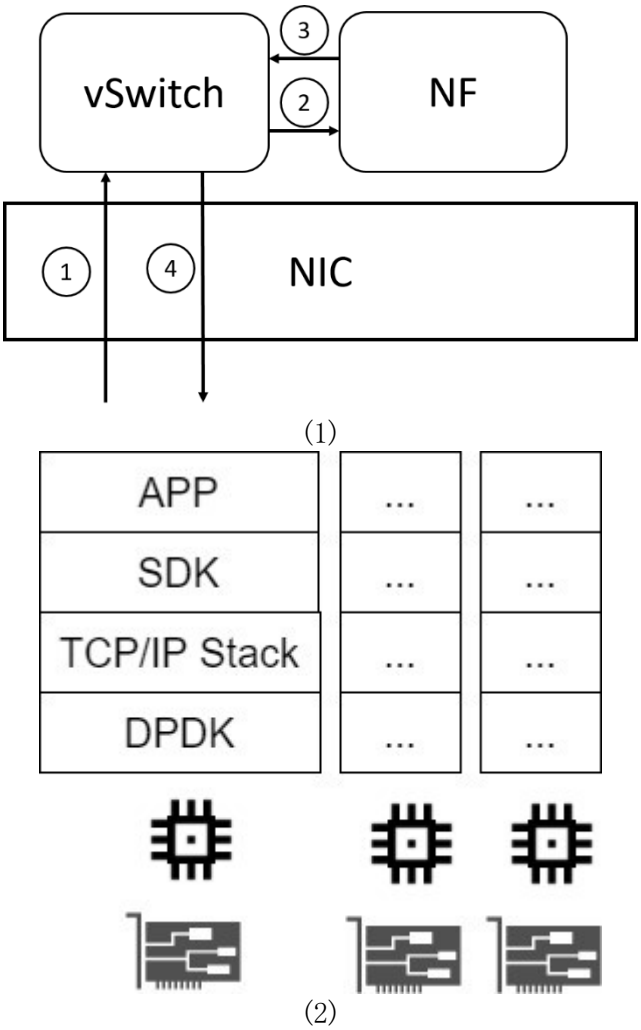


Figure 2-4 (1) OpenVSwitch and (2) f-stack

NFV does not provide smart community services requirements because its target is to optimize the core network. The use of NFV in the smart community requires implementing and interoperating the network stack for executing L7 service applications. Smart community services are different from the core services as

they are applied to the application layer, where most IoT terminals use the HTTP protocol and publisher, subscriber methods for data transactions. f-stack [58] provides a DPDK network stack to support applications using DPDK, TCP/IP, and f-stack SDK, as shown in Figure 2-4(2). The f-stack is designed to support web servers by improving its network throughput using the DPDK library. This allows the development of web services such as Nginx. However, f-stack was developed to replace the Linux network stack in web servers and does not support multiple services using virtualization technologies.

As described above, the conventional software-based routers and DPI [11], [12], [49], [59] solutions are not designed to support multiple edge services. The extant research [17], [18], [51], [60] focuses on supporting core network services, improving the network stack, or managing packet routing through DPDK technology. Using the distributed rule change method to identify and share the sensor data among services can overcome the limitations of using virtualized switches or buffers to share raw packets.

2.3.3 Service migration

Container migration is the process of moving a container between computers or storage devices. Migration technology has been developed to realize flexible services and to distribute services in the cloud dynamically. VM migration has been commonly used for the last decades in data centers. For instance, live migration of virtual machines was provided through VMware (vMotion), virtual disk migration by Storage vMotion [61], and live migration function of Xen and KVM [62] are used in VM migrations. These VM based migration techniques are developed as they operate entirely isolated from other VMs and physical hardware through hypervisors. Container virtualization uses similar techniques to migrate isolated containers. OpenVZ [63], LXC [64], Docker-runC [65] are some of the migration technologies used in typical container applications. These container migration technologies use Linux CRIU [66], where CRIU checkpoint and restore of containers are used in migration. CRIU enables us to save the state of the running container process to files using a checkpoint function. The container is restored in the migrated system using CRIU restore function. However, these techniques are not feasible for direct use of in-network service containers as they share resources and shared memory data structures among the application services and the host system for packet transactions.

Nadgowda et al. [20] proposed a migration architecture that supports the container running process and container storage migration through CRIU. This architecture speeds up the checkpoint function using a page server. Furthermore, the data transactions such as data copying are minimized using network-attached storage (NAS) [19] devices. NAS is used for storage sharing in containers in this implementation. However, it is challenging to use shared storage such as a NAS

in a software-based service management environment due to geographically widespread edge nodes. Furthermore, the smart community service platforms tend to operate using conventional hardware with limited processing power and lack attached middleware devices such as NAS. C. Dupont et al. [67] proposed migrations for IoT services in an edge computing environment. This method uses Docker [68] and Kubernetes [69] for containers and provides horizontal and vertical migration. Horizontal migration is used in IoT roaming, and vertical migration is used in IoT offloading. The horizontal migration performs application migration within the application layer; the vertical migration performs inter layer migration. However, these horizontal and vertical migrations are cold migration methods. Additionally, the system was designed toward stateless application migration through cold migration. Therefore, it is not possible to provide live migration or stateful application migration using this method. Smart community services such as ancillary services and traffic management are stateful services. Therefore, a novel migration method is needed for such services.

Table 2-3 Summary of migration methods

Method	Usage	Limitations
Nadgowda et al. [20], L. Ma et al. [67], and A. Machen et al[68]	• Data transactions such as data copying are minimized using NAS or share file systems	• Challenging to use shared storage in distributed SCE nodes • Doesn't consider application data management and data consistency required for network applications.
Dupont et al. [64]	• Provide horizontal and vertical migration for edge	• Only support cold migration
Gember-Jacobson et al. [69]	• Buffer all packets in transit on the network	• Susceptible to buffer overflow as it doesn't identify packets affected by migration flow before buffering.

Also, to live stateful migration, packet processing services should reduce the downtime for low latency services. Therefore, migration time should be reduced to minimize the overall downtime of application services. Some studies intend to decrease migration time by reducing bandwidth usage [70], [71]. They reduce bandwidth usage by separating the container layer's image layer and container layer and transferring only the container layer toward to destination node. L. Ma et al. [70] propose a shared file system under a distributed environment that consists of nodes with limited resources. In addition, L. Ma et al. [70] proposed a method to decrease migration time by reducing the amount of transferred data

using a shared file system. A. Machen et al. proposed a migration method that can provide low-latency services by leveraging LXC under Mobile Edge Clouds (MECs) [71]. This architecture leverages NAS storages and decided which layers of the container to migrate according to destination. However, these studies do not consider the transferred application data management and data consistency required for network applications. Therefore, these methods would cause packet loss and unordered packet streams. To resolve these in smart community services, we need a migration method that considers throughput and downtime while supporting latency reduction and data consistency.

Virtual network function migration was designed to reduce packet loss in network functions. A. Gember-Jacobson et al. proposed OpenNF, which controls forwarding rules and NF instances [72]. OpenNF achieved loss-free and order-preserved migration by buffering all packets until the migration finishes and then resending the packets to the destination node. However, this causes additional overhead and increases the migration time due to the significant number of forwarded packets to a controller [73]. Furthermore, a buffer overflow can occur due to the increased migration time. Moreover, this technique is bandwidth-consuming as it resending all packets from the source to the migration destination. L. Nobach et al. proposed the SliM, VNF migration method, which reduces bandwidth usage by transferring only the necessary packets [74]. However, these methods cannot support multiple services on a single node because it cannot detect the specific data flow. Furthermore, since these VNF migrations change the network flow, the other services run on the same node are affected by modifying the network flow. Therefore, a multi-service supported migration solution must support smart community services while minimizing the buffered data transactions.

2.3.4 Service Chaining

The service function chaining(SFC) can be used to chain multiple services for smart community networks. The SFC architecture can be separated into four layers [75]: service, overlay network, underlay network, and link. The service layer comprises SFC elements such as classifiers, SF forwarders (SFF), and SFC proxy. It uses the overlay network to ensure connectivity of SFC data plane elements. The overlay network uses overlay network technologies to interconnect SFC elements and works transparently to the service layer. The underlay network comprises networking techniques such as IP and MPLS. Finally, the link-layer consists of link-layer technologies that allow physical connections of the network.

The SFC is a service layer divided into operational, administrative, and management components (OAM) [75], including the SF, SFC, and classifier components. These components provide different services in SFC creation. The SF

component is an OAM solution that includes testing the SFs in any SFC-aware network devices such as classifiers and controllers. The SFC component includes solutions for testing SFC and service function path(SFP) that monitor the SFC forwarding path for packet matching a particular SFC as shown in

Figure 2-5. Classifiers are solutions for testing the validity of classification rules and detecting incoherence among different rules in different classifiers. These SFC components can be classified into management, control, and data planes based on their operation. Management elements include the SFC orchestrator, which is responsible for SF instances and SFC management. The Control plane creates the SFPs for SFCs by formulating forwarding rules. The Data plane includes the actual SFs, SF forwarders, and classifiers.

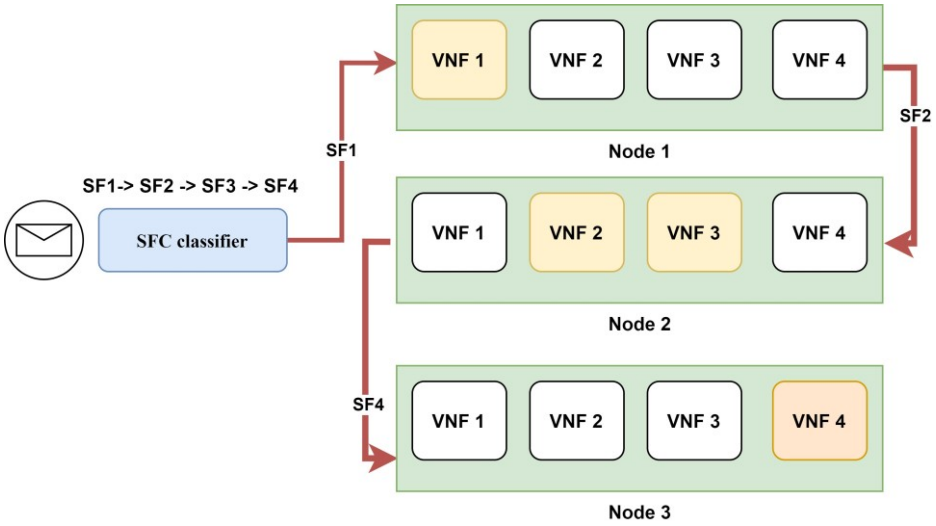


Figure 2-5 A simple illustration of a service function path

SFP selection algorithms were developed using OpenDayLight (ODL) [76]. ODL implements round-robin, random, load balance, and shortest path first algorithms. The round-robin algorithm distributes SFs in the SFC among the next available instance from all the available VNF instances. The random algorithm randomly selects VNF instances for the SFC SF abstracts. These two techniques do not consider network delays or capabilities of the VNF instances. Load balance uses the VNF instance load to deploy the VNFs in the SF without considering the network delay. Therefore, the load balancing algorithm tends to select nodes with lower loads, even though the SFP would have to go through multiple nodes, increasing the overall SFP delay. Later, near-optimal service-function path algorithm (NSP) [77] and optimal path selection algorithm (OPS) [78] were developed to address these issues by using dynamic programming; however, they have certain problems. NSP algorithm considers all instances of VNFs while selecting the SFP. The SFP selector uses the local load data of VNFs and network delay between the VNFs to select the SFP. However, in the selection, the algorithm selects one of the possible paths rather than the best path.

Additionally, this algorithm does not consider the hardware resources of the VNF instance; this can lead to the poor performance of the SFs because the NSP assumes that all VNF instances have similar hardware resource allocations to complete the SF. OPS algorithm was designed to address this issue by reactively collecting the load and queue load from the VNF instances. However, this technique has a severe drawback when there is considerable network delay between the SFP selector and the nodes. The network delay can cause the SFP selection algorithm to wait until the data arrives from the nodes, causing a delay in SFP allocation. Therefore, an efficient SFP selection method that could proactively collect node data could significantly improve the smart community services end-to-end delay.

Chapter 3 Software-accelerated SCA for SCE

3.1 Introduction

SPL provides services to end-users by performing SCA over network traffic flows. SCA is a new concept for providing services by analyzing stream contents, such as TCP streams using regular-expression-based string matching and extraction, rather than general IP packet analysis. Initially, SPL was to implement SCA using stream reconstruction, L7 decoding, and data inspection. The SPL provides contents analyzed using SCA hereafter, referring to SCA content to smart community services. However, SPL capabilities were tested during its initial development using the Hypertext Transfer Protocol (HTTP)[79] traffic. Analyzing HTTP streams in traffic and collecting useful data from HTTP transactions requires HTTP decoding and the gzip decoding process; therefore, the first implementation of SPL, while running on conventional hardware, faced several limitations.

The most commonly used packet processing library is the Packet Capture library (Libpcap)[80]. The Libpcap library's use leads to interrupts and memory copy from a NIC device to kernel space and then from kernel space to user space, thus increasing the processing time. String matching function causes a considerable delay because it requires several complex memory accesses, thus lowering the performance level. These are the top two performance bottlenecks of Libpcap-based SPL. Using the Libpcap library for packet handling and string filtering based on Boyer-Moore algorithm can only handle 8Mbps throughput without packet loss.

This chapter discusses foreseeable solutions for the problems described earlier using the DPDK and Hyperscan Library. The DPDK bypasses the kernel driver, thus avoiding interrupt triggers at packet arrival. Furthermore, it supports zero-copy that allows direct memory access in the user space, enabling high-speed packet access. This increases the layer-4 session-reconstruction performance by reducing the packet copying time. The packet filtration was developed using the Hyperscan library, which is optimized for Intel Xeon processors to match higher throughput strings. Even though it is not as highly parallel as GPGPU based implementations, the library achieves a high level of performance within the Intel processor architecture. Therefore, these two technologies can be used to solve performance bottlenecks without introducing any new network hardware.

The contributions of this chapter can be identified as follows:

- The design and implementation of an SPL with SCA using Libpcap library.
- The re-engineering of SPL using Intel DPDK and Hyperscan technology is explained, and SCA performance is benchmarked.
- The performance of the DPDK-based SPL is compared with the Libpcap-based SPL.

3.2 Implementation of Software-accelerated SCA

SPL was developed to provide new services brought up by the smart community. As discussed in the introduction, smart community sensors such as IoT devices are small terminals with limited processing power and memories to be small enough to get installed anywhere. In some cases, it is difficult to install a new protocol or give security patches because of its limited function and update cost. Moreover, the privacy information needs to be either encapsulated or anonymized before the data gathered by sensors arrives at the servers in a cloud.

SPL adds to the possibilities of smart community services. SPL enables the prevention of security attacks on the IoT devices by monitoring the communication streams leading to the devices. SPL can modify the streams without changing the IP headers. This function is called the SCA. The stream is selected, and the streamed contents are analyzed and updated if required. The implemented SPL consists of several modules, such as packet receiving, transmission, TCP stream reconstruction, L7 decode, static routing, stream filtration module, and information databases.

SPL packet receiver module accesses the NIC and stores packets in a buffer to be processed by the TCP stream reconstruction module; the TCP reconstruction process of a single thread is shown in Figure 3-1. The TCP reconstruction module accesses the packets loaded by the receiver module and then reconstructs the packets into separate TCP streams. The reconstructed streams then pass it to the L7 decode module. Initially, the L7 decode module decodes the HTTP messages and identifies the HTTP transmission parameters, such as the encoding methods and their HTTP version. Afterward, it decodes the stream into HTTP messages to obtain clear text as HTTP traffic information. The reconstructed HTTP traffic then passes to the stream filtration module, which checks the stream payload for the L7 transmission. The matched L7 traffic is then saved to the information databases for further processing by service applications such as data anonymization. Here, we suppose the web-based API or RESTful API to be the protocol between the IoT terminals and cloud servers. However, this function also extends to other protocols because the processes were designed using software algorithms.

While the packet payload is processed using the above modules, the header is used by the routing module to route the packet. The packet filtration module applies rules to the routing module to determine whether the outbound interface

packet should be forwarded. The transmission module then copies the processed packets to the NIC for transmission.

3.2.1 Stream processing layer with Libpcap

A Libpcap-based SPL is designed using the Packet Capture library. It uses the Libpcap offline-packet-dump file or physical network interface card in the promiscuous mode to input the packet receiver module. Simulations using packet dump files can be conducted using the pcap_open_offline function. We can access the interface traffic in live mode using the pcap_lookupnet function.

The transmission module contains a Libpcap buffer to handle the packets. The default allocation of this buffer needs to be adjusted for complex stream processing applications. We can achieve the required custom allocation of this buffer via the pcap_set_buffer_size function. The transmission module accesses packets using the pcap_loop function. Once the packet processing is complete, the loop function automatically operates on the buffer's next packet.

When the other modules consume the payload, the pcap_callback function separates the packet header and payload in the packet receiver module, as shown in Figure 3-2. The TCP reconstruction module identifies the five tuples of the packet and assigns the separated packets to various TCP streams. The packets are removed from the buffer after the streams expire in the time given by the stream timeout.

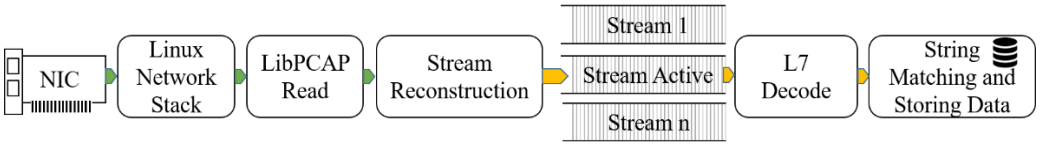


Figure 3-1 Libpcap single-threaded implementation of SPL

The L7 decoding module decodes packet payload by analyzing the HTTP header information. The decoding process uses the HTTP version of the payload, and if it is compressed with the GZIP algorithm, it is first decompressed. The plain text is processed using the Boyer-Moore algorithm[81]. The Libpcap-based packet pre-filtering is applied at the SPL initialization stage to the interface packet buffer along with the pcap_compile and pcap_setfilter functions. The pcap_compile function precompiles the rule base to improve the packet filtration performance in the application. The filtered packets, after reconstruction and decoding, are then matched using the Boyer-Moore algorithm.

In the Libpcap-based implementation, after the TCP stream reconstruction and filtration, the matched streams are stored in either the MySQL database[82] or on-memory databases, and the other streams are discarded after the TCP timeout.

The database insertion would not affect the system's total bandwidth, as the captured stream content is less than 10% of the total traffic flow.

It is necessary to implement a multi-threaded Libpcap-based SPL instead of a single-threaded operation to improve the Libpcap-based SPL's performance. Therefore, a multi-threaded system was developed using the Linux POSIX thread library[83]. The packets need to be distributed into stream processing threads to prevent packet loss. A packet of the same stream is assigned to the same stream processing thread using the five tuples, and a stream process distribution is achieved. Moreover, it can avoid the inter-threaded share memory access that deteriorates the stream processing performance.

In the threaded implementation, the pcap_callback function calculated the hash, then assigned the packet to a stream processing thread. Afterward, read the next packet in the Libpcap buffer as shown in Figure 3-2. When the selected thread buffer is full with packet processing, the pcap_callback function waits to select threads and assignment of a new packet until the last packet is processed and the buffer becomes available.

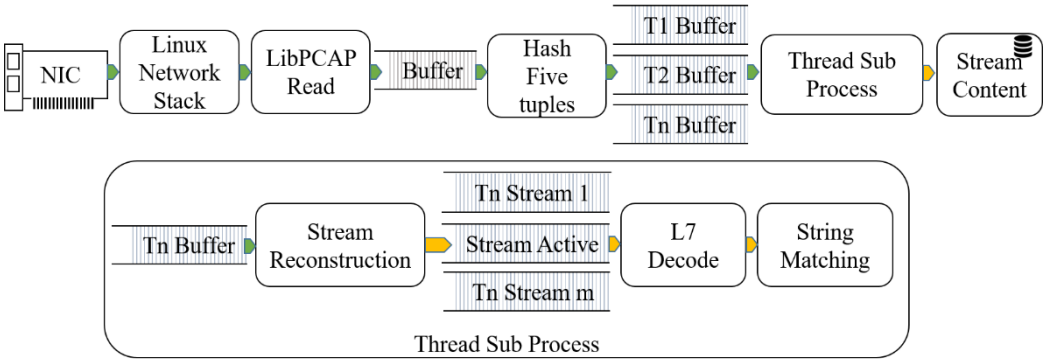


Figure 3-2 Libpcap multi-threaded implementation of SPL

Additionally, a thread operation of a single packet copy issued by the pcap_callbak function automatically cleans the memory. The callback function copies the packet data to the thread buffer if the thread buffer is empty, and the process then returns to the pcap_loop function to process the next packet.

