Universidade de Aveiro
2022

Alexey Kononov

Execução remota e offloading de computação em NDN

Remote execution and Computation offloading in NDN

Universidade de Aveiro
2022

Alexey Kononov

Execução remota e offloading de computação em NDN

Remote execution and Computation offloading in NDN

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob orientação científica de Daniel Corujo, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e de José Quevedo, Investigador do Instituto de Telecomuncações da Universidade de Aveiro.

agradecimentos / acknowledgements

Em primeiro lugar, quero agradecer ao meu orientador, Prof. Doutor Daniel Corujo, e ao meu co-orientador, Investigador José Quevedo, por terem proposto o tema desta dissertação que se revelou muito interessante e inovador. Agradeço a orientação prestada e as indicações dadas para que o trabalho seguisse na direção correta. O incentivo, a disponibilidade e o apoio dado foram fundamentais para a realização desta tese. Os seus valiosos conhecimentos fizeram grande diferença no resultado deste trabalho e o seu entusiasmo pela área foi uma inspiração para mim e uma mais valia.

Agradeço a minha família, especialmente à minha mãe, Kira Aristova, e à minha irmã, Anna, por todo o apoio que me foi dado de forma incondicional, pela motivação, por estarem sempre disponíveis para me ajudar, em tudo o que foi preciso e principalmente pela preocupação com o meu bem estar.

Obrigado ao meu amigo Paulo Rocha pela sua amizade, por ter sempre uma palavra de apoio e incentivo, por acreditar em mim e por estar presente nos momentos importantes da minha vida.

Aos mestres Rodrigo Santos e Daniel Lopes, pelo seu companheirismo e amizade, pelo seu positivismo que ajudou a ultrapassar muitos desafios nestes cinco anos de estudo, pela sua dedicação nos trabalhos de grupo e pela sua boa disposição que torna todos os convívios ainda mais divertidos.

A todos as outras pessoas que estiveram no meu percurso académico e deixaram o seu contributo para o meu sucesso.

À Universidade de Aveiro pela qualidade do ensino e das condições

Esta dissertação foi realizada com o apoio do Instituto de Telecomunicações.

keywords ICN, NDN, Remote execution, Edge computing, Computation offloading

abstract The way the Internet is currently used is completely different from how it was intended to be used, which results in a number of shortcomings for the current demands. To address IP's deficiencies, new clean-slate architectures started to emerge. Information-Centric Networking (ICN) is one of them. It provides architectural advantages like resource naming, built-in caching and security, effective forwarding and others. One of the Internet usage segments that has expanded the most is edge computing. Applying new architectures to the edge computing has many benefits. The advantages of using Named Data Networking (NDN), an ICN implementation, in the context of edge computing were examined in this dissertation. The emphasis is on remote execution and offloading of functions to the network. A framework is proposed for enabling remote execution and its application in edge offloading in NDN. New methods for delivering input arguments to functions and obtaining the execution result were studied. The framework is validated and its performance is evaluated.

palavras-chave

ICN, NDN, execução remota, computação na edge, offloading de computação

resumo

A forma como a Internet é usada atualmente é completamente diferente de como ela se destinava a ser utilizada, o que resulta numa série de deficiências para as exigências atuais. Para resolver as deficiências do IP, novas arquiteturas clean-slate estão a surgir. A Information-Centric Networking (ICN) é uma delas. Oferece vantagens de arquitetura como recursos nomeados, cache e segurança integrados, encaminhamento eficaz e outros. Um dos segmentos de uso da Internet que mais se expandiu é a computação na edge. A aplicação das novas arquiteturas à computação na edge tem muitos benefícios. As vantagens de usar Named Data Networking (NDN), uma implementação de ICN, no contexto da computação na edge foram examinadas nesta dissertação. Execução remota e offloading de funções para a rede são focos principais deste trabalho. É proposto um framework para viabilizar a execução remota e a sua aplicação para offloading de funções na edge em NDN. Novos métodos para entregar argumentos de entrada para funções e obter o resultado da execução foram estudados. O sistema é validado e o seu desempenho é avaliado.

Contents

List of Tables

Intentionally blank page.

List of Figures

Intentionally blank page.