3.2.2 Stream processing layer with Intel Data Plane Development Kit and Hyperscan

The DPDK-based SPL implementation with the DPDK library comprises all the SPL model components explained in the above section. The DPDK-based SPL process is illustrated in Figure 3-3, and the generic DPDK application processes are described in [54].

The major architectural difference in the Intel DPDK-based and Libpcap-based SPL is the modularization and core assignment for different processes without the

kernel scheduler's involvement. The Intel DPDK-based implementation uses multi-core support to allocate different modular processes to the CPU cores. In this architecture, the data transmission and receiving of NIC are allocated to two separate CPU cores that would poll and push packets toward a network without the intervention of Linux kernel system calls. The other cores are allocated toward the forwarding of packets and processing of payload to create streams. Furthermore, the string matching can be optimized using the Hyperscan library.

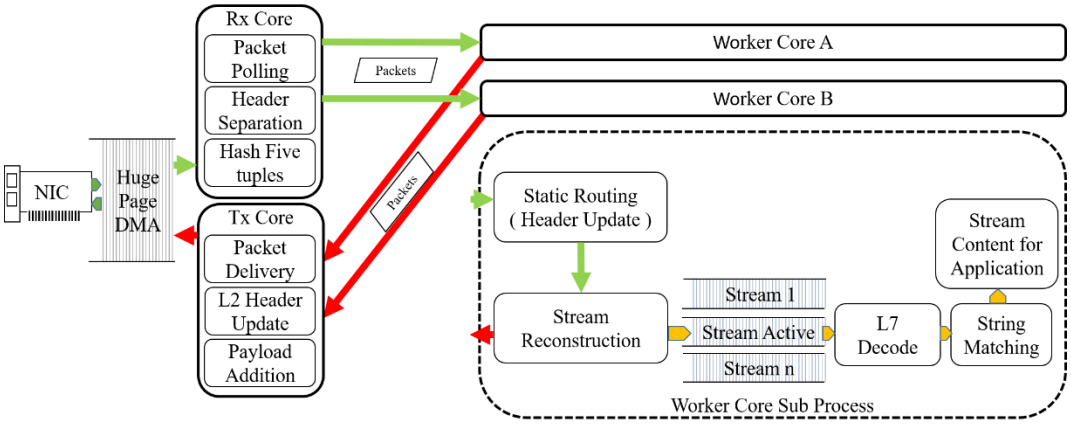


Figure 3-3 DPDK based implementation of SPL

In the DPDK-based SPL implementation, CPU cores are divided into worker cores, receiver cores, and transmission cores. These CPU cores are specifically isolated for the SPL functions using the isolcpu system call at the kernel scheduler. The DPDK-based SPL first parses the user application inputs required to initialize the DPDK runtime environment and SPL specifications. The parsed specifications are then used to initialize the SPL and DPDK runtime environment with the rte_eal_init system call. This sets up the DPDK ring buffers, CPU processors, and logs.

The application initializes the Hyperscan library[84], as shown in Figure 3-4. The hs_compile compiles the rule base into a binary to use in the string matching[84]. Moreover, the compiled Hyperscan rule databases need scratch space to buffer and scan the strings. The scratch space allocation is completed with the hs_scratch function[84]. Finally, to scan the packet stream, the stream needs to be opened and scanned with the hs_open_stream and hs_scan function, respectively[85]. In the Hyperscan stream scan, it is possible to scan packets as they arrive because the hs_scan function saves the previous packet's match state to be used for the next packet scanning [85] without creating a copy of the packet. After matching the Hyperscan, stream matching can be closed and released using the hs_close_stream function [85]. Moreover, when a match is detected, Hyperscan calls the on_match function with the specified rule number, allowing

operation on the matched streams [85].

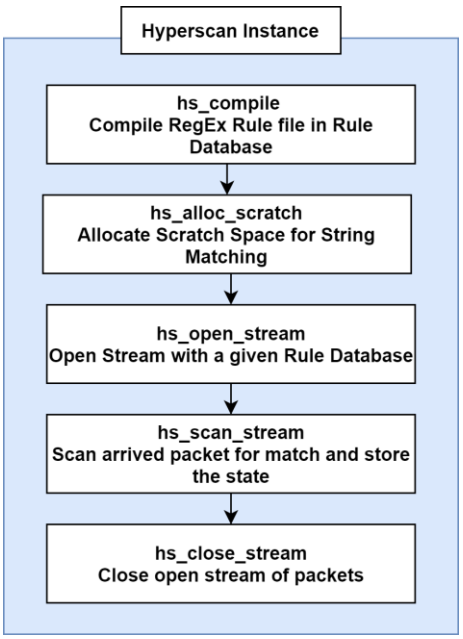


Figure 3-4 Hyperscan string matching process

After initializing the Hyperscan and other libraries, such as MySQL, the main processes call the other modules using the rte_eal_remote_launch DPDK function [85]. The DPDK-based SPL processor cores are divided into receiver-cores, transmission-cores, and worker-cores, as shown in Figure 3-5. The transmission-core will poll the NIC and share the packets to relevant worker cores through ring buffers. Similarly, receiver cores will receive packets from worker cores through ring buffers. Receiver cores then push the packets back to the NIC. The worker-cores are responsible for stream processing and analyzing functions other than packet receive and transmission. The DPDK enabled NICs to act as a poll mode driver because the NIC driver is directly mounted on the user space and the user program polls for packets from the huge page memory allocated for the NIC zero-copy-packet access. The receiver core in the SPL implementation first polls the NIC for packet bursts, and then the packets are assigned to worker cores according to the five tuples, similar to the multi-threaded implementation.

The worker core contains the TCP stream reconstruction, L7 decoding, and string matching module, along with a database insertion. The packets are first decoded, similar to the Libpcap-based SPL, and then assigned to Hyperscan-based string matching. The Hyperscan rule base is precompiled before the runtime for performance, and the TCP streams that match the Hyperscan rules are saved to databases housing the stream information. The packet routing is processed using another module that takes the packet header and updates the MAC address for forwarding. According to the Patricia tree algorithm, the packet routing module

forwards the packets, which sends the packets into the packet transmission module according to the forwarding and access control. The packet transmission module copies the packets back to the NIC memory, then forwards the packets to the output port.

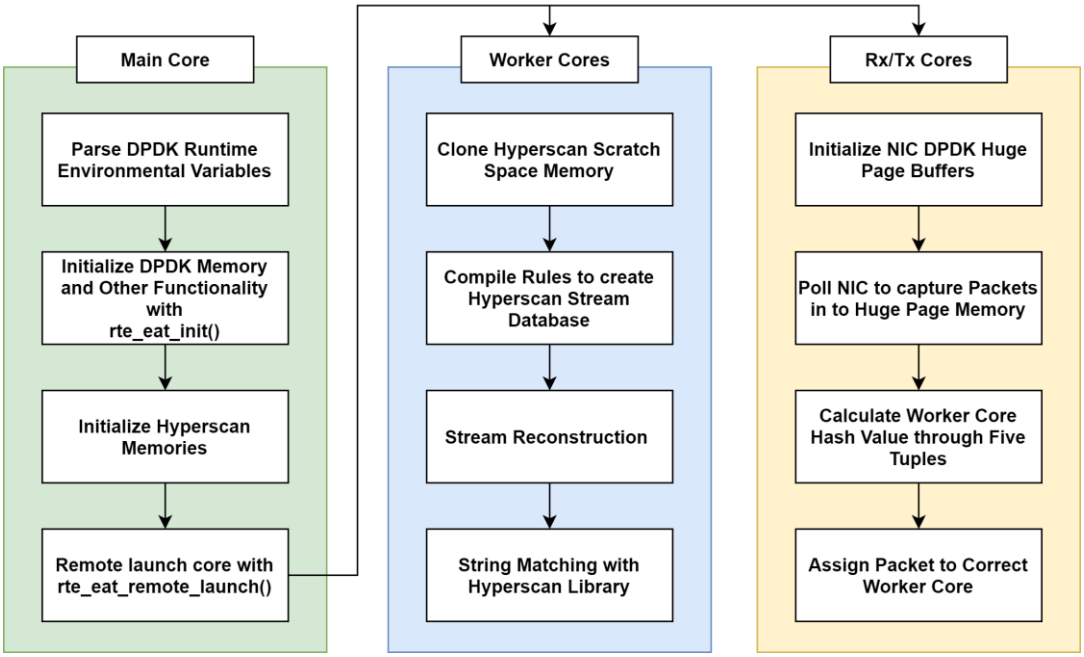


Figure 3-5 DPDK library multi-core assignment

3.3 Evaluation

The SPL performance was tested using a Dual CPU workstation configured with the following specifications:

- Server Type: HPC workstation
- Processor 1: Xeon ES-2620 v4 2.10 GHz
- Number of cores: 8
- Number of threads: 16
- Random Access Memory NUMA 1: 16 GB
- Processor 2: Xeon ES-2620 v4 2.10 GHz
- Number of cores: 8
- Number of threads: 16
- Random Access Memory NUMA 2: 16 GB
- Operating System: Centos 7.2
- Kernel version: Linux v3.1

The hardware usage was constrained in DPDK implementations using the DPDK environment abstraction layer. SPL is expected to work as routers capable of working as customer premises equipment (CPE) with 1Gbps of bandwidth and enterprise core at 10Gbps. Therefore, the setup was tested with a minimal hardware requirement. A 100GB packet dump was collected from Interop Tokyo 2016 Day 1.

Table 3-1 Experimental Results of SPL

Test	String Matching Algorithm	SPL Bandwidth	Average Packet Rate	# Threads	Memory Usage	# String Matching Threads	# CPU Cores
Libpcap single thread	Boyer-Moore	8Mbps	1.5kpps	1	640MB	1	1
Libpcap multi-thread	Boyer-Moore	150Mbps	29kpps	16	640MB	15	16
Libpcap multi-thread	Aho-Corasick	160Mbps	31kpps	16	640MB	15	16
DPDK-based SPL	Aho-Corasick	1Gbps	200kpps	6	4GB	2	6
DPDK-based SPL	Hyperscan	1Gbps	200kpps	6	4GB	2	6
DPDK-based SPL	Hyperscan	10Gbps	1200kpps	16	16GB	14	16

Table 3-1 shows the results of single and multi-threaded implementations tests for different line rates. The initial testing found that the single-threaded Libpcap-based SPL implementation could only handle 8Mbps without any packet loss. The throughput limitation in the worker subprocess is given in Table 3-2, where the string matching algorithm was operated, and the TCP reconstruction took place. This reduced the throughput of Libpcap-based SPL, causing packet loss in the Libpcap buffer. The multi-threaded Libpcap-based SPL implementation was developed

to avoid these limitations, which improved the Libpcap-based SPL bandwidth to 150Mbps and used all 16 CPU cores, as given in Table 1. However, the test performance was not adequate for the traffic speeds required for a CPE.

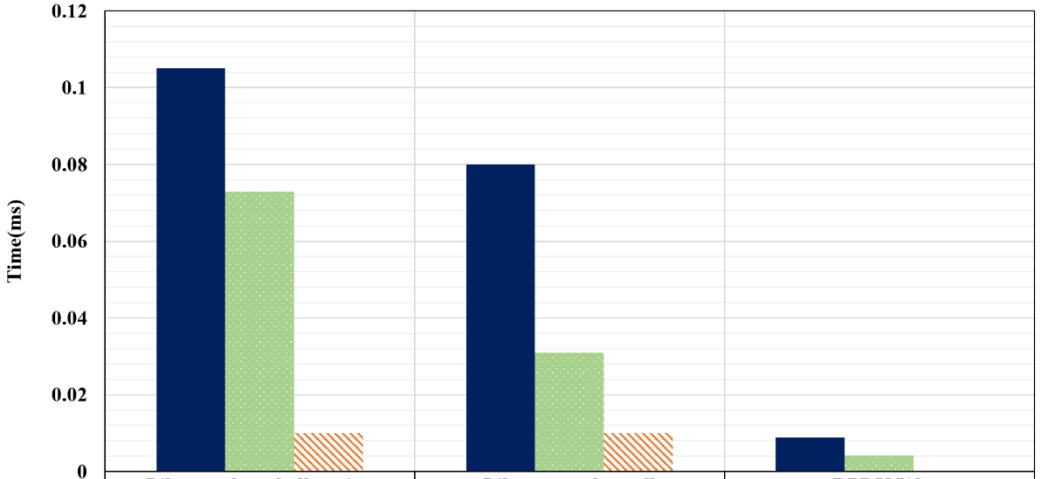
Table 3-2 Packet Throughput of SPL

Process	Packet Throughput
LibPcap thread sub process	1.7kpps
DPDK Aho-coarcisk worker core	190kpps
DPDK Hyperscan worker core	420kpps
LibPcap I/O thread allocation for packet	32kpps
DPDK I/O packet polling core	2400kpps

Additionally, the TCP streams usually arrive in bursts, overloading a single core operating on that TCP stream. This causes waiting in the receiver module and packet loss beyond 160Mbps. Both the single and multi-threaded Libpcap-based SPLs required 1GB of memory, which is caused by the initial buffer size allocation of 512MB corresponding to the bandwidth and latency of the SPL.

The DPDK-based SPL was then developed to handle the 1-10Gbps bandwidth with less than 16 CPU cores facilitating the SPL application to function in the fog as a CPE. The DPDK-based implementation performance and resource consumption were measured with test data with a line rate of 1Gbps and 10Gbps, as given in Table 3.1. The SPL handled the required 1Gbps line rate without any packet loss using 2GB of huge page allocation and 6 CPU cores. Additionally, the DPDK-based SPL with Hyperscan library could handle a 10Gpbs line rate without any packet loss using 16 CPU cores and 16GB of memory.

The DPDK packet buffers can handle a large number of packets, and the buffering is efficient than the Libpcap packet accessing method. The default page size is set to be 1GB to increase the performance of the memory access [60] in the SPL. Additionally, the SPL bandwidth was increased due to the poll mode driver. The isolcpu command allows user programs to be executed in isolated CPUs. Therefore, the worker modules operate in isolated CPUs without the intervention of the kernel scheduler. This increases the operating speed of the worker module. Furthermore, the Hyperscan string matching algorithm is faster than the other software-based algorithms [61]. This further increases the bandwidth of DPDK-based implementations. Though the DPDK-based implementations outperform Libpcap-based SPL, the CPUs are isolated from the kernel scheduler, causing them to be inaccessible for other operating system processes. Moreover, the memory allocation is dominated by the initial DPDK buffer allocation. The allocation is larger than the Libpcap-based allocation, as the DPDK buffer should handle the NIC buffers directly by the allocated huge page memories.

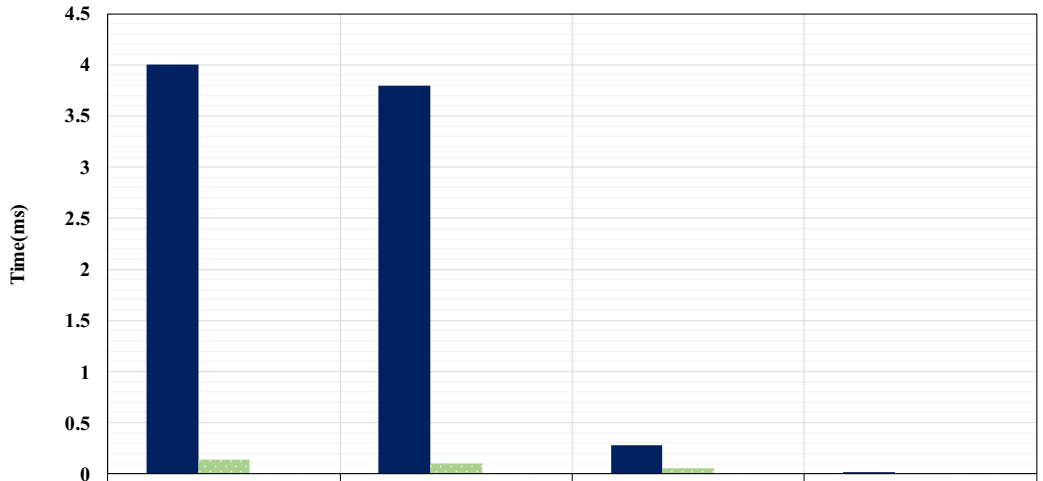


	Libpcap thread allocation	Libpcap packet call	DPDK I/O
Max	0.105	0.08	0.008834
Average	0.073	0.031	0.004216
Min	0.01	0.01	0.00001

Figure 3-6 Execution time of Libpcap-based and DPDK-based SPL transmission and receive cores

The DPDK-based and Libpcap-based SPL improvements are analyzed in terms of NIC access and packet thread allocation times. The yielded results are depicted in Figure 3-6. The average value of time is depicted by the horizontal bar. The Libpcap packet access time is considerably slower than DPDK, where the DPDK library polls the NIC in an average time of 0.0042ms compared to Libpcap 0.073ms. This is a considerable improvement from Libpcap when we consider the DPDK library poll all the buffer packets at this rate. Therefore, the DPDK library zero-copy capabilities can improve the NIC polling time to 0.0042ms and reduce the latency by 0.068ms for SPL.

Then the string matching was executed in SPL, and the obtained results are given in Figure 3-7. The Boyers-Moore, and Aho-Corasick average time needed to process a packet in Libpcap-based SoR was 0.143ms and 0.106ms, respectively. The worker core's average time for Libpcap-based implementation is higher than the DPDK implementation due to lockless buffer access and CPU isolation provided by the DPDK library. The Hyperscan library-based stream processing time is better than other algorithms, where 0.05ms reduce average execution time compared to the Aho-Corasick implementation. This improvement is essential as the Hyperscan process reduces the maximum time spent on a stream to 0.013ms, resulting in an improvement of 420kpps per core packet throughput as given in Table 3-2.



	Libpcap Bayers Moore	Libpcap Aho-corasick	DPDK-Ahocorasick	DPDK Hyperscan
Max	4	3.8	0.2842	0.013547
Average	0.143	0.106	0.053	0.0026
Min	0.0001	0.0001	0.000011	0.000011

Figure 3-7 Execution time of Libpcap-based and DPDK-based SPL string matching process

The DPDK-based SoR was demonstrated and tested in Interop 2017 Tokyo [62], ShowNet [63], and Network and Global City Team Challenge (GCTC) 2017 [64]. In these demonstrations, we displayed the user preference identification, security, and elastic location services. In particular, SPL could handle over 10Gbps of the traffic of relevant internet access of the Interop 2017 users in ShowNet.

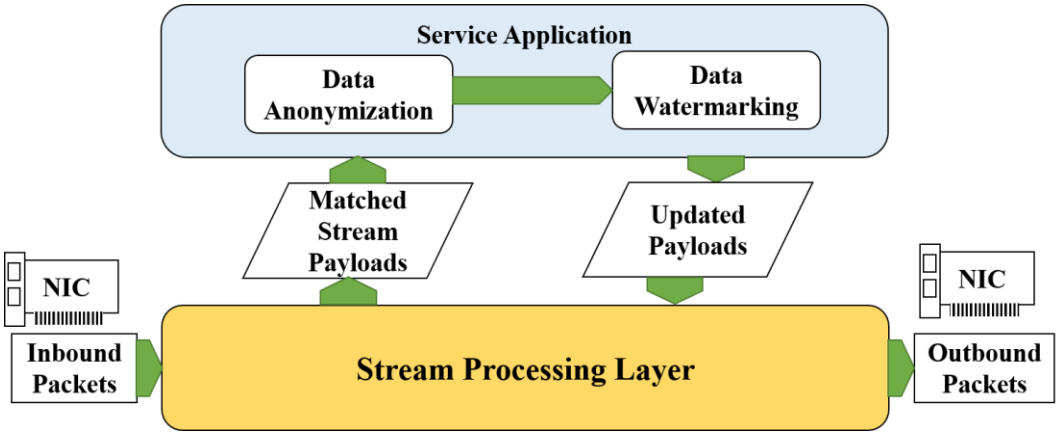


Figure 3-8 Data anonymization using SPL

DPDK-based SPL is currently operated under the UDCMi project [66] to deploy smart city services. The usage and operation of DPDK-based SPL were demonstrated in GCTC [67] by providing privacy for end IoT terminal devices. As shown in Figure 3-8, the DPDK-based SPL captured the personal energy usage data and sent the data for anonymization and watermarking. Anonymization protects personal information

in energy usage data. Watermarking protects the anonymized data from being published elsewhere. The watermark is used to identify the energy data that was anonymized by the service. The anonymized and watermarked payload was then sent to the remote servers. In this transaction, the SPL and the IoT terminal devices that handled all the privacy-preserving mechanisms were not required to perform any additional data processing.

DPDK-based SPL was used to identify the type of user in the ShowNet network in Interop 2017 Tokyo[62]. The implementation contained a service that ran on the SPL and identified the user types and preferences by the HTTP words and word2vec algorithm [65], as shown in Figure 3-9. This implementation was carried out in 16 worker cores at a line rate of 10Gbps.

In summary, Libpcap-based SPL implementations are limited by the 160Mbps throughput and packet access and thread allocation throughput of 32kpps. The DPDK-based SPL implementations reduce the packet access delay by more than 90%. DPDK-based SPL implementations require an intel processor with 8 CPU cores and 4GBs of memory to handle 1Gbps of line speeds and 16 CPU cores and 16GBs of memory to handle 10Gbps line speeds. In 10Gbps line rates, it is recommended to change the system default huge page size to 1GBs [60] that increases the page access performance of DPDK programs. Additionally, the Intel CPUs with SSSE3+ instruction extension is required to support Hyperscan implementations. The NICs selected for such a system should also support the DMA technology of the DPDK library [58]. The SPL bandwidth scalability is limited by the per core packet throughput of 0.4Mpps as the reader-writer cores can handle higher packet throughput than worker cores.

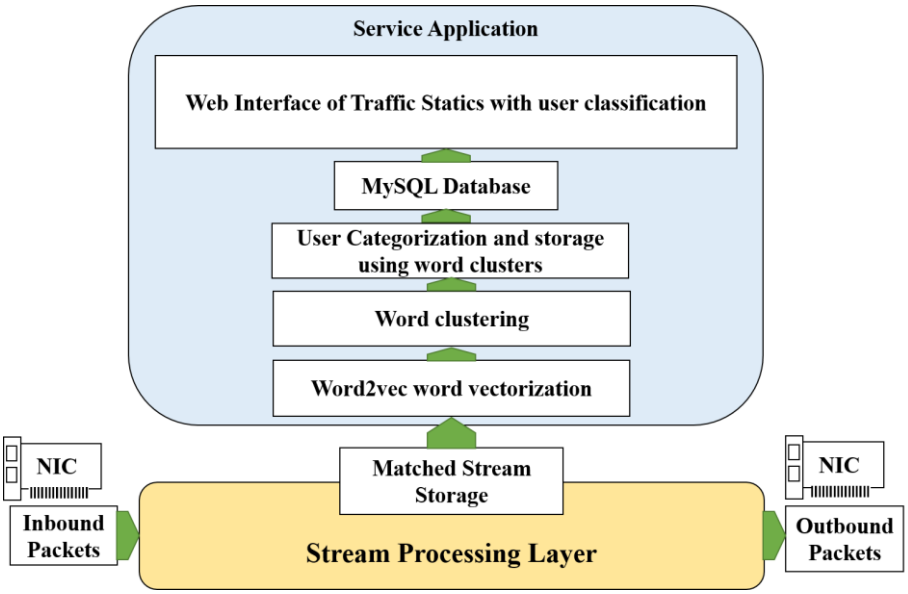


Figure 3-9 Application data identification through word2vec algorithm

3.4 Conclusion

This chapter summarized the implementation, experiments, and test results of upgrading the SPL to use the Intel DPDK and Hyperscan technology. The results demonstrate that the upgraded SPL can perform SCA at 1Gbps line rates using only eight cores. SPL realized this by using DPDK and Hyperscan to carry out SCA in zero-copy packet buffers. This is a 0.8Gbps throughput increase compared to the older versions of Libpcap-based SPL. Moreover, upgraded DPDK-based SPL achieved a 10Gbps line rate with 16 CPU cores. The DPDK-based SPL was demonstrated in ShowNet to prove its capability to work in core ISP networks.

Moreover, the SPL was demonstrated in GCTC to provide anonymization services using electrical power usage data. Therefore, the experiments discussed above and the obtained results demonstrate that the DPDK-based SPL can work as a gateway device to perform SCA in smart community networks. However, SPLs applicability was limited as it cannot provide distributed rule change for multiple smart community services. Furthermore, SPL should support service virtualization techniques to provide SCA content for multiple services.

Chapter 4 SCE node for containerized services

4.1 Introduction

This chapter proposes an SCE node using multi-service SCA (MSSCA) to support smart community services in conventional hardware systems. SCE node is designed to execute L7 services without affecting network flows. Therefore, SCE services can capture and operate on sensor data independent of the network protocols, location, and IoT nodes. SCE nodes aim to acquire and separate the information required by different services through MSSCA and then transform it into smart community services. MSSCA requires SCE nodes to monitor the data streams and use string matching to acquire the L7 data, that is, the SCA content. Although MSSCA acquires SCA content from network traffic. it provides additional capabilities than SCA. MSSCA is designed to support multiple services, remote deployment, and runtime changes in services. Therefore, MSSCA allows the SCE node to initiate, terminate, or migrate services at runtime by supporting runtime distributed string matching regular expression changes. In addition, MSSCA identifies and tags MSSCA content according to services that allow data isolation. The proposed SCE node will act as an edge node for separating and sharing the SCA content among the service containers. SCE node uses conventional and compatible virtualization through Docker containers for service isolation, deployment, and migration. SCE node consists of a modular architecture that encompasses a management layer, which provides service management and deployment.

According to the data format, current IoT protocols use XML-based data descriptions. Popular smart community-oriented protocols such as IEEE1888 [86] and OpenADR [87] are composed in XML.-It is assumed that an SCE node using human-readable protocols is required to handle at least four services. Running these services at the network edge removes the round trip network delay in a cloud-based deployment. As an example, a text-based protocol such as OpenADR requires approximately 50 parameters to achieve demand control. Therefore, an edge node should extract approximately 10-100 parameters for sensing and controlling dedicated facilities' current status.

For high-throughput, low-latency network stream data handling, the SCE node uses a DPDK [42] and Hyperscan [84] library. Furthermore, the service management layer (SML) and the service application programming interface (API) use Docker [88] container technology for service management. SCE nodes leverage DPDK with SML to increase the packet access and forwarding performance of an SPL through

direct memory access (DMA) of packets and service isolation from the packet forwarding layer. MSSCA string matching is carried out using the Hyperscan library [85], [89]. SCE platform MSSCA supports runtime rule change, data tagging, and data separation using SML and shared memories. This enables faster MSSCA for containerized services. MSSCA proposes a distributed Hyperscan dictionary change method using shared memories to support dynamic regular expression changes from multiple services. Smart community services are supported using Docker containers because of their deployment speed over virtual machines (VMs) [90]. SML manages the MSSCA content transfer between the SML and services through inter-process communication using shared memories. Finally, the service layer API allows the services to access the MSSCA content to acquire IoT data.

In summary, this chapter makes the following contributions:

- The SCE node's architecture is proposed, which supports smart community network services using conventional hardware devices.
- MSSCA is proposed, and its implementation and functionality are explained, which supports distributed regular expression dictionary switching capabilities at runtime through Hyperscan and shared memory techniques without interrupting the packet flow.
- MSSCA-based edge node fulfills the requirement of 10 ms maximum delay for service provisioning over eight services at one node and over 100 string matches to support all target services parameters.

4.2 Implementation of the SCE node

SCE node was implemented to provide MSSCA and service management for deploying services. SCE node contains three main layers: SPL, SML, and an application layer comprising a service API. The layered architecture isolates services from the packet forwarding flow. The layered architecture allows packet forwarding without any interference from the services. These functionalities are provided by three major components in the SCE node, as shown in Figure 4-1. SPL processes the packets, creates the stream data for services, and routes the packets. SML manages the services and works as a gateway between the services and SPL. The service API supports the deployment of services by providing library functions to access SPL and SML. SPL and SML interact to transfer SCA content toward the applications and regular expressions to SPL. SML interacts with services to transfer the application of regular expressions and MSSCA content to applications. SML regular expression transfer allows SPL to operate without being aware of the application services.

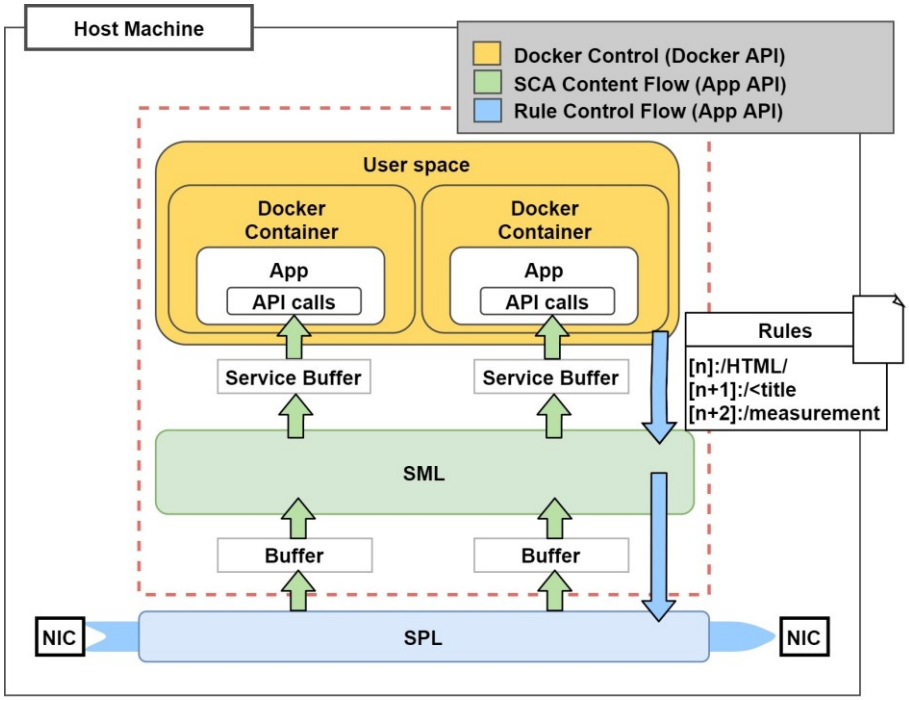


Figure 4-1 Three layers on the SCE node

4.2.1 Stream processing layer implementation with SML communication

SPL is designed to forward the packets while supporting TCP reconstruction and string matching for MSSCA. SPL is separated from SML such that it allows regular expression dictionary change at runtime without any significant disturbance to the routing layer. SPL is designed to provide the following functionalities:

- Static routing
- MSSCA, according to regular expressions of the services
- Pushing MSSCA content toward the SML
- Provides a virtual interface for services to access the network.

SPL's core modules and dataflow are shown in Figure 4-2. SPL uses the DPDK library DMA to avoid interrupts and memory copy system calls generated by generic NIC drivers. DPDK-based SPL improves packet access bandwidth and supports lockless multicore processing of packets. Additionally, the DPDK-based multicore modular architecture provides scalability by allowing instruction scheduling on a given processor core without a kernel scheduler. SPL core modules are allocated as transmission (Tx) cores, receive (Rx) cores, and worker cores. The SPL can adapt to bandwidth requirements by increasing the number of worker cores.

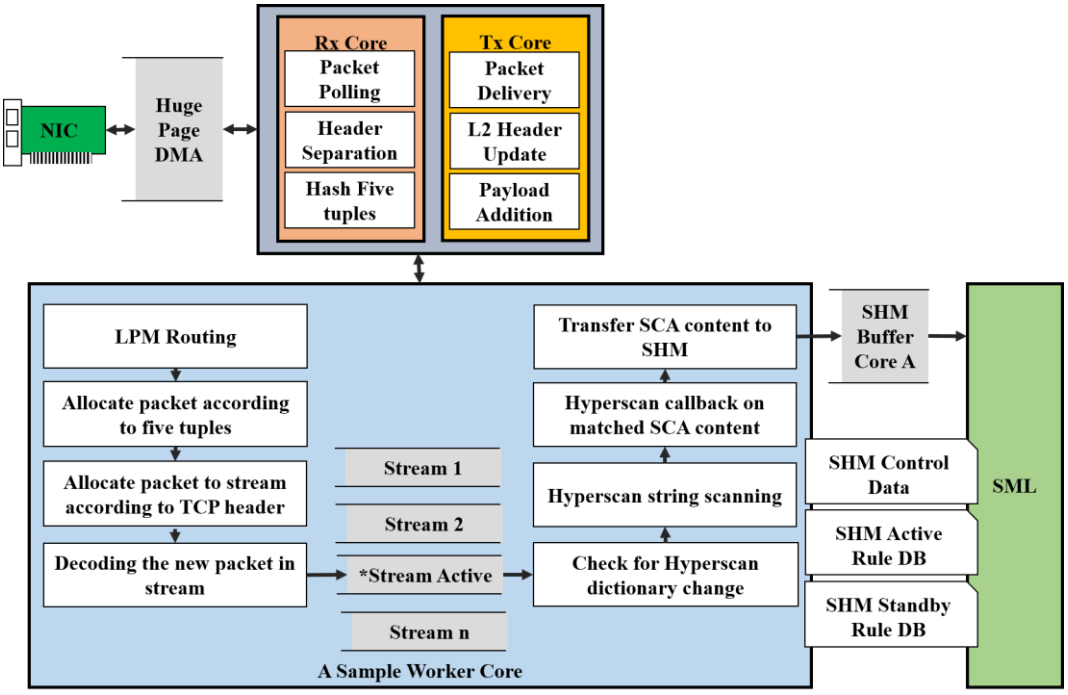


Figure 4-2 Implementation of SPL with SML communication

The operation of Rx and Tx cores of SPL is similar to that of the Rx and Tx cores of chapter 3 implementation. The worker cores are designed to provide routing and MSSCA and share the data with SML. SPL uses the longest prefix matching (LPM) rule to calculate the exit port of the packets. The LPM is implemented through DPDK LPM functions, allowing developers to define static routes for forwarding packets to different Tx ports. The worker core accesses packets through the Rx core to worker core ring buffers. The packets are then assigned to an MSSCA process according to five-tuple information and client-server communication information through layers with two, three, and four headers. MSSCA uses a stream reconstruction process similar to chapter 3 implementation. However, the string matching and multi-service content capture and separation are different from the SPL SCA.

To achieve faster MSSCA, the SPL scans each arriving packet using the Hyperscan stream scanning technique. The first decoded packet in a stream is assigned to a Hyperscan hs_stream_t object. The Hyperscan hs_stream_t object allows the string matching functions to track the last string matching the stream's state. The Hyperscan hs_stream_t object is then called to scan a new packet in the stream using the hs_scan_stream function. If a match occurs, it will execute a call back function. The call back function identifies the SCA content through a unique identifier—a rule identifier (ID) associated with the regular expressions applied by different services. The MSSCA content is then transferred to the management

layer through the SHM buffer of the worker core with the associated rule ID.

Prototype SPL implementation supports TCP and HTTP protocols because IoT transactions commonly use the HTTP protocol. The captured SCA content will push into the SML ring buffers. SPL and SML are separated using ring buffers to allow uninterrupted packet forwarding at SPL. Finally, Tx cores will read the ring buffers connecting worker cores for forwarded packets and write the packets back into the NIC using the rte_ring_sc_dequeue_bulk function.

4.2.2 Service management layer

SML is designed as a separate entity to allow SPL to operate without interruptions and the knowledge of services. Additionally, this allows for a faster regular expression compilation and switching mechanism without disturbing SPL. SML provides the following functionalities:

- Pre-compiling and managing regular expression dictionary
- Communication and management of SPL
- Facilitating communication for services using shared memory and UNIX sockets
- Sorting and distributing MSSCA content into relevant services
- Facilitating service initialization, termination, and migration

SML contains three main threads for MSSCA content sorting, rule management, and service communication, as shown in Figure 4-5 Processors of SML. The processes are isolated using threading for continuous delivery of MSSCA content to services without any interruptions. Additionally, a separate communication thread is required to communicate with multiple services and detect new service deployments.

Fast string-matching libraries such as Hyperscan use precompiled regular expression dictionaries to improve performance. Therefore, rule pre-compilation prevents dynamic runtime rule change in a string-matching library without stopping the string matching and packet forwarding processes. In addition, regular expression management and rule pre-compilation require significant processing time. This becomes a significant issue for network applications such as SCE services, where multiple services try to change the string matching rules of the network flow. SCE node resolves this by separating regular expression dictionary management and string matching to SML and SPL, respectively. The layer separation allows SPL to process and forward packets without being aware of the management and pre-compilation of regular expression dictionaries.

The proposed remote dictionary change method separates stream processing and dictionary compilation from SPL and SML's isolated processors. However, the compiled dictionary is shared with SPL through shared memory buffers of SHM active rule DB and SHM standby rule DB, as shown in Figure 4-3. The proposed rule change

method is shown in Figure 4-4. Once an application adds or removes a regular expression rule, SML will add or remove its rule manager rule. Then, the SML rule manager requests SPL to release the SHM standby DB shared memory.

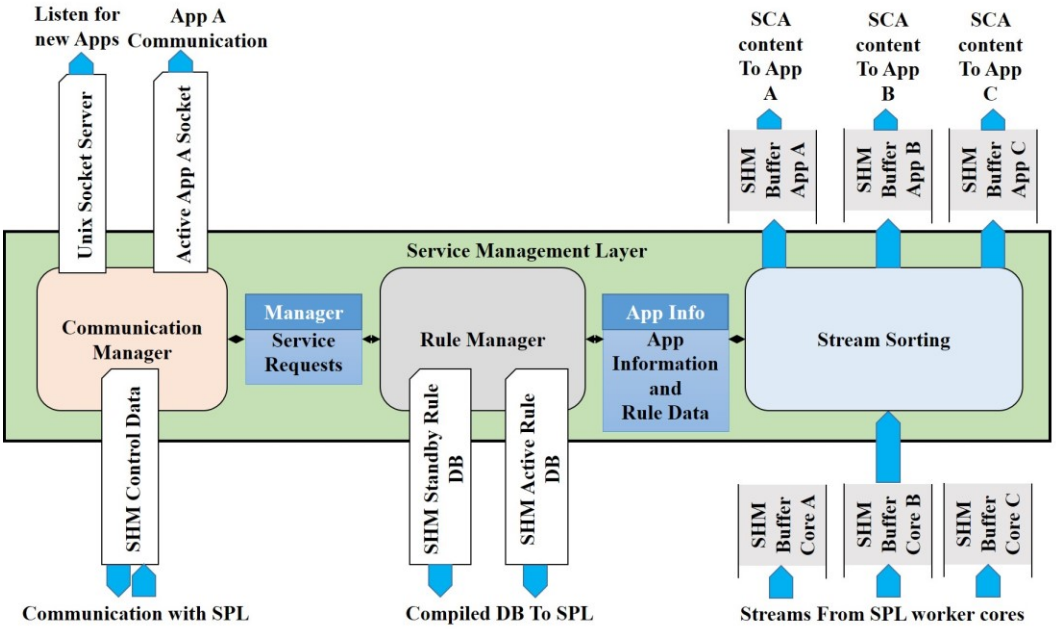


Figure 4-3 Implementation of service management layer

Further, the SML process compiles the regular expressions with the hs_compile function to generate an hs_database structure. However, this structure needs to be serialized in order to be shared with a remote process. Therefore, the compiled rule DB is serialized on the SHM standby DB shared memory location. Then, the SML process informs SPL about the regular expression dictionary change. The SML dictionary change allows the SPL process to de-serialize the new rule database and allocates a scratch space for string matching with the new rule DB. Finally, SPL changes the SHM standby rule DB to the active state and the SHM active rule DB to the standby state. This methodology allows remote dictionary change in SCE nodes without any significant interruptions to the string matching process.

The rule manager's primary processors, stream sorter, and communication threads are shown in Figure 4-5. The SML allows continuous content delivery and communication through multi-threading. The communication thread continuously listens to socket communication from the services. It also initializes the services and creates shared memory for communication and MSSCA content management. Further, the communication thread stores any service requests in a shared structure. The rule manager thread consumes the requests in the shared structure.

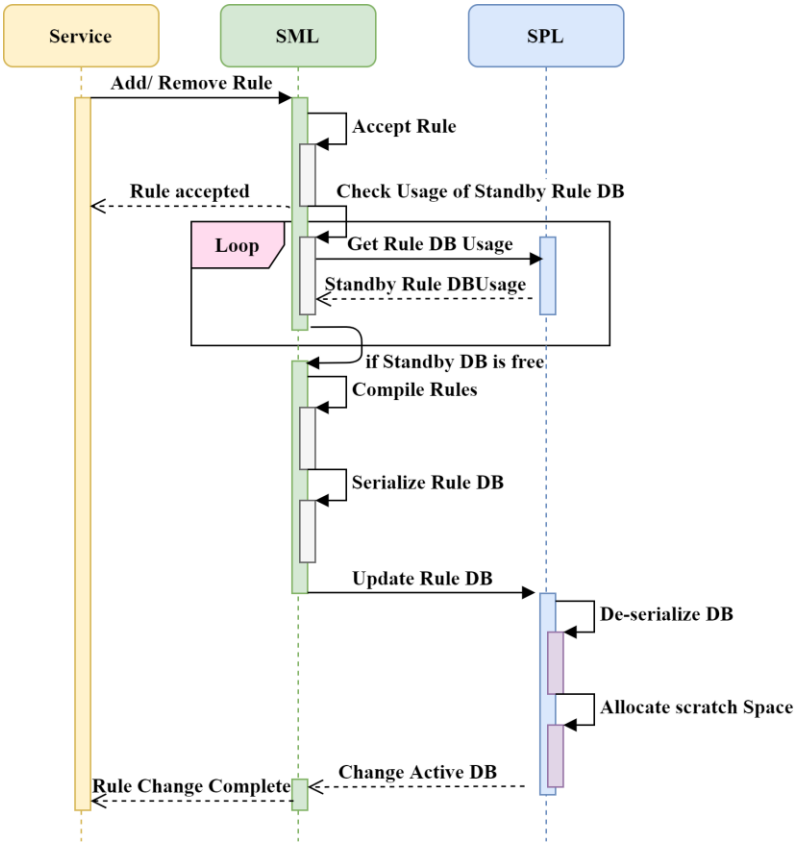


Figure 4-4 Runtime rule change process of MSSCA

Services that need to acquire MSSCA content for a given rule set should initially communicate with SML to apply the ruleset to SPL. Then, the rule manager thread pre-compiles the Hyperscan database and updates the rule database. Following initialization and rule application, the communication thread sends a reply to the service. Once the application receives this message, it can use threaded API calls to communicate with SML without interrupting the service application. The stream sorter thread sorts the MSSCA content according to the service using the app information. The rule manager thread is separated from the stream sorter thread because of the higher processing time of dictionary pre-compilation. Stream sorter and communication threading allow multi-service communication and continuous MSSCA content delivery to the services.