Nomenclature

List of acronyms

ACK	Acknowledgement
AI	Artificial intelligence
CDN	Content Delivery Network
CI/CD	Continuous Integration, Continuous Delivery
CLI	Command-line interface
CPU	Central Processing Unit
CS	Content Store
DNS	Domain Name System
ELF	Executable and Linkable Format
FIB	Forwarding Information Base
GPU	Graphics Processing Unit
IBM	International Business Machines Corporation
ICN	Information Centric Networking
IDC	International Data Corporation
IoT	Internet of Things
IP	Internet Protocol
IT	Information Technology
LTE	Long Term Evolution
NAT	Network Address Translation
NDN	Named Data Networking
NFD	NDN Forwarding Deamon
NIC	Network interface controller

NLSR	Named-data Link-State Routing protocol
OCI	Open Container Initiative
OS	Operating System
PIT	Pending Interest Table
QoE	Quality of experience
RIB	Routing Information Base
TCP	Transmission Control Protocol
TLV	Type-Length-Value
URI	Uniform Resource Identifier
VM	Virtual Machine

Chapter 1

Introduction

The current Internet architecture was created more than three decades ago for a small scientific community. Nobody could have foreseen the success and permanence that the Internet architecture began to gain in the early 1990s, which enabled the connectivity of more than 3 billion mobile and desktop devices. Nowadays, people use networking devices for a wide range of activities, from basic web browsing to video conferencing, content distribution and cloud computing, with the requirement to be always connected with a minimal delay, wherever they are. The explosion of IoT (Internet of Things) has provided exponential growth of edge computing because of improvements it brings to IT (Information Technology) enterprise, such as reduced latency, higher performance, less costs, and security advantages. Due to the incompatibility between the original concept and contemporary use of the IP (Internet Protocol), researchers and industry professionals have been compelled over time to identify potential solutions to overcome IP Internet constraints. Researchers began developing new Internet designs that might, eventually, replace the current one. The most promising of these are those that follow the ICN (Information-Centric Networking) paradigm, a new network communication model in which the conventional host-centric paradigm has been replaced with the new information-centric paradigm.

1.1 Motivations

The constantly expanding number of IoT devices and demand for in-network computing cannot be supported by the current Internet at this pace.

Whereas IoT is producing a vast amount of data that travels in the opposite direction from conventional flows (that is, from the edge towards the core for processing), processing must therefore be relocated closer to the edge in order to provide low service latencies and manage enormous amounts of IoT data. This is where Edge computing comes in, a strategy for computing close to the location where data is collected or used, allowing IoT data to be gathered and processed at the edge, rather than sending the data back to a data center or cloud [5]. Edge computing, in contrast to cloud computing, encourages the use of resources closer to the network edge to be used by a variety of applications, effectively minimizing the latency and the volume of traffic flowing towards the network core.

Worldwide spending on edge computing is expected to reach $176 billion in 2022, an increase of 14.8% over last year, according to [8]. Although the use cases for edge computing were initially very limited, their range and application have quickly grown. Five years ago, an edge network consisted of a few mid-range servers in a ruggedized container, and now Nvidia and Lenovo are deploying GPU (Graphics Processing Unit)-based Artificial Intelligence (AI) systems [8].

ICN is one of the proposed architectures that might eventually replace the IP architecture. The idea behind ICN is to make uniquely named data a fundamental Internet principle. In-network caching and replication are made possible by the fact that data is no longer affected by a device's location, application, storage, or mode of transportation.

ICN principles can directly solve some of the problems present in current edge-computing. Explicitly named functions can be resolved by network nodes, network-layer requests can carry input information for edge-executable functions. In response to user requests, function code may be stored in a node's caches and transported around the network. Expected benefits are greater efficiency, better scalability with respect to bandwidth, and better robustness in challenging communication scenarios.

With this in mind, this work aims to contribute to the development of edge computing using new solutions, more specifically Named Data Networking (NDN), an ICN implementation, and take advantage of its properties in environments with a large volume of data whose processing has to be done closer to the edge.

1.2 Objectives

The following objectives were defined for this work:

1. Create support for remote execution of functions in NDN. The solution should:
   - not disregard the NDN principles;
   - specify how to transmit input parameters for remote functions, have support for direct and indirect input passing;
   - specify at the application level how to execute functions;
   - have two operating modes. One in which the function notifies the requester when the execution has ended, and another in which the requester detects the end of the execution.
2. Allow nodes that are performing computation to offload some of their functionalities to the network.

1.3 Structure

The remaining of this document is organised as follows:

- Chapter 2 covers the state-of-the-art of NDN and discusses developments in the domain of in-network computing in NDN. Also presents tools and technologies that support edge computing development.
- In Chapter 3 the developed framework is presented. All specifications and operating principles are described.

- Chapter 4 exposes framework implementation details and explains deployment decisions. The test scenario used to validate and evaluate the system is described. Additionally, evaluation results are presented and discussed.
- The final Chapter 5 summarizes the ideas explored in this master's thesis and the potential future work that may be done.

Intentionally blank page.

Chapter 2

Background and related work

This Chapter will present the current state of the telecommunications network architecture and relate the current trends in the network use and the difficulty in improving the IP architecture to satisfy these requirements, namely in the area of in-network computing. It also presents NDN as a new alternative network architecture that aims to solve current challenges efficiently. It will talk about works developed in the context of NDN for the implementation of remote execution and support for edge computing. Other covered topics are related to the used technologies.

2.1 Virtualization

International Business Machines Corporation (IBM) invented the Virtualization technology in 1960 basically to maximize the utilization of hardware resources [15]. IT used to build the abstraction layer over the hardware that strongly resembles the primary hardware, operating system (OS) and other system components that means cloning the functionality of the original components into software [26].

This practice allows to operate multiple operating systems, more than one virtual system and various applications on a single server. The benefits of virtualization include greater efficiencies and economies of scale.

A key use of virtualization technology is server virtualization, which uses a software layer - called a hypervisor [16]. Hypervisor creates and runs Virtual machines (VMs). Multiple guests can be supported by the host computer by virtually sharing its resources such as memory, processor, etc. [20]. Hypervisors take the physical resources and separate them so they can be utilized by the virtual environment. They can sit on top of an operating system or they can be directly installed onto the hardware. The latter is how most enterprises virtualize their systems [16]. While the performance of this virtual system is not equal to the performance of the operating system running on a true hardware, the concept of virtualization works because most of the guest operating systems and applications do not need the full use of the underlying hardware [16]. There are six areas of IT where virtualization is making headway:

1. Network virtualization - can combine multiple physical networks to one virtual, software-based network, or it can divide one physical network into separate, independent virtual networks [2].

2. Storage virtualization - combination of physical storage from multiple storage devices into single virtual storage device, managed from a central console.

3. Server virtualization - is the abstraction of server resources from server users, including the size and identification of specific physical servers, Central Processing Units (CPUs) and operating systems. The goal is to increase resource sharing, consumption, and capacity for future growth while sparing the user from having to understand and handle complex server resource specifications.

4. Data virtualization - enables to access, manage and integrate data from multiple sources without knowing its physical location and format.

5. Desktop virtualization - makes possible a user workstation be accessed and controlled by remote connected device.

6. Application virtualization - enables the use of a program on a computer other than the one it is installed on. It allows to install remote apps on a server and deliver the programs to an end user's machine without installing any dependencies.

Advantages of virtualization [16]:

- Lower costs. Virtualization reduces the amount of necessary hardware servers lowering the overall cost of buying and maintaining large amounts of hardware.

- Easier disaster recovery. Regular snapshots provide up-to-date data, allowing virtual machines to be feasibly backed up and recovered. Even in an emergency, a virtual machine can be migrated to a new location within minutes.

- Improved productivity. Fewer physical resources result in less time spent managing and maintaining the servers.

- Isolation provides a completely separate environment in which guests are executed. The guest program performs its activity by interacting with an abstraction layer, which gives access to the underlying resources.

- Scalability refers to the use of virtual tools to increase server workload. Vertical scaling deals with increasing the system's resources, whereas horizontal scaling increases the number of virtual machines.

- Portability is a concept that can be applied if running software in a new environment does not require significant rework.

- Easier migration to the cloud. Virtualization brings companies closer to experiencing a completely cloud-based environment.