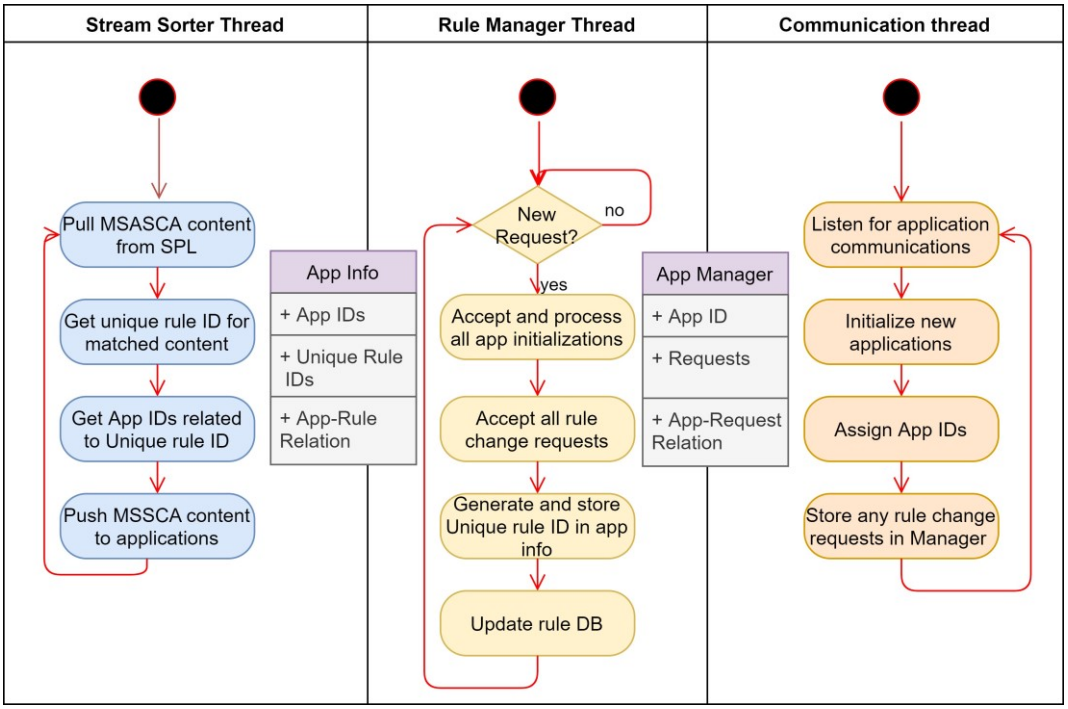


Figure 4-5 Processors of SML

4.2.3 Service container API

A Docker container environment provides service isolation, lightweight service development, and migration for applications. The service API is designed to enable easy deployment and access to network traffic with threaded UNIX socket communication. Additionally, the API automatically converts the stream content into a readable format that allows quick filtration of the SCA content. The service API provides the following main functionalities:

- Communication with SML
- Maintain signaling with the SML
- Pull stream content from ring buffers
- Support for sockets and share memory.

In a typical service deployment, the API InitClient function initializes the service by creating a UNIX socket toward the SML socket server, sending application initialization requests, and listening to SML's acceptance and completion responses in the sample application process of Figure 4-6. Subsequently, applications can add any regular expressions to the network flow by the SendRule function. The SendRule function sends the regular expressions to SML, and if the

regular expression is compatible with the Hyperscan library, SML will reply with a rule-accepted acknowledgment. Once this is received, the application can initiate data processing functions. Then, the data processing functions can access the sensor data through the SML buffers. The application program then uses API calls to access the MSACSA content for processing. The MSSCA content is associated with a structure to identify its information, such as five-tuple data. The MSSCA content structure allows services to identify network flow details and sensor details for data manipulation or communication with the sensor or cloud services through virtual interfaces. Furthermore, the API contains additional support functions to create sockets and other ring buffers for service-to-service data buffering.

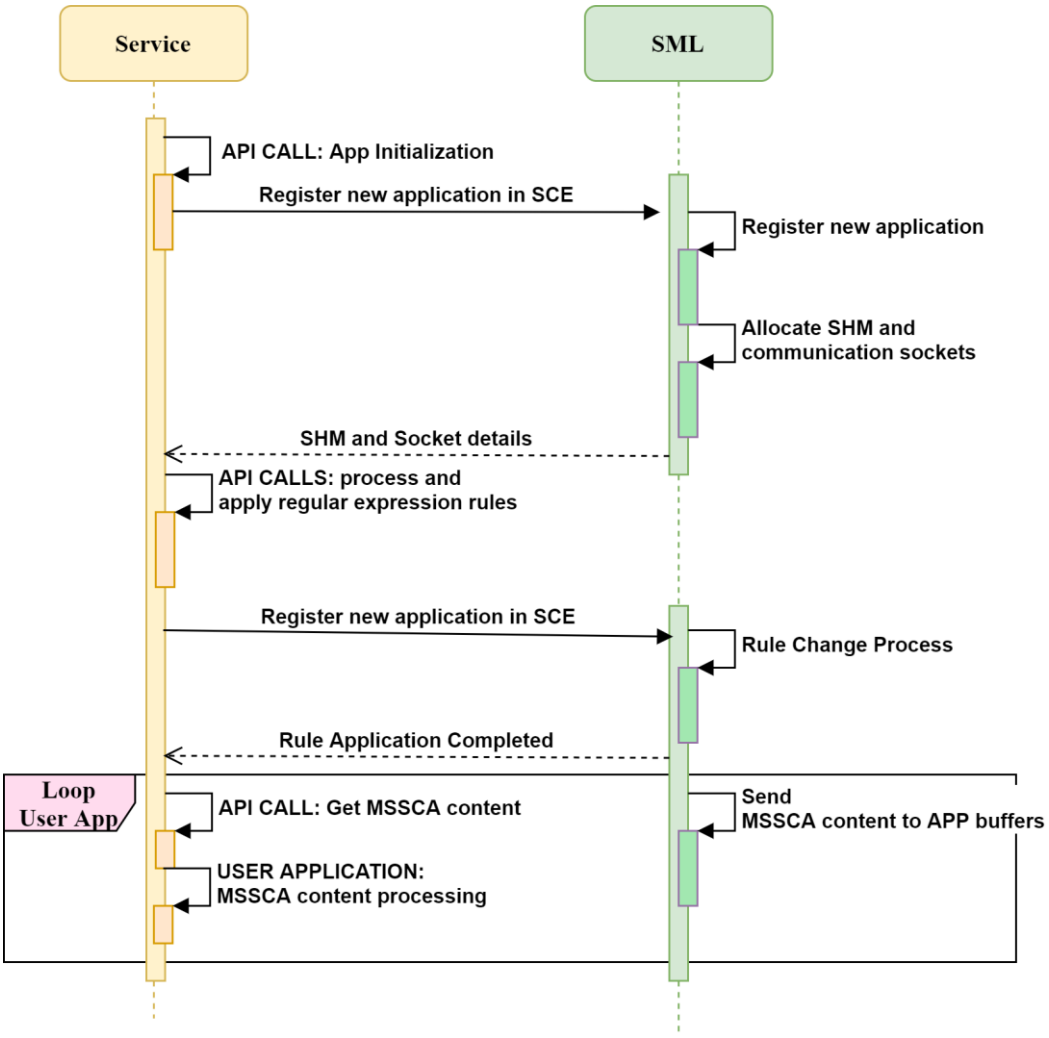


Figure 4-6 Process of a sample application

4.3 Evaluation

The SCE node was tested to confirm its scalability and limitations in supporting IoT services at 1-10 Gbps line rates. The SCE node performance was tested using a dual CPU workstation with the following configuration:

- Server Type: HPC workstation
- Processor: Xeon ES-2620 v4 2.10 GHz x2
  - Number of cores: 8
  - Number of threads: 16
- Random-access memory NUMA 1: 16 GB
- Operating System: Centos 7.2
- Kernel version: Linux v3.1

The SCE nodes performance was compared with the f-stack node, which uses DPDK for network applications in the host machine. The f-stack node provides a DPDK-based network stack for Linux servers. Additionally, the f-stack library is scalable with DPDK core assignment, similar to the SCE node. Therefore, a fstack-based web server [58] is selected to compare SCE nodes' for measuring performance under different numbers of DPDK CPU core assignments. The results of the f-stack server and SCE node are given in Table 4.1. In this test, the SCE node ran a single service to gather all HTTP traffic and classified 10 HTML tags. SCE node and f-stack require four CPU cores to achieve a 1 Gbps line rate. The f-stack node can achieve 1.43 Gbps throughput using four CPU cores, whereas the SCE node can only achieve 1.0 Gbps. The performance limitation of SCE under four core assignments is because of its architecture. In the SCE platform architecture, two cores are always assigned to NIC transmission and reception. The core assignment allows the SCE node to poll the NIC without any interruptions from the stream processors. Therefore, in four core assignments, only two cores are used for the MSSCA analysis, while the others are assigned for NIC polling. The Rx and Tx core assignments limit the performance of SCE in four core assignments. However, the results show that SCE achieved the required 1 Gbps bandwidth with only four CPU cores. Therefore, it is possible to run SCE on machines with four CPU cores to handle a line rate of 1 Gbps.

Furthermore, SCE achieves a 10 Gbps line rate using 16 CPU cores, whereas the f-stack only achieves 5.15 Gbps, illustrating the scalability of the SCE for a higher number of CPU cores. The scalability is considerably improved in SCE as it uses a modular architecture to support MSSCA. However, when compared with DPI solutions, SCE shows a limited performance than nDPI in reference [49], which provides a 10 Gbps packet capturing throughput for a single-core Xeon processor. However, nDPI does not offer any software isolation techniques such as VMs or containers that generate overhead on the nDPI performance. In addition, IoT

services require service isolation through containers or VMs to provide security for service applications. In contrast, the container-based SCE node outperforms VM-based vDPI, with its bandwidth limited to 864.77 Mbps [50] per instance. Additionally, the vDPI-based solution would require DPI processing in each VM, causing overhead because of the DPI of the same packets in multiple VMs that run different services.

Table 4-1 SCE node and f-stack throughput for different core allocations

CPU cores	Memory Allocation	F-Stack throughput	SCE throughput
4	4 GB	1.43 Gbps	1.0 Gbps
16	16 GB	5.15 Gbps	9.8 Gbps

SCE node's MSSCA reduces the data transfer for services by filtering other L7 protocols. The experimental results show the MSSCA content results in around 10% of the total traffic under testing, as packets contain unrelated data that will not match any MSSCA rules, such as XML tags. Therefore, the SPL-based architecture reduces the amount of data transfer toward the services to the percentage of sensor data in the line rate. This is an advantage over packet sharing methods, where all packets need to be routed through all services. Additionally, the MSSCA content provides L7 streams of IoT traffic directly used by smart community services.

Table 4-2 Delay of the components of the SCE node

Component	Average Latency	Maximum Latency
SPL	0.07 ms	0.09 ms
SPL to SML	0.1 ms	0.12 ms
SML to service	0.3 ms	0.5 ms
Total service latency	0.47 ms	0.71 ms

SCE node should be able to support latency-sensitive services at the SCE. Therefore, the latency of the SCE node was measured to verify the delay of each of its components. The results show that SPL stream reconstruction and Hyperscan-based string matching have a maximum delay of approximately 0.09 ms. Additionally, a 0.4 ms delay was needed to sort and transfer data from the SPL to a service. The 0.4 ms delay is caused by the MSSCA content buffer between the SPL and service. The total latency of the SCE node for a service to capture SCA content from NIC is 0.71 ms, as given in Table 4.2. Therefore, the results demonstrate the SCE

node's capability to support delay-sensitive services with latencies less than 1 ms. SCE node's latency can be further reduced by reducing the SPL to SML and SML to service buffer sizes. As an example of a smart grid service, IEC 61000-4 permits a 10 ms delay in control, with 9 ms for request transmission, reply transmission, and service application, as the SCE node only needs 1 ms time to extract and provide MSSCA content to the service.

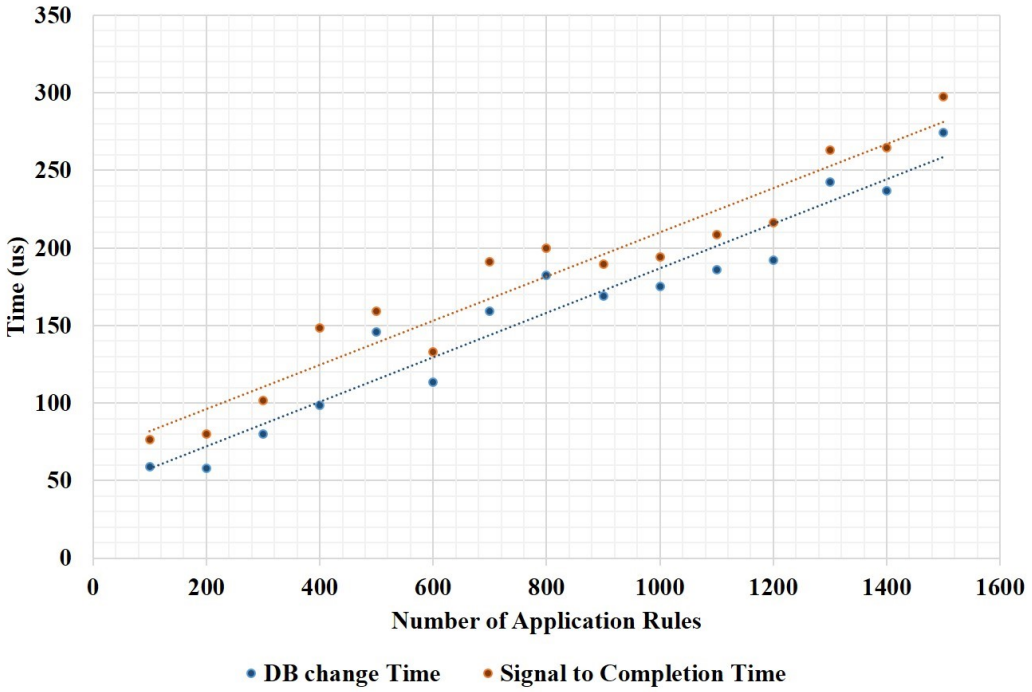


Figure 4-7 Runtime rule change time in MSSCA

The proposed distributed runtime dictionary change mechanism's performance was measured to determine its effect on the MSSCA. The performance was measured using HTTP packet filtering, where SML would change the string matching rules at runtime to produce a new rule dictionary and measure the time when the MSSCA was interrupted. The Hyperscan runtime compilation takes approximately 100 ms. Therefore, rule compilation would cause significant delays in the MSSCA. However, the proposed method compiles the rules in SML, allowing continuous MSSCA. The compiled rule database is serialized and de-serialized at the shared memory between SML and SPL.

The MSSCA analysis was interrupted only for the dictionary change in the proposed method. The interruption time in the SML and SPL to change the ruleset was measured for different rule sizes, as shown in Figure 4-7. The proposed method changes the database between 50 μs to 300 μs for a rule set size ranging from 100-1500. The time taken for dictionary change linearly increased, as expected, because of the increase in dictionary size and serialized database size. However, the actual delay of 0.3 ms is negligible, as only 0.2 Mb of the packet buffer would be

consumed for a 1 Gb link within the dictionary change process. The signaling time between SPL and SML has an additional overhead of approximately 20 μs because of the time to read and write the completion flags and other control information to and from the shared memory. The signaling time is also negligible and does not significantly affect the performance of SML. Therefore, the proposed Hyperscan distributed rule change allows multiple applications to dynamically apply regular expression rules for scanning any disturbance to the packet flow.

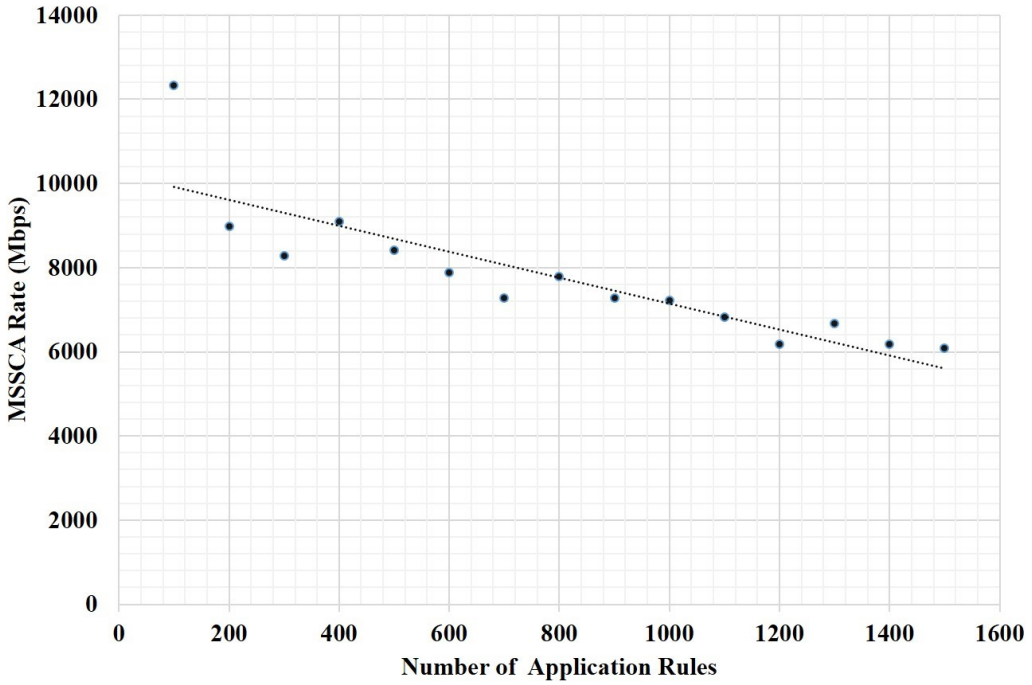


Figure 4-8 MSSCA bandwidth for a different rule set sizes

The MSSCA performance under regular expression dictionaries of different sizes was measured to confirm the limitations in the SCE node for a different number of rules. The results show that the rule set size is inversely proportional to the scanning rate, as shown in Figure 4-8 and Figure 4-9. The rule match rate increases from 0.2023 matches/kB to 0.4095 matches/kB for 100 rules and 1500 rules, or 10 to 150 regular expressions per service for ten services, respectively, as shown in Fig. 4-8. Therefore, the SCA throughput drops from 10 Gbps to 6 Gbps for a total of 1500 rules. This drop is expected because of the increase in the rule-matching rate. However, the results confirm that the SCE node can operate at 10 Gbps for ten services with ten rules per service or 100 accumulated service rules. Furthermore, the SCE node can manage the throughput of 6 Gbps for 1500 accumulated service rules.