- The ability to embrace a cloud-based mindset with virtualization makes migrating to the cloud even easier.

- Lack of vendor dependency. Virtual machines are agnostic in hardware configuration.

2.1.1 Containers

The virtual machine and the container are two technologies that are frequently utilized in hardware virtualization. Unlike virtual machines, containers do not require the provisioning of an operating system. Containers operate on top of the operating system as isolated processes with their own address space. Containers provide an isolated, resource-controlled environment for running applications by utilizing a variety of operating system abstraction and virtualization technologies. It essentially acts as a sandbox around a typical application OS process and is often seen as being significantly more isolated than an uncontainerized process but not as isolated as a VM [9]. Container images are more lightweight than VM images since they specify how applications are packed and only include the application and any dependencies it requires, such as libraries, configurations, runtimes, and tools. Through the Open Container Initiative (OCI), the container image and runtime have been standardized, making containers very portable and universal [9].

Cgroups is the kernel mechanism in Linux that enables the dynamic management of resource allocation for running containers, including memory, network, and CPU, preventing the container from using more resources than intended. Linux namespace provides an abstraction for the kernel resources, giving the appearance that each container has its own distinct set of resources. As a result, when two containers share the same process id, there will not be any conflicts [27].

Containers are more portable than other application hosting technologies as they can move across any hosts that use the same host OS without code changes. The main advantages of containerization over conventional virtualization are increases in memory, CPU, and storage efficiency. An infrastructure can host many more containers on the same infrastructure than VMs.

However, the absence of OS isolation from the host could be a problem with containerization. Comparatively to hypervisor-based virtualization, security risks result from easier access to the entire system with containers because they share a host OS. OS rigidity is yet another drawback of containerization. While hypervisor instances have more flexibility, containers typically have to use the same OS as the base OS [7].

Enterprises use containers widely, and Figure 2.1 presents the results of an International Data Corporation (IDC) survey [14] that revealed 14 factors that are driving this adoption.

The main reason is that using containers is more cost effective. Another aspect is related with Agile development that speeds up the workflow and software pipelines with the help of continuous integration/continuous deployment (CI/CD) technologies. Using containers, the code may be pushed through these new software pipelines and deployed more quickly because they are lightweight, portable and consistent [9]. Containers make system scaling simple and enable faster system startup times than VMs, which is crucial for distributed applications. And container's tools for orchestration and management enable automation of deployment and management of systems.

2.1.2 Docker

Docker is an open source platform that enables developers to build, deploy, run, update and manage containers—standardized, executable components that combine application source code with the OS libraries and dependencies required to run that code in any

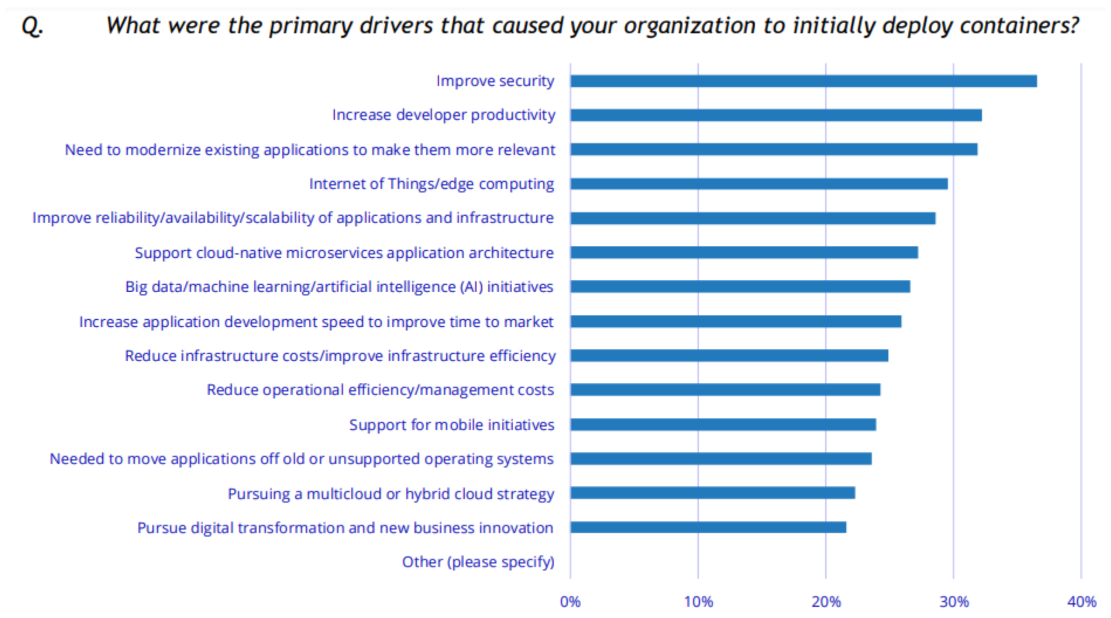


Figure 2.1: Top Container adoption drivers [9].

environment. Docker uses a client-server architecture [10]. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing Docker containers. The Docker client and daemon can run on the same system, or it is possible to connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API (Application Programming Interface), over Unix sockets or a network interface. Another Docker client is Docker Compose, that enables working with applications consisting of a set of containers. Docker images are maintained in a registry [10]. An open registry called Docker Hub is available to everyone, and by default, Docker is set up to search there for images. Operating own private registry is possible.

The necessary images are obtained from the defined registry when the docker pull or docker run commands are used. A push of the image to the defined registry occurs when the docker push command is invoked.

The containerized workspace offered by Docker is created using namespaces, mentioned in section 2.1.1. A set of namespaces is created by Docker for each container when it is running. A layer of isolation is offered by these namespaces. Every component of a container operates in its own namespace and can only be accessed there.

Figure 2.2 shows the differences between VM and Docker containers. Docker containers better fit cloud architectures because of their agility. An application or service that is scalable, dependable, and high-performance is said to be cloud-native if it possesses a certain set of properties and is developed using a specific approach. Containers speed up the development and deployment processes, provide workload portability and mobility across different servers and clouds, and serve as the appropriate foundation upon which to create software-defined infrastructure [11].

Main Docker's tools [11] are:

- DockerFile: DockerFile automates the process of Docker image creation. It is

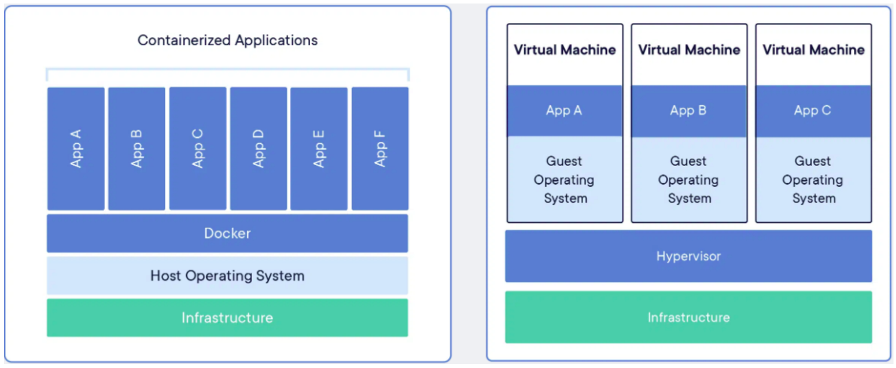


Figure 2.2: VM and Docker comparison [11].

essentially a list of command-line interface (CLI) instructions that Docker Engine will run in order to assemble the image.

- Docker images: Docker images contain executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container.
- Docker containers: Docker containers are the live, running instances of Docker images.
- Docker Hub: is the public repository of Docker images that calls itself the "world's largest library and community for container images".
- Docker Compose: Developers can use Docker Compose to manage multi-container applications, where all containers run on the same Docker host.

2.2 In-network computing

In-network computing focuses on performing computations within a network utilizing equipment that is already present in a networked system and is being utilized to forward traffic. It differs from the term network computing, which refers to networked systems or computers situated within the network. In-network computing utilizes switches and Network Interface Cards (NICs), therefore adding new hardware to the network is not essential. As a result, in-network computing has a low overhead because no additional resources like space, money, or idle power are needed. By finishing transactions as they go through the network, in-network computing even reduces rather than increases the network load. In-network computing has been made possible by the development of programmable switches and the emergence of SmartNICs. In the past, network devices had fixed functionality and could only handle what was specified by the manufacturer, whereas programmable network devices allow creating new functionalities by writing code in high level languages [23].