Finally, the transfer of MSSCA content from SPL to services was tested to measure the service support capability of the SCE node. It was measured by adding a default rule dictionary to transfer all data to all SCE node services. The SCE

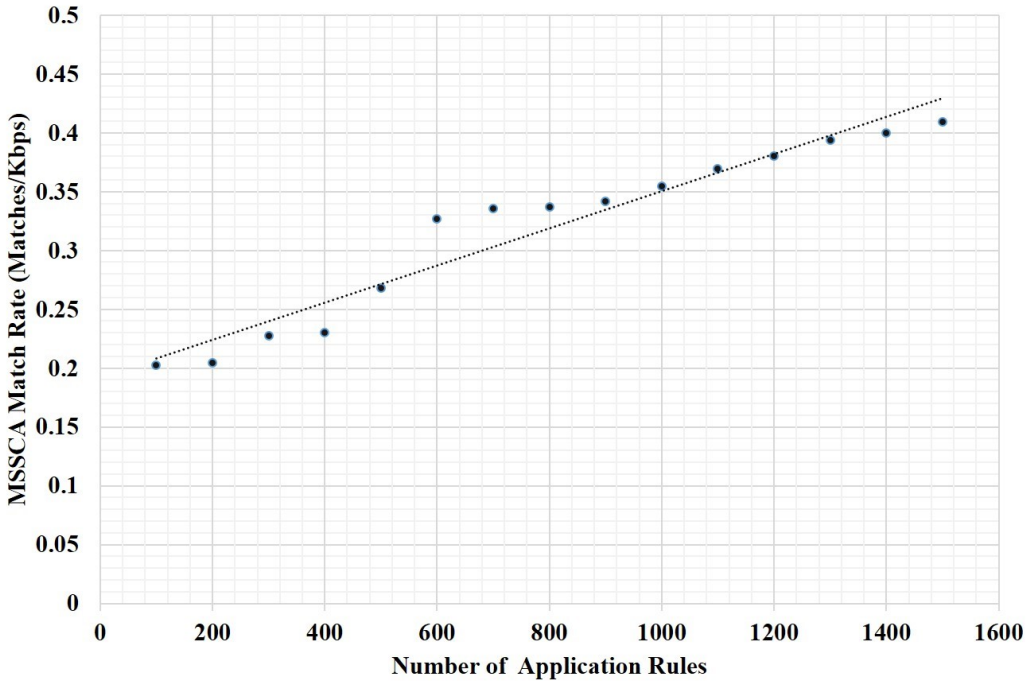


Figure 4-9 MSSCA match rate for a different rule set sizes

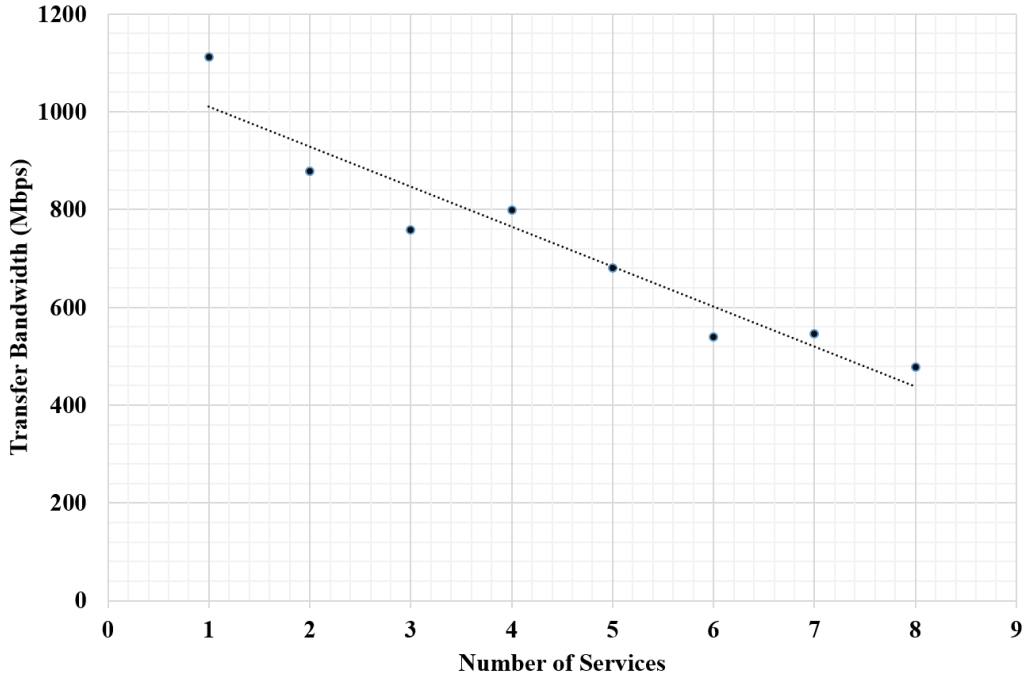


Figure 4-10 SML content sharing bandwidth for multiple applications

node's MSSCA content transfer rate is around 1.1 Gbps per service, and the data rate drops to 500 Mbps when eight services require the same data, as shown in Figure 4-10. The decline in bandwidth was caused by the eight copies generated

for the same MSSCA content. This transfer rate is adequate for MSSCA content handling because the content rate is expected to be about 10% of the line rate, as the content matched by regular expressions in the tested traffic was lower than the total traffic in the line. Moreover, the results confirm that the system can share data at the required 1 Gbps line rate when services use different MSSCA content.

4.4 Conclusion

This chapter summarized the architecture and performance of the proposed SCE node using DPDK, Hyperscan, and Docker technologies. The proposed SCE node employs MSSCA and container technology to support multiple smart community services at the edge. SCE node allowed MSSCA content to be directly transferred to services without network packet processing at the service containers. The SCE node's MSSCA achieved a throughput of 1-10 Gbps with 4-16 CPU cores in conventional hardware systems. In addition, the SCE platform proposed a distributed rule change method for the Hyperscan library to change regular expression without affecting network flow. The SCE node achieved a 10 Gbps SCA throughput for 100 accumulated rules, which allowed more than ten rules per service. In addition, the proposed dictionary change method needs less than 0.3 ms to execute and does not affect the performance of SPL network flows. SCE node supported eight similar services while providing a 500 Mbps MSSCA content bandwidth for each service, where each service can support 5000 sensors with a 100 kbps bandwidth. Additionally, the total maximum delay of the SCE node is maintained at less than 1 ms, allowing delay-sensitive services to operate at SCE nodes. Therefore, SCE nodes show adequate performance and applicability in smart community networks to support multiple smart community services at the network edge using MSSCA.

Chapter 5 Consistency guaranteed service migration

5.1 Introduction

Container migration is the process of moving a container between computers or storage devices. Migration technology has been developed to realize flexible services and to distribute services in the cloud dynamically. In live stateful container migration, packet processing services should reduce the downtime for low latency services. Therefore, migration time should be reduced to minimize the overall downtime of application services. Some studies intend to decrease migration time by reducing bandwidth usage [70], [71]. They reduce bandwidth usage by separating the container layer's image layer and container layer and transferring only the container layer toward to destination node. To resolve these in smart community services, we need a migration method that considers throughput and downtime while supporting latency reduction and data consistency. It is assumed that downtime below 10s is acceptable for container live migration at fog[91].

The currently available migration methods cannot support multiple services on a single node because it cannot detect the specific data flow. Furthermore, since VNF migration methods change the network flow, the other services run on the same node are affected by modifying the network flow. Therefore, a multi-service supported migration solution is required to support smart community services while minimizing the buffered data. Fog nodes such as SCE nodes could dynamically distribute services to multiple nodes to accommodate network loads. The migration of service containers is required to achieve dynamic reconfiguration in smart community networks. The available container migration techniques focused on resource utilization and bandwidth management[70], [71]. However, SCE should support consistency guaranteed one to N migration for containerized network services.

A novel migration method must support SC service migration and provide data consistency, network flow preservation, and one-to-many migration to resolve the above issues. This chapter proposes multiple migration methods for the above scenarios on SCE platforms. The proposed stateful migration does not modify the existing network flow or affecting other services. Moreover, in hierarchical smart community networks, one-to-many migration is supported to handle overloading conditions. The proposed adaptive migration method consists of the following migration methods.

- Consistency Guaranteed Migration (CGM)
- One to N Consistency Guaranteed Migration (O2NCGM)

CGM and O2NCGM use Layer Leveraging Migration (LLM), a method designed to minimize the migration time and network bandwidth usage for container migration by reducing image layers transmitted to destination nodes. The LLM minimizes the downtime by separating the container and image layers. The CGM was designed to guarantee network consistency by communicating and buffering network flows between a source node and destination nodes using the SCE service management layer. Through CGM, and O2NCGM is created to support a multi-client or one-to-many consistency guaranteed migration. The contributions of the section are the following.

- Provides a container migration technique that reduces migration time and guarantees service consistency without affection for other services in a node.
- One-to-many migration method to handle dynamic loads The proposed container migration method offers migration without modifying the existing network flows.

These migration methodologies expect to achieve guaranteed service consistency without affecting network flows. CGM strategy guarantees network consistency. O2NCGM provides one-to-many migration with all the above features. The strategies are separated to provide easy applicability in different network scenarios.

5.2 Implementation of consistency guaranteed migration

SCE platform and Docker migration architecture are shown in Figure 5-1. The system consists of four main components: SPL, SML, Docker daemon interface (DDI), and APP service API. IoT sensor information and data tags are used in the scanning process to differentiate the data. The matched streams are transferred to the service buffer. This process prevents any changes to network flows even under migration as the migration process uses the service buffer to manage the network consistency. SML manages the service containers in a single host and executes commands such as container deployment through sockets. SML manages the state information of services and the state of traffic flow toward the service. DDI contains an API to control images and containers in a node. DDI RESTful API allows administrators to deploy, terminate, or migrate the services using SML.

In the prototype implementation, SPL matches any stream with a rule such as “XML,” “<title,” and “measurement.” SPL captures any associate streams with these keywords and transfers them into the service buffers. The captured streams contain five-tuple information, layer four-stream information, and time stamp data to identify the flow information. The service buffers use a shared memory structure to communicate between the user space and the network flow. Service buffers are similar to network packet buffers that exist in NFV infrastructure nodes. The proposed approach such system containers to migrate with shared structures with the host machine while keeping the network consistency.

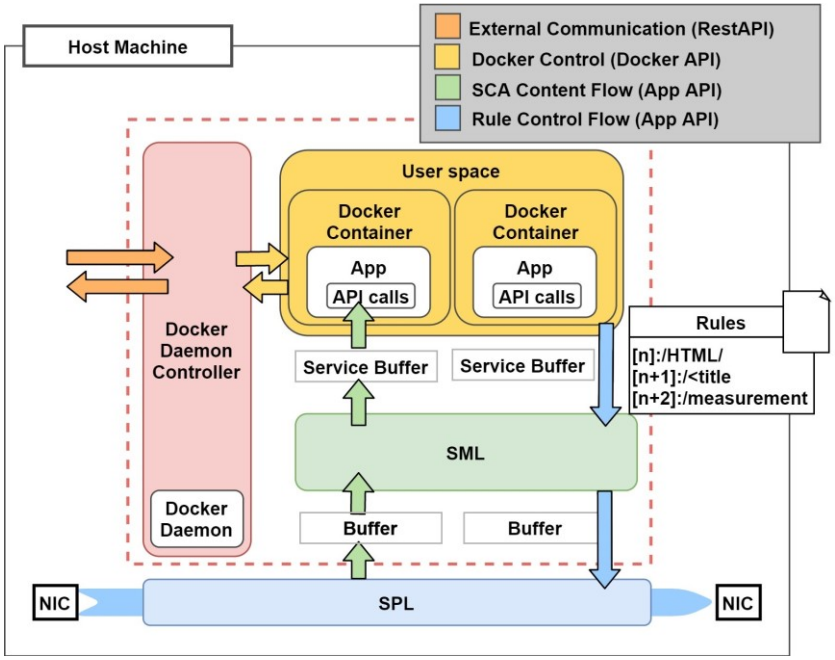


Figure 5-1 Architecture of SCE based Docker migration

LLM minimizes data transfer between the migrating of container image layers in source and destination nodes. Figure 5-2 shows the migration flow of LLM. LLM consists of two steps: layer ID remapping and container layer extraction. The layer ID remapping is used to reduce the migration time of the live migration. Additionally, container images are already distributed to target nodes through the Docker registry to quickly execute container layer extraction without image transfer between the source and destination nodes.

In the Docker migration process, the container should send the container image to the destination node to restore it in the last state. The Docker containers are unable to restore checkpoints on a different host with only container image information. If the container restores in a different host, it causes a mismatch of layer IDs, thus failing the destination node's restoration process. The layer ID is a unique local ID of containers with different states of layers in their host machine. In the run time, a cache ID is used instead of a layer ID. These two ids are generated through SHA256, and Docker daemon use cache ID in the restoration process. Therefore, this ID should be similar in the destination node to restore the container. In this method, the Docker container cache IDs are remapped with layer IDs in the source. The remapped layer ID can then be directly used to check the image and layer availability in the target node for the migrating Docker container. The overlay2 storage requires relinking the symbolic links generated using the new layer ID. Finally, the Docker daemon is restarted in the source node to update this information in the migrating container. Before migration, the remapped cache IDs are updated in the target Docker node allowing correct restoration of the container file system after migration.

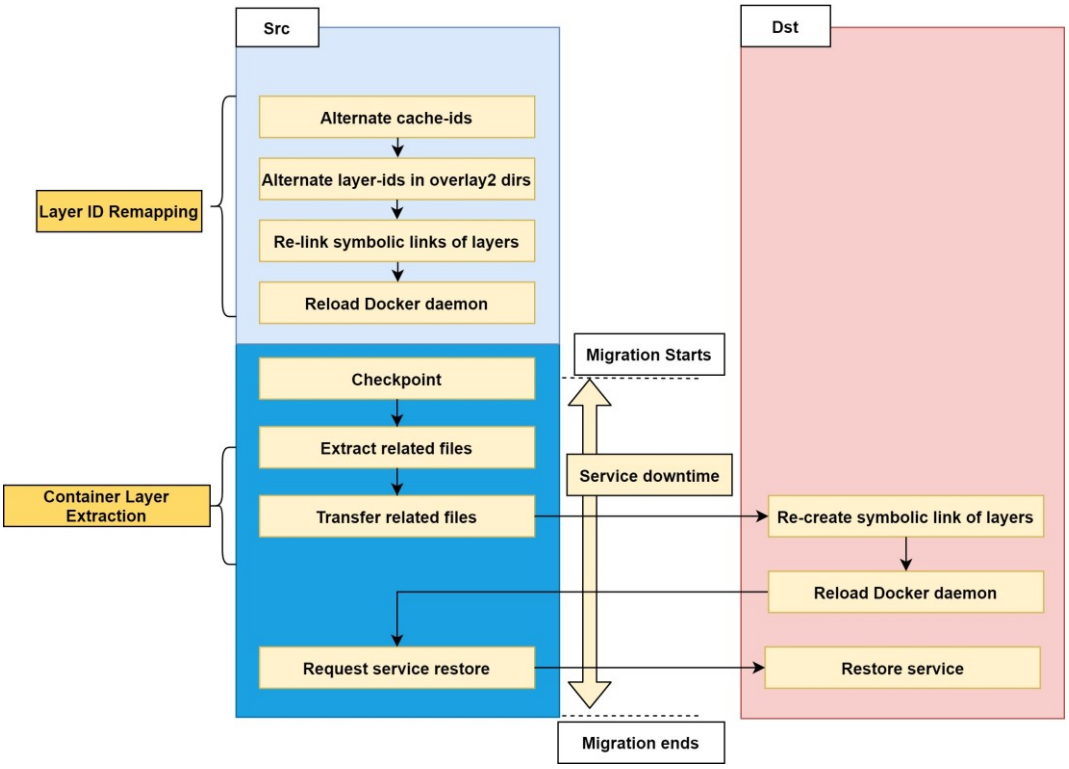


Figure 5-2 Process of LLM migration

Once the layer ID remapping is completed, the Docker host could start the migration process. The process starts with checkpoint the live container. Docker checkpoint freezes the state of the container state. However, the checkpoint container cannot directly migrate without the container configuration files, layer files, file mount information, and directories containing volumes. Docker has different directories about the image layer and the container layer. LLM copies container running state, layer, mount information, and volume files to the destination node, then relinks the files to the correct location using layer ID mappings. Afterward, the destination node will restore the container using already available container images and layers using the relinked symbolic links. In conventional migration, source and destination should send container image and all the layers for migration, causing high overhead on the source-destination link. This method reduces the bandwidth consumption, downtime, and migration time by minimizing the data transaction between the source and destination nodes.

5.2.1 Consistency guaranteed migration

The smart community services should handle network stream consistency under the migration. Therefore, Docker migration should handle shared MSSCA content and service states to migrate and process data consistently. The packets are identified on the SML buffers using stream identifiers. In the proposed method, packets already passed the source and still has not arrived at the destination should be stored and handled to guarantee consistency, as shown in Figure 5-3. Figure 5-3 example shows i number of packets upstream from source to destination and j number of packets downstream from destination to source. In CGM, sources and destination nodes identify the read and write offset values of services buffers using the SML and APP information.

Table 5-1 Nomenclature for consistency guaranteed migration

Symbol	Description
B_{src}	App. Service buffer at src
B_{dst}	App. Service buffer at dest
$P_{dir}(i)$	Network packet i in a direction (s2d: src-dest, d2s:dest-src)
q_{dir}	Destination buffer packet (s2d: src-dest, d2s:dest-src)
x	Buffer upstream offset at src
y	Buffer downstream offset at src

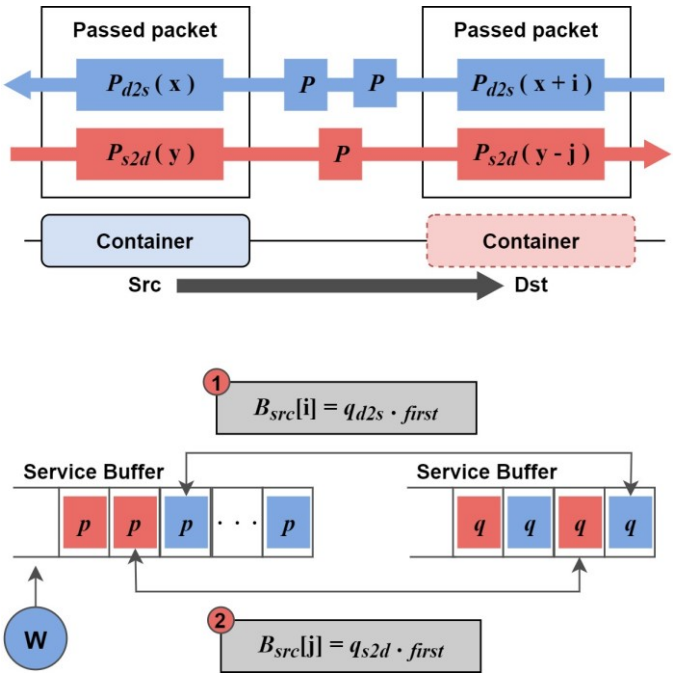


Figure 5-3 Buffering of traffic before migration

Additionally, SML creates temporary buffers to store the streams before the migration. This preserves the consistency of streams and avoids packet loss. Furthermore, the layered design of SCE continuously forwards the network flows

while buffering packets in SML for services in temporary buffers. Network stream unique identifiers and saved offset values are used by the migrated services when selecting the buffers' initial offset values.

Figure 5-4 shows the complete migration flow of CGM. The CGM flow contains additional steps to guarantee consistency. CGM initially gather state metadata of service applications to create temporary service buffers in the destination node. Afterward, the destination SCE applies the stream capturing rules in the destination node and stores the SML buffers' network streams until the service is migrated and initialized at the destination node. SML removes temporary service buffers and application rules once the buffer is synchronized in source to destination and destination to source flows. Afterward, LLM migration starts as the service at the source node stops accessing the IoT data buffers. After the LLM migration, the migrated Docker service will use the state information to remove duplicate IoT data and restore the correct offset values using IP flow information. In comparison to the VNF migration SCE platform reduce the network flows that buffers under migration by service-based flow identification. Furthermore, container layer separation reduces the migration bandwidth further by reducing the downtime under migration.

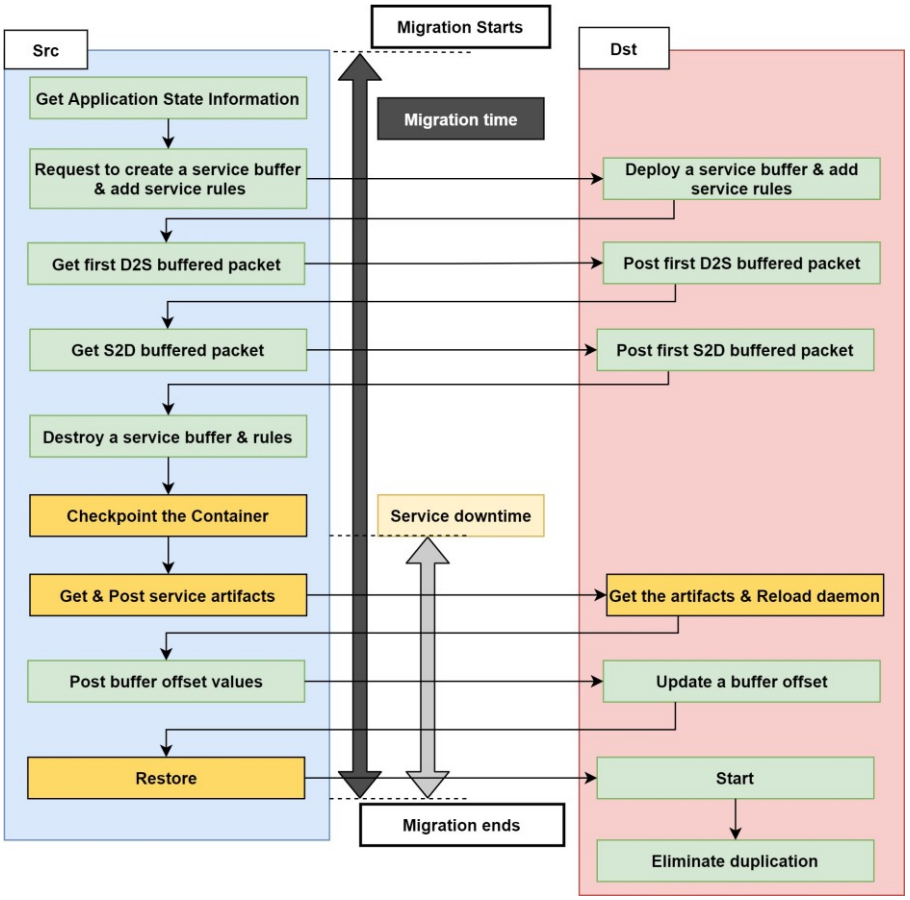


Figure 5-4 Process of CGM migration

5.2.2 One to N consistency guaranteed migration

O2NCGM extends CGM allowing migration from one to multiple hosts. In O2NCGM. The proposed method first uses multiple threads to gather state information of applications and IoT data buffers through SML similar to CGM. Then SCE SML creates IoT data buffers on all destination nodes and stores new packets on the same IP flows on the temporary buffers similar to CGM. Afterward, the system sends the buffered traffic to all the target nodes. The target nodes remove the duplicate traffic using their local offset values and IP traffic flow details. Finally, the system initialized the migrated containers through LLM using multiple socket connections.

5.3 Evaluation

The proposed CGM and O2NCGM methods were evaluated using three SCE nodes with the specification given under Table 5-2.

Table 5-2 The amount of migration data of containers

(KB)	Methods	Total	Image	Container			
				FS	Checkpoint	Mount	Others
Busybox	LLM	203	1,160	0.14	195	0.20	8.57
	Conservative	1,357	1,160		193		0.00
Elasticsearch	LLM	2,357,931	486,000	33.72	2,357,886	0.20	10.65
	Conservative	2,847,739	486,000		2,361,739		0.00
Original App	LLM	618	1,070,000	1.24	311	305.50	0.20
	Conservative	1,070,311	1,070,000		311		0.00

The network consists of two edge nodes and one cloud node connected in a hub-spoke architecture. First, the effects of the LLM method is evaluated for migration time and service downtime. LLM and conventional migration resource usage were evaluated using three different sizes, as shown in Table 5-3. The busy box container is a lightweight service container, while the elastic search and original application are extensive memory-intensive services. The original application reads data of IoT temperature sensors in the network. Conventional migration methods consume additional data in migrating due to container image layer migration. LLM reduces this data usage by only transferring the container layer and memory data between the target nodes. Therefore, LLM can be used to reduce the overall data transaction in container migration.

Migration time and service downtime under the different bandwidths were

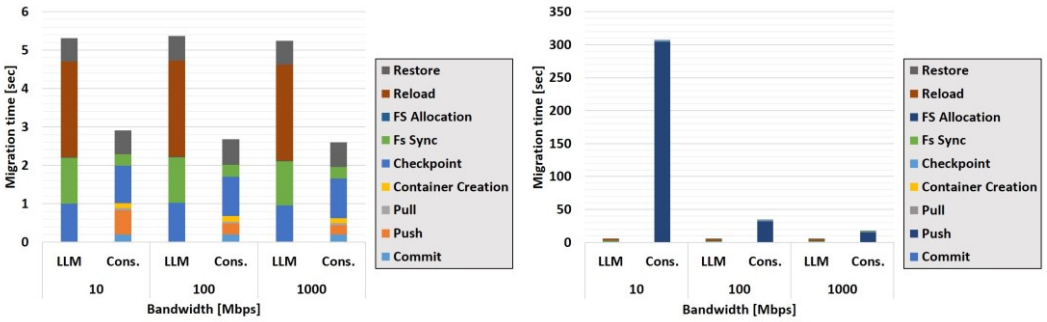
Table 5-3 Details of the evaluation environment of container migration

Sensor App	Intel NUC
CPU	Intel® Core™ i3-6100U CPU @ 2.30 GHz
Total memory	8GB
OS	Ubuntu 16.04.4 LTS
Intermediate node	Shuttle DH310
CPU	Intel® Core™ i7-8700U CPU @ 3.20 GHz
Total memory	32GB
OS	Ubuntu 18.04.1 LTS
Docker	17.09.1-ce
Criu	3.7

evaluated to compare LLM and conventional methods' performance. Link bandwidth was limited to 10, 100, 1000 Mbps, respectively, and the migration performance was evaluated.

Figure 5-5(a) shows the comparison of the migration time of Busy-box. LLM has a 1.8s longer migration time than the conventional method, and service downtime was increased by 2.62. The downtime is caused by the smaller size of the container image where the LLM container isolation and reload process cause more overhead than the amount of data reduced by the layered migration. Layer migration shows that the LLM causes additional overhead for shared buffer isolation in a container for small image sizes. However, LLM provides superior performance in the case of large containers such as elastic search Docker containers.

Figure 5-5(b) shows the comparison of the migration time of Elasticsearch. LLM's migration time was 10.8% or around 200s lower than the conventional method.



(a) Busy Box (b) Original data collection application

Figure 5-5 LLM migration results for BusyBox and original application

The migration time of LLM was 5.61 sec on average, while conventional methods were 307sec (10 Mbps), 34.7 sec (100 Mbps), and 17.7 sec (1000 Mbps). In the LLM

method, the original network application only migrates the packet buffers and the packet processing program compared to the large container image with all programming libraries. Therefore, LLM can considerably reduce the migration time compared to conventional methods when container applications contain more than 400MB images.

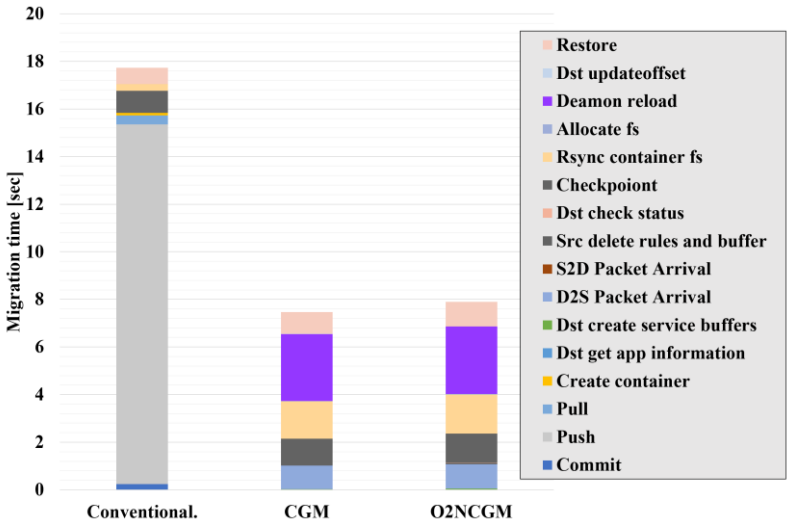


Figure 5-6 Comparison of conventional, CGM, and O2NCGM migration

Finally, all CGM and O2CGM methods were compared with conventional methods at 1Gbps, as shown in Figure 5-6. CGM and O2NCGM reduce service downtime compared to the conventional method. The proposed method reduces migration time using layer migration, where conventional migration pushes and pulls a full container image between the target nodes. However, the proposed migration methods incur overhead in container reloading due to the LLM's complex reload process. Furthermore, the proposed CGM and O2CGM increase migration time due to data synchronization. In addition, O2CGM migrates a single service to two nodes causing additional overhead in data transactions and buffer synchronization. However, the proposed methods provide better migration time compared to conventional container migration. Additionally, the proposed CGM and O2NCGM migration achieve zero data loss without affecting the network flows.

5.4 Conclusion

This chapter proposed the CGM and O2NCGM migration that supports consistency guaranteed multi-container migration for smart community services. The proposed migration method handled network consistency without affecting the network flows. Container layer separation was used to reduce data transactions between the nodes in container image layers. Furthermore, CGM was applied with LLM to achieve zero packet loss while reducing the network downtime by more than 10% compared to

conventional methods. The CGM migration used container layer separation and service-based flow identification to reduce the data transfer between the source and destination SCE platforms. The overall results show that the CGM and O2CGM reduce the migration time by more than 10% for containers with image size higher than 400MBs. Additionally, O2CGM provides consistency guaranteed one to N migration to support service distribution under different network loads. Through CGM and O2NCGM, the SCE platform improved its applicability to support smart community services.

Chapter 6 Computational delay aware service function chaining

6.1 Introduction

SFC has shown an increased interest due to mobile edge computing [99] and fog computing [10], where computing resources are placed closer to the network edge to improve network services' performance and efficiency. SFC classifiers differentiate the traffic based on requested services and other predefined rules. The SF instance selection for a particular traffic flow can be complicated because there can be multiple SFs in the network due to the reliability, locality, and load distribution of smart community networks. The SFF is used to forward these packets to the next SF through the network according to the encapsulated SFP data. The SF proxy is used when VNFs/SFs do not understand the SFC header; here, the proxy handles the SFC header data, forwards the packet to the SF for task completion, and applies processes to the SFC header once the packet returns from the SF instance. SFC traffic is steered using a single classifier and per-hop classifier techniques. One classifier is used in a single classifier to steer the traffic through the SFC using special headers. The per-hop classifier steers the traffic per SF [92]. The SCE service chaining operates as a data plane SF forwarding rather than management plane orchestration and classification functionalities.

SCE nodes should manage the distributed computing resources to provide optimal performance using services run as VNFs, considering the sensor traffic generated through terminal devices. SCE service VNFs are distributed in multiple data centers or SCE nodes to create efficient network performance considering the latency, cost, and network locality [93]. SCE service VNFs can be distributed among available SCE nodes to create a dynamic system with distributed service nodes.

The distribution of multiple VNFs in multiple fog nodes or data centers causes complexity in creating SFCs. Although a single data center is usually homogenous, an environment with multiple data centers and fog nodes create a heterogeneous computation resource with different data centers and fog nodes having hardware resources with different capabilities. Therefore, simple selection such as round-robin or random selection can lead to the overloading of some VNFs instances lacking proper computational capabilities. These overloads can cause service level agreement (SLA) violations and lead to the poor performance of VNFs. Additionally, the distribution of SFs requests without considering the network infrastructure can lead to unnecessary overheads on the network traffic

where VNFs further away from the source path get selected for the SFCs. With the rapid growth of IP traffic over networks, overhead network traffic has become an essential factor. Furthermore, this additional traffic and NFVI techniques require proper management of hardware resources to provide reliable services. An optimal SFC selection algorithm that provides reliable services.

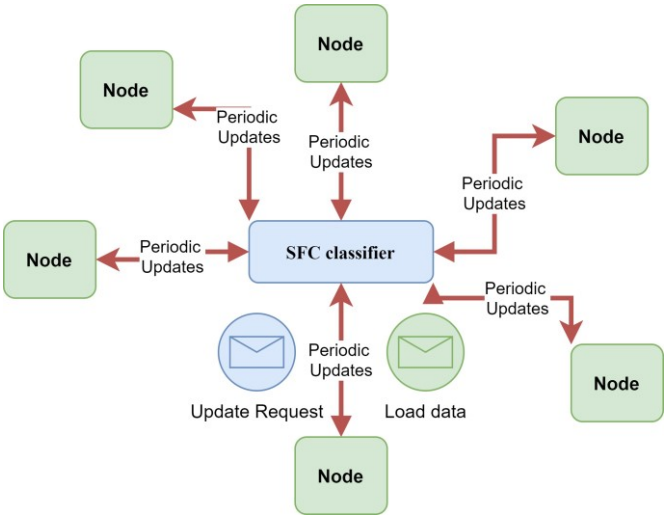


Figure 6-1 SFC SFP periodic update collection

In fog and multi-data center environments such as smart communities, computation resources are distributed among datacenters or SCE nodes. The data centers are usually separated by long distances and create network delays when the traffic transfers between the data centers. In contrast, SCE nodes have lower network delay in between each other. Therefore, an SFP selection can cause traffic to be transferred through one of the VNF instances, causing intra-node or inter-node traffic. Traffic flow depends on the SFP creation algorithm. This section considers an equal VNF distribution among all the SCE nodes for performance measurement of the proposed algorithm without considering dynamic management and orchestration of VNF instances. The SFC algorithm distributes the requests to different VNFs by finding the best SFP.

The proposed algorithm uses VNF instances' computation capabilities, load, and network delay for SFC's optimal execution. It uses periodic proactive data gathering, as shown in Figure 6-1. The systematic data collection reduces the end-to-end delay introduced by a completely reactive VNF data gathering algorithm such as the OPS [78]. Additionally, such an algorithm improves the performance compared to a completely passive algorithm such as the round-robin algorithm using the computational capability data of the VNF instances. The algorithm was tested in the CloudSim [94] simulation environment and compared with the Nearly-optimal service-function path algorithm (NSP) [77] and the optimal path selection algorithm (OPS) [78].

The contributions of this chapter can be identified as follows:

- Provide an implementation of a novel SFP selection process for SCE
- Evaluate and compare SFP creation methods in cloudsim environment

6.2 Implementation of optimized computational delay aware service function chaining

The convention of symbols used in this chapter is described in Table 6-1. The cloud environment contains the set of nodes: Y. Each SCE node $y \in Y$ is considered to be situated at a known distance from each other with a static average network delay. Each SCE node would likely consider having a complete fiber network inside it; therefore, the inter container delay is considered zero. In SCE nodes, inter VNF is zero as it would be a host on a single machine. The nodes support instances of different SFs. Each SF $x \in X$ has at least one or multiple instances of VNFs in each fog nodes y, known as $S_{xy,j}$. Each instance is executed on a different container, and the VNF has an allocated computation rate of $C_{xy,j}$. The total calculation capacity of a node for an SF x is given as C_{xy}.

Similarly, the load values of a VNF instance and fog nodes for SF x is given as $L_{xy,j}$ and L_{xy}, respectively. This study assumes that each node y contains homogenous VNF instances of SF_x. Different nodes allocate heterogeneous resources for VNF instances of SF_x.

The proposed algorithm operates as a request manager/broker in the CloudSim environment. The cloud manager receives the SFC requests with different SF lengths and SFs, such as $SF_1 \rightarrow SF_3 \rightarrow SF_2$. The cloud manager contains the network delay within itself and between the nodes. Additionally, the cloud manager periodically requests computation rates from node managers for different SF types in X. The cloud manager requests computation rate data from all the known nodes $y \in Y$ for each SF $x \in X$. This data includes the total allocated computation rate of a node for SF x and a list of VNFs, and their individually allocated computation rates in ascending order. After the initialization, the cloud manager periodically requests the computation rates to check for any SCE node changes.

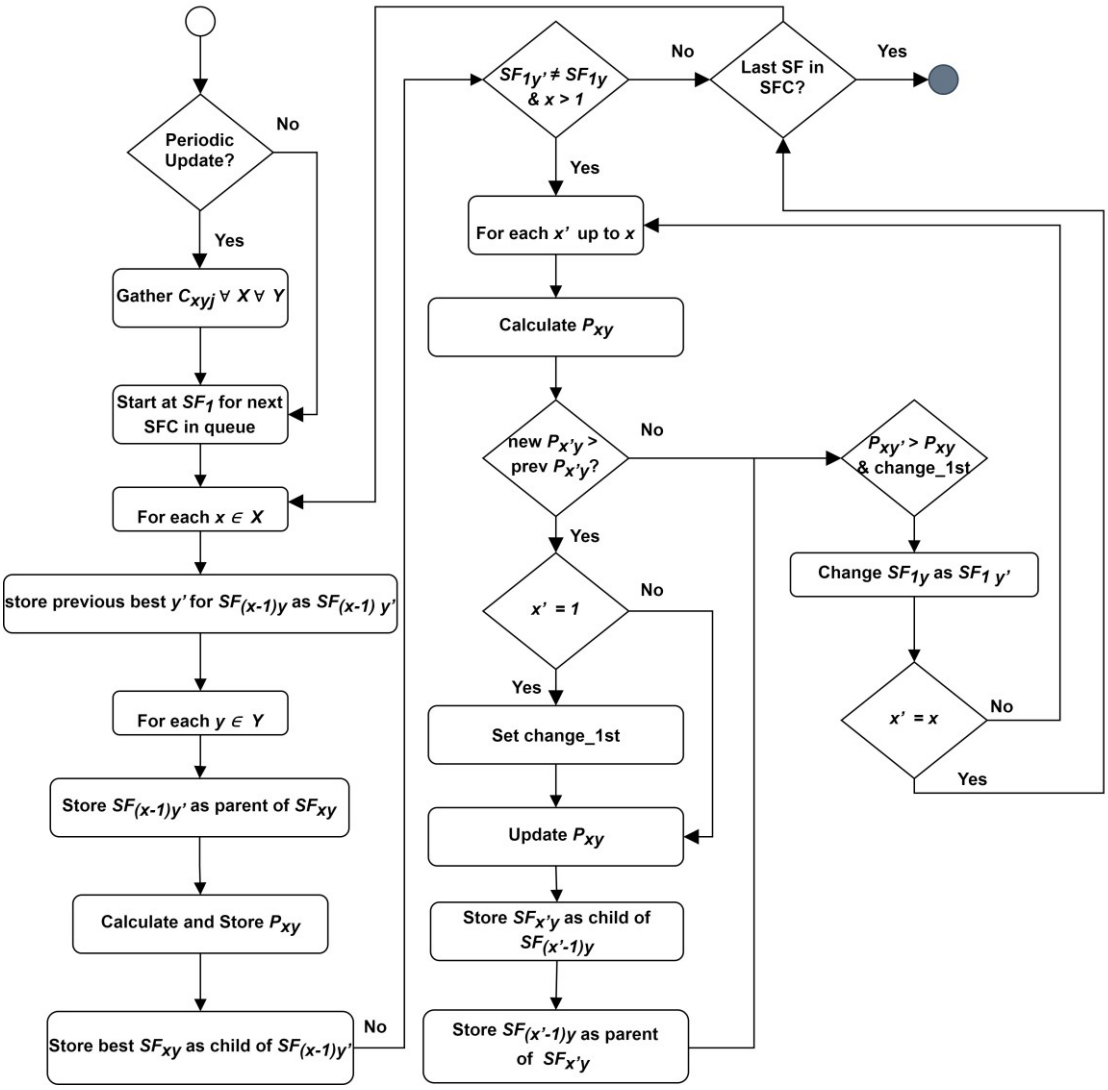


Figure 6-2 Proposed SFP process

Table 6-1 Nomenclature of SFC

Symbol	Description
X	Set of SFs in SFC
Y	Set of nodes in the Environment
I_x	Instruction Length of SF x
S_{xy}	Set of SF abstracts of $x \in X$ in node $y \in Y$
S_{xyj}	SF instance j of type $x \in X$ in node y
C_{xy}	Computation rate of node y for all SF_x instances
C_{xyj}	Computation rate of SF instance SF_{xyj}
L_{xy}	Load of node y for all SF_x
L_{xyj}	A load of SF instance SF_{xyj}
$T_{xx'}$	Communication Delay in between SF_x and $SF_{x'}$
$T_{yy'}$	Communication Delay in between node y and y'
R_{xy}	Approximate task completion rate of node y for SF_x
R_{xyj}	Approximate task completion rate of SF instance SF_{xyj}
P_{xy}	Path cost for SF_x if node y selected for SF_x

The proposed algorithm uses a computation rate based number of requests to calculate the approximate task completion rate for a particular node and for VNFs using Equation 6-1 and Equation 6-2. The load is updated when a new SFP calculation is carried out.

$$L_{xyj} = \begin{cases} L_{xyj} - (\Delta T)C_{xyj}, & L_{xyj} - (\Delta T)C_{xyj} > 0 \\ 0, & L_{xyj} - (\Delta T)C_{xyj} \leq 0 \end{cases} \quad \text{Equation 6-1}$$

$$R_{xy} = \frac{(L_{xy}+1)}{C_{xy}}, \quad R_{xyj} = \frac{(L_{xyj}+1)}{C_{xyj}} \quad \text{Equation 6-2}$$

The load values are stored locally in the cloud manager. R_{xyj} is set to *MaxDouble* when a node does not support SF_x. The approximate task completion rate is used to load balance and approximate the SFC completion time.