Since the term in-network computing refers to processing that takes place within the network, it means that transactions are terminated along their path rather than when they reach an end-host, eliminating the latency introduced by the end-host, and any network devices along the way from the in-network computing node to the end-host.

Throughput, the second performance benefit, is a property of the packet processing rate with pipelined smart switches [23].

Several classes of applications have been using in-network computing [23]. The first class consists of network functions implemented within a network device, such as a load balancer, Network Address Translation (NAT), or Domain Name System (DNS) server. Caching within network device is a second class of applications that had success. Data aggregation and reduction applications are the third category of in-network computing applications. These applications use the network device to combine or fetch data from many sources and to minimize the quantity of data carried from the network device forward. The last one is particularly important because of the IoT. The IoT devices generate big amount of data that would be more efficiently processed closer to the source of the data, i.e. closer to the edge. In these scenarios network devices with computation capabilities are the game changers.

2.3 NDN

Layers are the foundation of the Transmission Control Protocol / Internet Protocol (TCP/IP) model's core design principle, starting with the physical device layer and progressing to the application layer. In IP architectures the most crucial functionalities are implemented in the application layer. As a result, the core of modern IP based architectures is the application layer, rather than the network layer with IP as it should be. It appears that in some circumstances, the appropriate abstraction required for modern applications was not achieved.

Applications are currently written with the data requirements in mind. They are not concerned with the location of the data. It leads to shifting the focus from identifying hosts to identifying resources and creation of the DNS service. Content Delivery Network (CDN) and DNS are a remedy for the name-to-content resolution technique used for retrieving most of the content from the network.

Named Data Networking (NDN) has evolved as a practical Internet data-centric architecture created to address TCP/IP's shortcomings and enable strategic edge computation. It moves the functionalities provided by the current IP-based stack from the application layer to the network layer.

The NDN vision is based on the following principles as stated in the NDN project [1]:

1. **Universality:** NDN should be a common network protocol for all applications and network environments.

2. **Data-Centricity and Data Immutability:** NDN should fetch uniquely named, immutable "data packets" requested using "interest packets".

3. **Securing Data Directly:** Security should be the property of data packets, staying the same whether the packets are in motion or at rest.

4. **Hierarchical Naming:** Packets should carry hierarchical names to enable demultiplexing and provide structured context.

5. **In-Network Name Discovery:** Interests should be able to use incomplete names to retrieve data packets.

6. **Hop-by-Hop Flow Balance:** Over each link, one Interest packet should bring back no more than one Data packet.

Figure 2.3 compares the IP and NDN protocol stacks. It is visible that NDN's protocol stack is similar with today's TCP/IP network, both in featuring an hourglass appearance, while the difference is in the "thin waist" , where content chunk replaces IP. From network perspective it brings some differences on data security regime and various routing strategies [19].

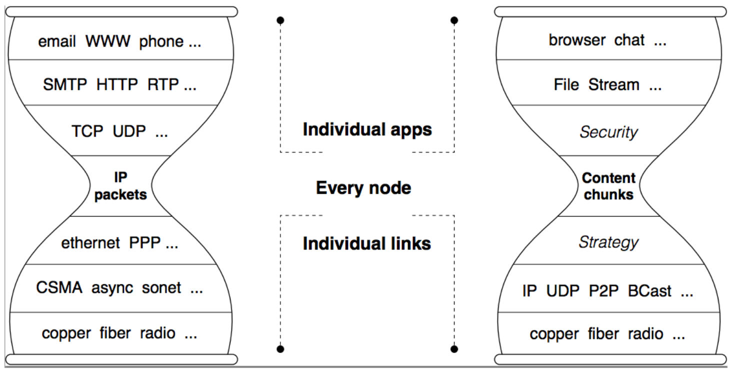


Figure 2.3: TCP vs NDN protocol stack [4].

NDN works at the network and transport layers. Unlike TCP/IP, which organizes data flow based on IP addresses to and from servers, NDN's requests are based on the named data. NDN handles content rather than hosts.