The cloud manager calculates the network delay between two SFs using network delay between the nodes, as shown in Equation 6-3. The cloud manager ignores the delay within a node because it is negligible compared to the nodes' delay.

$$T_{xx'} = \begin{cases} T_{yy'} & if\ y \neq y' \\ 0 & if\ y = y' \end{cases}$$ Equation 6-3

The network delay is created using BRITE in the CloudSim environment. SFP algorithm contains a path cost value for all possible combinations of the SCE nodes and data centers as given in Equation 6-4.

$$P_{xy} = P_{(x-1)y'} + T_{xx'} + R_{xy}$$ Equation 6-4

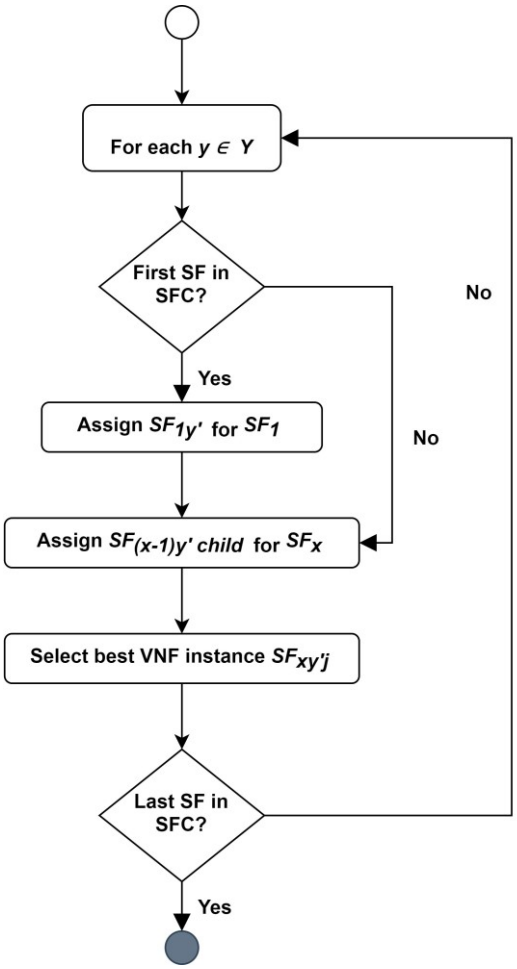


Figure 6-3 Process of VNF allocation to SFs

The cloud manager creates an empty array for each of the SFs and nodes possible combinations. Additionally, these elements manage the parent-child relationship to find the path through the SFC. SFP algorithm creates a path from the cloud manager to the end of SFC using the best child node for each SF, as shown in Figure 6-3.

The proposed algorithm starts with the first SF in the chain and calculates P1y ∈ Y. Further, it selects the lowest P1y node y for the SF1 task and assigns

that node to the first node variable, and further assigns the cloud manager as its parent. Further, the algorithm moves to the next SF_x and calculates the cost using the best y selected for $SF_{x-1,}$ and assigns $SF_{(x-1)y}$ as the parent of all the SF_x node combinations. Once the lowest cost node y' is selected, the algorithm assigns $SF_{xy'}$ as the child of $SF_{(x-1)y}$. If the new best node y' is different from the old selected node y, the SFP selection algorithm starts to calculate new costs from $P_{1y'}$ to $P_{xy'}$ assuming that all the SFs up to $SF_{xy'}$ are allocated to node y', and updates y' costs if the new cost values are lower than the old path cost values. The proposed algorithm would change the first node variable to y' if the new node y' based cost is lower than the $P_{(x-1)y}$ based cost. Once the SFP algorithm reaches the last SF in SFC, it assigns the path by retiring child values starting from the first node, as shown in Figure 6-3.

6.3 Evaluation

The algorithm performance was measured in the CloudSim simulation environment. The simulation consists of four SCE nodes with network delays, as shown in Figure 6-4. Each node contains a single host machine that runs VMs for different VNF types. CloudSim environment was modified to handle SFC and deploy SFCs/cloudlets at given delays to create consistent load toward the cloudlet manager. Inter-node SFC transfer methods were implemented such that the network delay would be added to such operations. The simulation was carried out, and the proposed algorithm's performance was measured compared to NSP and OPS algorithms.

Three types of SFCs were generated, and 1000 SFC instances of three types of SFCs were sent to the cloudlet manager for SFP allocation. The different SFC types are listed in Table 6-2. The nodes contain four different types of VNFs to support each SF type. CloudSim environment was modified to implement VNFs and deploy them according to the simulation environment. The network delay between the nodes was defined using BRITE network topology links.

Table 6-2 SFC chains used in the simulation

SFC Type	SFC chain
1	$SF_1 \rightarrow SF_2 \rightarrow SF_3$
2	$SF_1 \rightarrow SF_2 \rightarrow SF_3 \rightarrow SF_4$
3	$SF_1 \rightarrow SF_2$

The end-to-end delay for different SFCs was measured in the simulation environment, as shown in Figure 6-5. The proposed algorithm exhibits better performance compared to NSP and OPS algorithms. The proposed algorithm performs better than NSP because NSP only uses local load values and selects the VNF with the lowest load values without considering the VNF computation rates. The

proposed algorithm performs better than OPS because OPS reactive data collection incurs additional network overhead on each SFP selection, causing an extra delay in the SFP creation, even though the algorithm creates an optimal path. The proposed algorithm exhibits better performance because it improves the performance using proactive computation rate usage while minimizing any delay caused by the reactive live data collection.

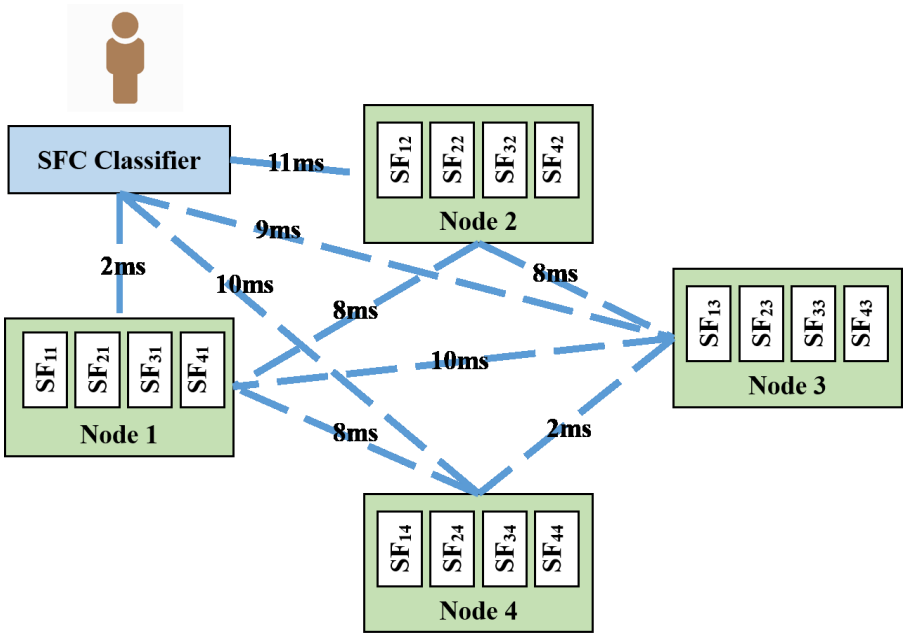


Figure 6-4 Simulation environment of SFC distribution

The computational overhead generated by OPS reactive data collection can be seen in Figure 6-5, which shows the computation time of different algorithms. NSP and the proposed algorithm can compute the SFP path relatively quickly than the OPS algorithm because OPS live reactive data collection causes a significant network delay in the calculation. Additionally, the OPS type method can create network overhead with reactive data collection traffic when there is a considerably large SFC traffic. Finally, the proposed algorithm was tested on a large scale model with 100 SCE nodes in a full mesh environment. The calculation time and SFC end to end delay are shown in Figures 6-5 and 6-6. The NSP algorithm calculation outperforms the proposed algorithm. However, the calculation time is negligible compare to the end-to-end delay of the network. Therefore, the proposed algorithm improves end-to-end SFC delay, even with the increased number of SCE nodes.

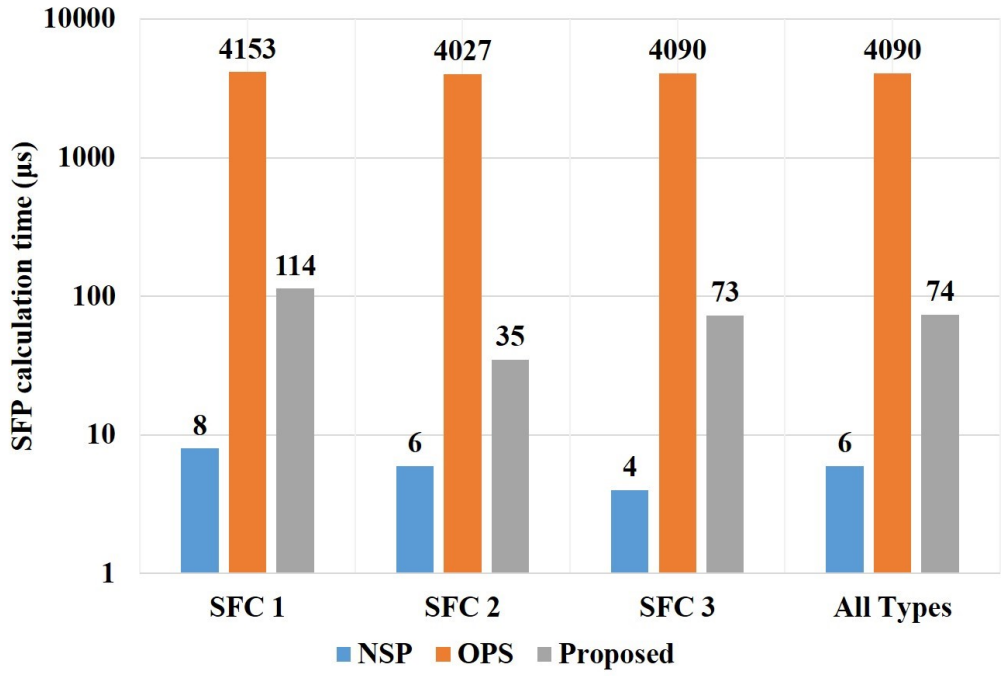


Figure 6-5 SFP calculation time using logarithmic scale

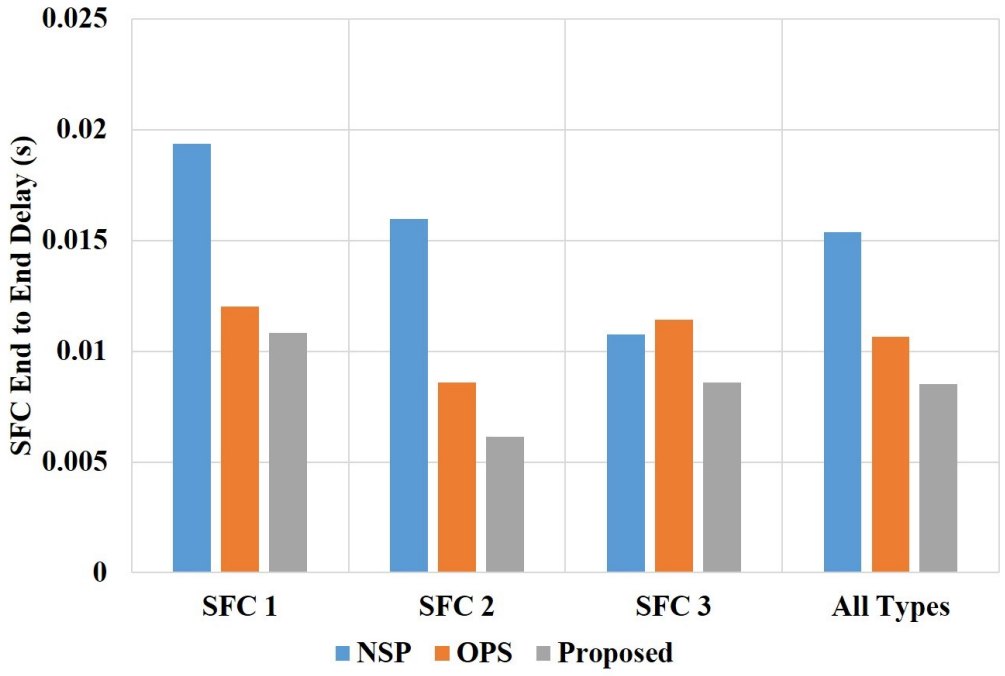


Figure 6-6 SFC end-to-end delay

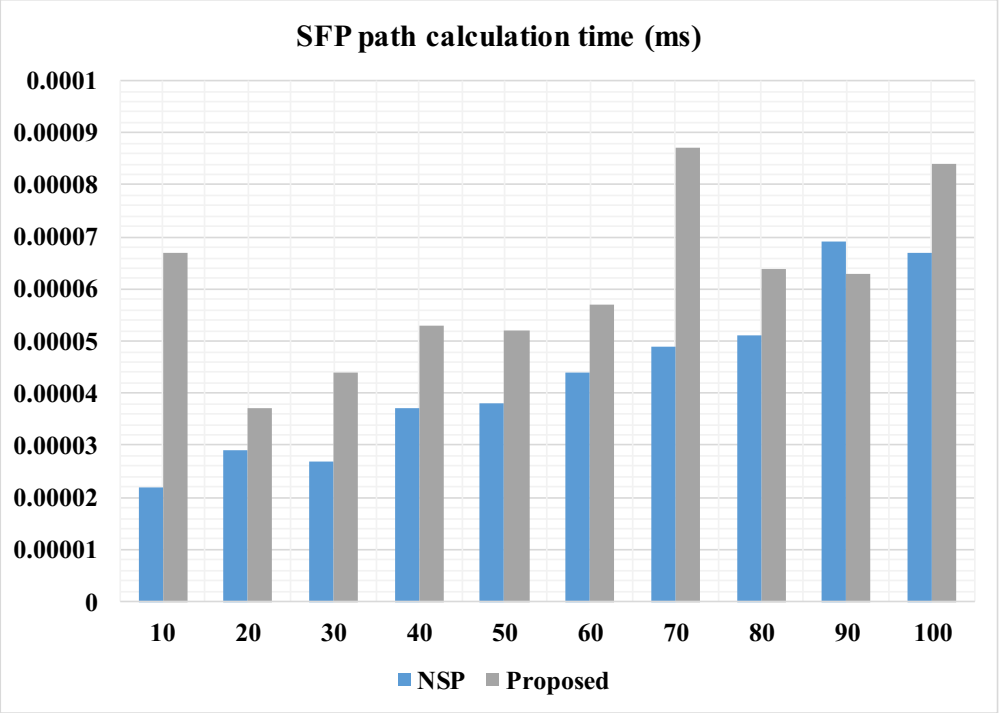


Figure 6-7 SFP calculation time against the number of nodes

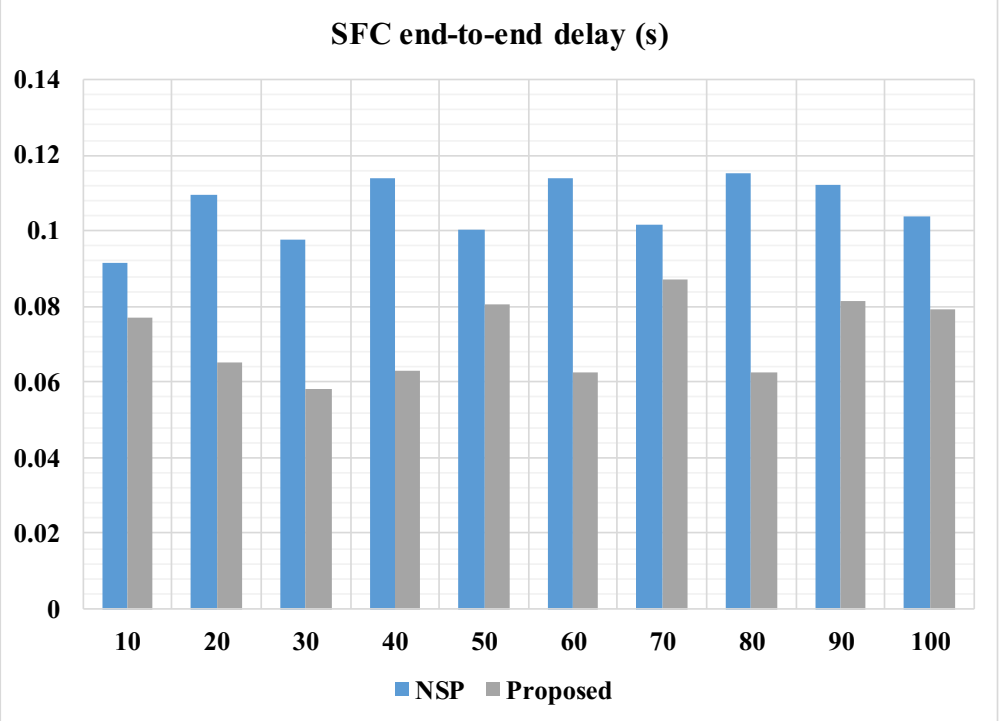


Figure 6-8 SFC end-to-end delay against the number of nodes

6.4 Conclusion

This chapter proposed a novel service SFP allocation algorithm using data collection, network delay, and performance-based on the SF instance selection method. This method can be used in the SCE platform to support and distribute network flows to support service chaining. The proposed algorithm resolves proactive data collection issues and local queue time calculation using systematic data collection and queue time approximation method. In addition, the path selection uses heuristics to assign SFCs to a single node, if possible, to execute multiple services through the same node. The proposed algorithm minimizes the end-to-end delay by more than 10% compared to available SFP path selection algorithms. The proposed algorithm shows its ability to improve the end-to-end delay of the smart community services by distributing the requests through the SCE nodes. This would allow SCE nodes to chain and route user requests to support the smart community services.

Chapter 7 Summary of the study

7.1 SCE platform

The SCE platform integrates SPL, and SCE containerized service with consistency-guaranteed migration and service chaining to support smart community services. The SCE platform was implemented and evaluated through DPDK, Hyperscan, and Docker container technology. The SCE platform provided SCA and distribute rule change method to support multiple services using Docker containers. The SCE platform can use SCA to analyze in-transit data by using a distributed rule application to provide data to multiple services. Furthermore, the SCE platform can provide a live container migration that allows smart community services to migrate without affecting the network flows. The SCE's consistency guaranteed migration method used the SML and Docker container API of the SCE platform to buffer the packets to support loss-free live container migration. In addition, The SCE platform can route user request through computational delay aware service function chaining. The SCE platform integrates the service function chaining to reduce the end-to-end network delay of the user requests. The SCE platform integrates SCA, containerized services, service migration, and service chaining to support smart community services while reducing network delay for the end-users.

7.2 Conclusion

This study summarized the architecture, implementation, and performance of the proposed SCE platform using DPDK, Hyperscan, and Docker technologies. Chapter 3 summarized the implementation, experiments, and test results of upgrading the SPL to use the Intel DPDK and Hyperscan technology by using SCA on zero-copy packet buffers. The results demonstrate that the upgraded SPL can perform SCA at 1Gbps line rates using only eight cores. This is a 0.8Gbps throughput increase compared to the older versions of Libpcap-based SPL. Moreover, upgraded DPDK-based SPL achieved a 10Gbps line rate with 16 CPU cores. Moreover, the SPL was demonstrated in GCTC to provide anonymization services using electrical power usage data. Therefore, the experiments discussed above and the obtained results demonstrate that the DPDK-based SPL can work as a gateway device to perform SCA in smart community networks.

In Chapter 4, the SCE node was proposed with the use of SPL. The proposed SCE node employed MSSCA and container technology to support multiple smart community

services. SCE platform allowed MSSCA content to be directly transferred to services without network packet processing at the service containers. The SCE node's MSSCA achieved a throughput of 1-10 Gbps with 4-16 CPU cores in conventional hardware systems. In addition, the SCE platform proposed a distributed rule change method for the Hyperscan library to change regular expression without affecting network flow. The SCE node achieved a 10 Gbps SCA throughput for 100 accumulated rules, which allowed more than ten rules per service. The proposed distributed rule change method needs less than 0.3 ms to execute and does not affect SPL network flows' performance. SCE node supported eight similar services while providing a 500 Mbps MSSCA content bandwidth for each service, where each service can support 5000 sensors with a 100 kbps bandwidth. Additionally, the total maximum delay of the SCE node is maintained at less than 1 ms, allowing delay-sensitive services to operate at SCE nodes.

In Chapter 5, the SCE platform's capabilities were extended by introducing CGM and O2NCGM migration that supports consistency guaranteed multi-container migration for smart community services. The proposed migration method handled network consistency without affecting the network flows. Container layer separation was used to reduce data transactions between the nodes in container image layers. Furthermore, CGM was applied with LLM to achieve zero packet loss while reducing the network downtime by more than 10% compared to conventional methods. The overall results show that the CGM and O2CGM reduce the migration time by more than 10% for containers with image size higher than 400MBs. Additionally, O2CGM provides consistency guaranteed one to N migration to support service distribution under different network loads.

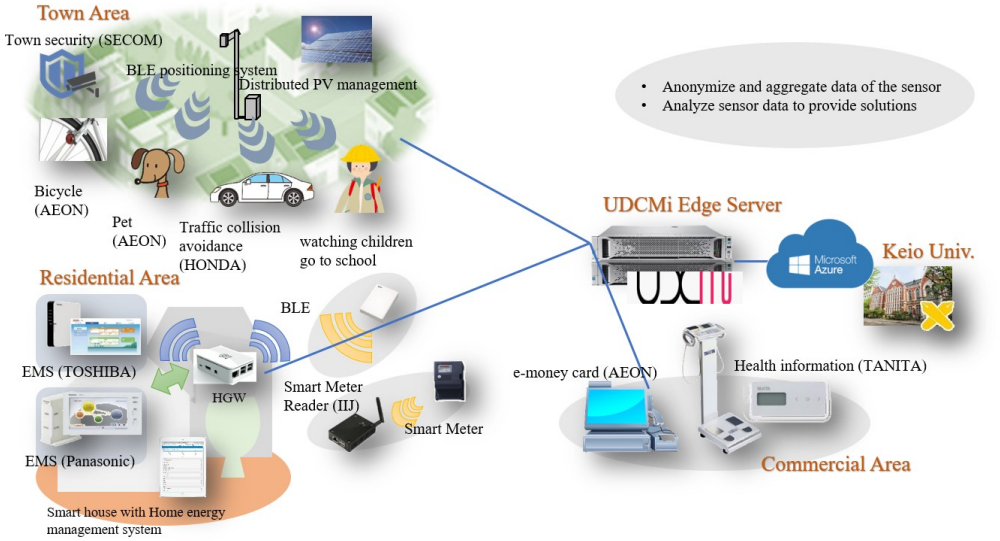


Figure 7-1 Real-world applications of the SCE platform at UDCMi smart city

The SCE capability of user request distribution was improved with a novel

service SFP allocation algorithm using systematic data collection, network delay, and computational delay-based on the SF instance selection method. This method can be used in the SCE platform to support and distribute IoT requests at the SCE. The proposed algorithm minimized the end-to-end delay by more than 10% compared to available SFP path selection algorithms showing the applicability to use at SCE.

Finally, the SCE platform is applied in the real world at UDCMi smart city[95] as a 1Gbps edge server. In this real-world application, the SCE platform was used to support different smart community services such as health care data management, smart building management, and smart house management, as shown in Figure 7-1. The SCE platform provides anonymization, watermarking, and aggregation to remove any personal information collected by the sensors by applying these multiple services to the network flow. The UDCMi SCE platform also provides aggregate smart energy data and provides recommendations to users. The SCE platform operation in UDCMi shows the applicability of the SCE platform in the real world.

This dissertation presented a real-world implementation of the SCE platform that is able to support smart community services while providing consistency, guaranteed service migration, and efficient user request distribution through service chaining.

References

[1] “History of ICT - CS1105 Group Reports 2008 - Wiki.nus.” https://wiki.nus.edu.sg/display/cs1105groupreports/History+of+ICT (accessed Nov. 18, 2020).

[2] “A Brief History of the Internet.” https://www.usg.edu/galileo/skills/unit07/internet07_02.phtml (accessed Nov. 18, 2020).

[3] “Who Coined 'Cloud Computing'?,” *MIT Technology Review*. https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing/ (accessed Nov. 18, 2020).

[4] “What is REST - REST API Tutorial.” https://restfulapi.net/ (accessed Nov. 18, 2020).

[5] “ETSI - Standards for NFV - Network Functions Virtualisation | NFV Solutions.” https://www.etsi.org/technologies/nfv (accessed Nov. 18, 2020).

[6] “Number of IoT devices 2015-2025,” *Statista*. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/ (accessed Nov. 18, 2020).

[7] gk, “Open Fog Computing and Mobile Edge Cloud Gain Momentum | Y.I Readings.” http://yucianga.info/?p=938 (accessed Jul. 07, 2017).

[8] “Home - JSCA Japan Smart Community Alliance.” https://www.smart-japan.org/english/ (accessed Nov. 18, 2020).

[9] “OSA | PON Roadmap [Invited].” https://www.osapublishing.org/jocn/abstract.cfm?URI=jocn-9-1-a71 (accessed Jul. 18, 2019).

[10] K. Tanaka, A. Agata, and Y. Horiuchi, “IEEE 802.3av 10G-EPON Standardization and Its Research and Development Status,” *J. Light. Technol.*, vol. 28, no. 4, pp. 651-661, Feb. 2010, doi: 10.1109/JLT.2009.2038722.

[11] S. Gallenmüller, P. Emmerich, R. Schönberger, D. Raumer, and G. Carle, “Building Fast but Flexible Software Routers,” in *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, Piscataway, NJ, USA, 2017, pp. 101-102, doi: 10.1109/ANCS.2017.21.

[12] Y. Ohara, Y. Yamagishi, S. Sakai, A. D. Banik, and S. Miyakawa, “Revealing the Necessary Conditions to Achieve 80Gbps High-Speed PC Router,” in *Proceedings of the Asian Internet Engineering Conference*, New York, NY, USA, 2015, pp. 25-31, doi: 10.1145/2837030.2837034.

[13] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon, “The Power of Batching in the Click Modular Router,” in *Proceedings of the Asia-Pacific Workshop on Systems*, New York, NY, USA, 2012, p. 14:1-14:6, doi: 10.1145/2349896.2349910.

[14] L. Rizzo, L. Deri, and A. Cardigliano, “10 Gbit/s Line Rate Packet Processing Using Commodity Hardware: Survey and new Proposals,” p. 8.

[15] “netmap: A Novel Framework for Fast Packet I/O | USENIX.” https://www.usenix.org/node/168897 (accessed Feb. 27, 2018).

[16] S. Higginbotham, “In a distributed world cache is king. Why routers are becoming the new server.,” Jan. 31, 2014. https://gigaom.com/2014/01/31/in-a-

distributed-world-cache-is-king-why-routers-are-becoming-the-new-server/ (accessed Jul. 07, 2017).

[17] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2016, pp. 203-216, Accessed: Apr. 04, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026894.

[18] J. Soares, M. Dias, J. Carapinha, B. Parreira, and S. Sargento, "Cloud4NFV: A platform for Virtual Network Functions," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, Oct. 2014, pp. 288-293, doi: 10.1109/CloudNet.2014.6969010.

[19] G. A. Gibson and R. Van Meter, "Network attached storage architecture," *Commun. ACM*, vol. 43, no. 11, pp. 37-45, 2000.

[20] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete Container State Migration," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2017, pp. 2137-2142, doi: 10.1109/ICDCS.2017.91.

[21] M. R. Islam, M. M. S. Pahalovim, T. Adhikary, M. A. Razzaque, M. M. Hassan, and A. Alsanad, "Optimal Execution of Virtualized Network Functions for Applications in Cyber-Physical-Social-Systems," *IEEE Access*, vol. 6, pp. 8755-8767, 2018, doi: 10.1109/ACCESS.2018.2805890.

[22] G. P. Hancke, B. de Carvalho e Silva, and G. P. Hancke, "The Role of Advanced Sensing in Smart Cities," *Sensors*, vol. 13, no. 1, pp. 393-425, Dec. 2012, doi: 10.3390/s130100393.

[23] *Smart Communities Guidebook: Building Smart Communities, how California's Communities Can Thrive in the Digital Age*. International Center for Communications, College of Professional Studies and Fine Arts, San Diego State University, 1997.

[24] X. Li, R. Lu, X. Liang, X. Shen, J. Chen, and X. Lin, "Smart community: an internet of things application," *IEEE Commun. Mag.*, vol. 49, no. 11, pp. 68-75, Nov. 2011, doi: 10.1109/MCOM.2011.6069711.

[25] J. Bryzek, "Trillion sensors: Foundation for abundance, exponential organizations, Internet of Everything and mHealth," *Sens. Mag.*, 2014.

[26] J. Biswas *et al.*, "Processing of wearable sensor data on the cloud - a step towards scaling of continuous monitoring of health and well-being," in *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*, Aug. 2010, pp. 3860-3863, doi: 10.1109/IEMBS.2010.5627906.

[27] C. Ji, Y. Li, W. Qiu, U. Awada, and K. Li, "Big Data Processing in Cloud Computing Environments," in *2012 12th International Symposium on Pervasive Systems, Algorithms and Networks*, Dec. 2012, pp. 17-23, doi: 10.1109/I-SPAN.2012.9.

[28] R. Hummen, M. Henze, D. Catrein, and K. Wehrle, "A Cloud design for user-controlled storage and processing of sensor data," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, Dec. 2012, pp. 232-240, doi: 10.1109/CloudCom.2012.6427523.

[29] H. Nishi, "Information and communication platform for providing smart community services: System implementation and use case in Saitama city," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, Feb. 2018, pp. 1375-1380, doi: 10.1109/ICIT.2018.8352380.

[30] M. A. Lema *et al.*, "Business Case and Technology Analysis for 5G Low Latency Applications," *IEEE Access*, vol. 5, pp. 5917-5935, 2017, doi:

10.1109/ACCESS.2017.2685687.
[31] “OpenFog Consortium.” https://www.openfogconsortium.org/#fog-computing (accessed Jul. 27, 2017).
[32] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog Computing and Its Role in the Internet of Things,” in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, New York, NY, USA, 2012, pp. 13-16, doi: 10.1145/2342509.2342513.
[33] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, “Fog Computing: A Platform for Internet of Things and Analytics,” in *Big Data and Internet of Things: A Roadmap for Smart Environments*, Springer, Cham, 2014, pp. 169-186.
[34] S. Yi, C. Li, and Q. Li, “A Survey of Fog Computing: Concepts, Applications and Issues,” in *Proceedings of the 2015 Workshop on Mobile Big Data*, New York, NY, USA, 2015, pp. 37-42, doi: 10.1145/2757384.2757397.
[35] J. Wijekoon, E. Harahap, and H. Nishi, “Service-oriented Router Simulation Module Implementation in NS2 Simulator,” *Procedia Comput. Sci.*, vol. 19, pp. 478-485, Jan. 2013, doi: 10.1016/j.procs.2013.06.064.
[36] J.-S. Sung, S.-M. Kang, Y. Lee, T.-G. Kwon, and B.-T. Kim, “A multi-gigabit rate deep packet inspection algorithm using TCAM,” in *GLOBECOM ’05. IEEE Global Telecommunications Conference, 2005.*, Dec. 2005, vol. 1, p. 5 pp.-, doi: 10.1109/GLOCOM.2005.1577667.
[37] “NFV - Network Functions Virtualization,” *Cisco*. http://www.cisco.com/c/en/us/solutions/service-provider/network-functions-virtualization-nfv/index.html (accessed Jul. 07, 2017).
[38] “U.S. vs. Japan: Residential Internet Service Provision Pricing,” *New America*. https://www.newamerica.org/oti/policy-papers/us-vs-japan-residential-internet-service-provision-pricing/ (accessed Apr. 03, 2018).
[39] L. Deri, M. Martinelli, T. Bujlow, and A. Cardigliano, “nDPI: Open-source high-speed deep packet inspection,” in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, Aug. 2014, pp. 617-622, doi: 10.1109/IWCMC.2014.6906427.
[40] “Wireshark · Go Deep.” https://www.wireshark.org/ (accessed Jul. 07, 2017).
[41] K. Ikeuchi, J. Wijekoon, S. Ishida, and H. Nishi, “GPU-based multi-stream analyzer on application layer for service-oriented router,” presented at the 2013 IEEE 7th International Symposium on Embedded Multicore/Manycore System-on-Chip, MCSoC 2013, 2013, doi: 10.1109/MCSoC.2013.34.
[42] “DPDK.” https://dpdk.org/ (accessed Apr. 25, 2018).
[43] Red Hat, Inc, “about DPDK.” http://dpdk.org/about (accessed Jul. 07, 2017).
[44] “Introducing PF_RING ZC (Zero Copy),” *ntop*, Apr. 14, 2014. https://www.ntop.org/pf_ring/introducing-pf_ring-zc-zero-copy/ (accessed Feb. 27, 2018).
[45] L. Rizzo and G. Lettieri, “VALE, a switched ethernet for virtual machines,” in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, Nice, France, Dec. 2012, pp. 61-72, doi: 10.1145/2413176.2413185.
[46] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, “Comparison of Frameworks for High-Performance Packet IO,” in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Washington, DC, USA, 2015, pp. 29-38, Accessed: Feb. 27, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?id=2772722.2772729.
[47] “PF_RING,” *ntop*, Aug. 04, 2011. https://www.ntop.org/products/packet-

capture/pf_ring/ (accessed Apr. 25, 2018).
[48] T. Konstantina, Betreuer, W. Florian, and R. Daniel G, "A Survey of Trends in Fast Packet Processing," in *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 2014, pp. 41-48.
[49] D. Luca, "DPDK Summit North America 2018: Using nDPI over DPDK to Classify and Block Unwanted Traffic." https://dpdksummitnorthamerica2018.sched.com/event/IhhK/using-ndpi-over-dpdk-to-classify-and-block-unwanted-network-traffic-luca-deri-ntop (accessed May 30, 2019).
[50] "DPDK PRC Summit 2018: Multiple vDPI Functions using DPDK and H..." https://dpdkprcsummit2018.sched.com/event/EsPY/multiple-vdpi-functions-using-dpdk-and-hyperscan-on-ovs-dpdk-platform (accessed May 30, 2019).
[51] "The Design and Implementation of Open vSwitch | USENIX." https://www.usenix.org/node/188961 (accessed Feb. 27, 2018).
[52] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Commun. Mag.*, vol. 53, no. 2, pp. 90-97, Feb. 2015, doi: 10.1109/MCOM.2015.7045396.
[53] F. Callegati, W. Cerroni, C. Contoli, R. Cardone, M. Nocentini, and A. Manzalini, "SDN for dynamic NFV deployment," *IEEE Commun. Mag.*, vol. 54, no. 10, pp. 89-95, Oct. 2016, doi: 10.1109/MCOM.2016.7588275.
[54] "Enterprise Network Functions Virtualization (NFV)," *Cisco*. https://www.cisco.com/c/en/us/solutions/enterprise-networks/enterprise-network-functions-virtualization-nfv/index.html (accessed Apr. 25, 2018).
[55] "NFV (Network Functions Virtualization) Solution - Juniper Networks." https://www.juniper.net/us/en/solutions/nfv/ (accessed Apr. 25, 2018).
[56] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," *J. Parallel Distrib. Comput.*, vol. 72, no. 11, pp. 1471-1480, Nov. 2012, doi: 10.1016/j.jpdc.2012.01.020.
[57] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges," *IEEE Commun. Surv. Tutor.*, vol. 18, no. 1, pp. 236-262, Firstquarter 2016, doi: 10.1109/COMST.2015.2477041.
[58] F-stack, "F-Stack | High Performance Network Framework Based On DPDK." http://www.f-stack.org/ (accessed Jan. 21, 2019).
[59] L. Rizzo, M. Carbone, and G. Catalli, "Transparent acceleration of software packet forwarding using netmap," in *2012 Proceedings IEEE INFOCOM*, Mar. 2012, pp. 2471-2479, doi: 10.1109/INFCOM.2012.6195638.
[60] N. ISG, "Network Functions Virtualisation (NFV)-Network Operator Perspectives on Industry Progress,"," *ETSI Tech*, 2013.
[61] S. Sreekanth, "VMware vSphere 5.1 vMotion Architecture, Performance and Best Practices," *White Pap. VMware Inc Palo Alto CA USA*, 2012.
[62] V. Medina and J. M. García, "A Survey of Migration Mechanisms of Virtual Machines," *ACM Comput Surv*, vol. 46, no. 3, p. 30:1-30:33, Jan. 2014, doi: 10.1145/2492705.
[63] F. Romero and T. J. Hacker, "Live Migration of Parallel Applications with OpenVZ," in *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, Mar. 2011, pp. 526-531, doi: 10.1109/WAINA.2011.156.
[64] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE*

Cloud Comput., vol. 1, no. 3, pp. 81-84, Sep. 2014, doi: 10.1109/MCC.2014.51.
[65] opencontainers, *opencontainers/runc*. Open Container Initiative, 2019.
[66] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti, "Migrating LinuX Containers Using CRIU," in *High Performance Computing*, 2016, pp. 674-684.
[67] C. Dupont, R. Giaffreda, and L. Capra, "Edge computing in IoT context: Horizontal and vertical Linux container migration," in *2017 Global Internet of Things Summit (GIoTS)*, Jun. 2017, pp. 1-4, doi: 10.1109/GIOTS.2017.8016218.
[68] "What is Docker," *Docker*, May 14, 2015. https://www.docker.com/what-docker (accessed Apr. 07, 2017).
[69] The Kubernetes Authors, "Production-Grade Container Orchestration," Sep. 19, 2019. https://kubernetes.io/ (accessed Sep. 19, 2019).
[70] L. Ma, S. Yi, and Q. Li, "Efficient Service Handoff Across Edge Servers via Docker Container Migration," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, New York, NY, USA, 2017, p. 11:1-11:13, doi: 10.1145/3132211.3134460.
[71] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Live Service Migration in Mobile Edge Clouds," *IEEE Wirel. Commun.*, vol. 25, no. 1, pp. 140-147, Feb. 2018, doi: 10.1109/MWC.2017.1700011.
[72] A. Gember-Jacobson *et al.*, "OpenNF: Enabling Innovation in Network Function Control," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, New York, NY, USA, 2014, pp. 163-174, doi: 10.1145/2619239.2626313.
[73] M. Peuster, H. Küttner, and H. Karl, "Let the state follow its flows: An SDN-based flow handover protocol to support state migration," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, Jun. 2018, pp. 97-104, doi: 10.1109/NETSOFT.2018.8460007.
[74] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer, "SliM: Enabling efficient, seamless NFV state migration," in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, Nov. 2016, pp. 1-2, doi: 10.1109/ICNP.2016.7784459.
[75] J. Halpern and C. Pignataro, "Service function chaining (sfc) architecture," 2015.
[76] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a Model-Driven SDN Controller architecture," in *2014 IEEE 15th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)(WOWMOM)*, Jun. 2014, pp. 1-6, doi: 10.1109/WoWMoM.2014.6918985.
[77] A. M. Medhat, G. Carella, C. Lück, M. I. Corici, and T. Magedanz, "Near optimal service function path instantiation in a multi-datacenter environment," in *2015 11th International Conference on Network and Service Management (CNSM)*, Nov. 2015, pp. 336-341, doi: 10.1109/CNSM.2015.7367379.
[78] M. M. S. Pahalovi, M. R. Islam, T. Adhikary, and M. A. Razzaque, "Optimal execution of virtualized network functions in a multi-data-center cloud," in *2017 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*, Dec. 2017, pp. 602-605, doi: 10.1109/R10-HTC.2017.8289032.
[79] R. Fielding *et al.*, "Hypertext Transfer Protocol -- HTTP/1.1," 1999, Accessed: Jul. 07, 2017. [Online]. Available: http://www.rfc-editor.org/info/rfc2616.
[80] "Manpage of PCAP." http://www.tcpdump.org/manpages/pcap.3pcap.html (accessed Jul. 07, 2017).
[81] "Boyer-Moore algorithm." http://www-igm.univ-mlv.fr/~lecroq/string/node14.html (accessed Jul. 27, 2017).
[82] "MySQL :: MySQL Community Edition." https://www.mysql.com/products/community/ (accessed Jul. 07, 2017).

[83] “POSIX Threads Programming.” https://computing.llnl.gov/tutorials/pthreads/ (accessed Jul. 07, 2017).
[84] INTEL CORPORATION, “Hyperscan,” *01.org*, Sep. 17, 2015. https://01.org/hyperscan (accessed Jul. 07, 2017).
[85] “Regex Set Scanning with Hyperscan and RE2::Set,” *01.org*, Jun. 20, 2017. https://01.org/hyperscan/blogs/jpviiret/2017/regex-set-scanning-hyperscan-and-re2set (accessed Apr. 25, 2018).
[86] “IEEE 1888-2014 - IEEE Standard for Ubiquitous Green Community Control Network Protocol.” https://standards.ieee.org/standard/1888-2014.html (accessed Jul. 18, 2019).
[87] C. McParland, “OpenADR open source toolkit: Developing open source software for the Smart Grid,” in *2011 IEEE Power and Energy Society General Meeting*, Jul. 2011, pp. 1-7, doi: 10.1109/PES.2011.6039816.
[88] “What is Docker,” *Docker*, May 14, 2015. https://www.docker.com/what-docker (accessed Apr. 07, 2017).
[89] Intel technologies, “Accelerating Snort* IPS Throughput Performance Using Hyperscan Pattern-Matching Software.” Intel technologies, 2017, Accessed: Jul. 27, 2017. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/hyperscan-scalability-solution-brief.pdf.
[90] “Containers vs VMs: Which is better for Cloud Deployments?,” *SDxCentral*. https://www.sdxcentral.com/cloud/containers/definitions/containers-vs-vms/ (accessed Sep. 28, 2017).
[91] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, “Container Migration in the Fog: A Performance Evaluation,” *Sensors*, vol. 19, no. 7, Art. no. 7, Jan. 2019, doi: 10.3390/s19071488.
[92] P. Quinn and J. Guichard, “Service Function Chaining: Creating a Service Plane via Network Service Headers,” *Computer*, vol. 47, no. 11, pp. 38-44, Nov. 2014, doi: 10.1109/MC.2014.328.
[93] “Why distribution matters in NFV.” Collaborative White Paper between Alcatel-Lucent and Telefonica, Aug. 2014.
[94] “CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms - Calheiros - 2011 - Software: Practice and Experience - Wiley Online Library.” https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.995 (accessed Jul. 18, 2018).
[95] “共通プラットフォームさいたま版の開発・実証｜プロジェクト｜UDCMi｜アーバンデザインセンターみその,” *UDCMi｜アーバンデザインセンターみその*. http://www.misono-tm.org/udcmi/projects/61.html (accessed Jan. 24, 2019).