NDN packets are encoded in the TLV (Type-Length-Value) format, which is another difference between NDN and IP. Without a packet header or protocol version, TLV encoding depicts an NDN packet as a group of sub-TLVs. A TLV block is made up of a series of bytes beginning with a predetermined number (Type), followed by the block's Length and Value. In order to carry out communication, NDN defines two sorts of packets: Interest and Data. Both packets include a name and may also include other data.

2.3.1 Naming

Content is identified through a hierarchical name that contains a sequence of name components.

Applications set the naming conventions, which give users a choice on how to name and request data. As a result, names are opaque to the network. Routers do not interpret the entire name; rather, they access name components independently for routing and forwarding purposes. Without having to keep track of a mapping between network requirements and application configuration, this enables application developers and users to construct the namespace that best meets their needs.

2.3.2 Communication process

As mentioned above, in NDN there are two types of packets, Interest and Data. The consumer drives communications by sending an Interest packet to request data. Data packets are made by producers by binding a name to content. Data packets contain the actual data, the name of the application requesting the data, some metadata and they are cryptographically signed by its producer, offering an always-available data verification. Data packets are immutable. For privacy reasons, sensitive payload or name components can also be encrypted.

Each NDN node requires three data structures to process packets: FIB (Forwarding Interest Base), PIT (Pending Interest Table) and CS (Content Store).

1. The PIT maintains an entry for every forwarded Interest until its corresponding Data is received or until the entry lifetime has expired.

2. The CS allows each Data packet to be reused to satisfy other Interests requesting the same content, since Data packets are self-secured and not related to specific hosts. This provides NDN with a native in-network caching.

3. The FIB contains information about the reachability of the content. A FIB entry connects a content-name prefix to the interface(s) from which the data can be retrieved. The FIB, which is filled by routing protocols, is checked each time a node must forward an Interest upstream using the longest prefix matching. When a match is found, the Interest is sent to the appropriate face. The face is a generalization of the network interface that can represent a physical network interface, an overlay inter-node channel, or an intra-node inter-process communication channel to an application.

Figure 2.4 shows the forwarding processes on NDN node. When the Consumer intends to retrieve data from the network, it sends an Interest to NDN router. On arrival of the Interest, the router checks the CS for matching data, if found, it forwards the data packets back to the consumer. Otherwise, the router looks up the name content in its PIT for matching entries, if entry is not found, it records the name of the content and incoming interface, and forwards the Interest to the next hop through FIB, otherwise, it aggregates and records the interface only. The same process takes place up towards the content producer. Once the data producer received Interest from the router, it forwards the Data packet back through the interface of the received Interest to the downstream interface recorded in PIT [13].

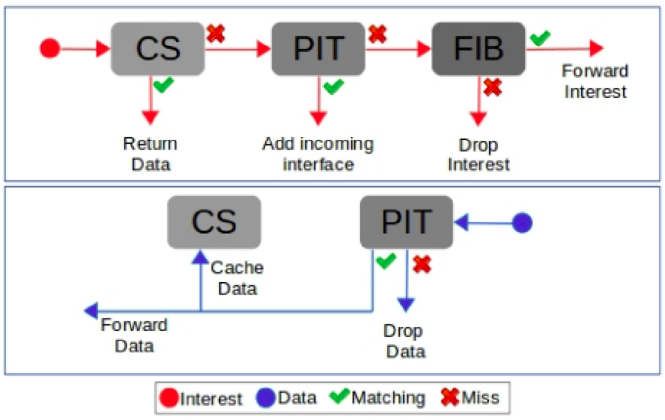


Figure 2.4: Packet forwarding in NDN [6]

2.3.3 Caching

In NDN, data in-network caching is made possible by named data, in contrast to TCP/IP where each data request is distinct. Even though a network appliance has the information that a second user might need in its buffer, the IP protocol requires the whole IP request to be sent. NDN Data that is still in a buffer or cache can be delivered if a Consumer expresses an Interest for the designated data.

2.3.4 Mobility

Mobile nodes in NDN do not need to acquire an address with each change in location and may continue communicating with little interruption by accessing content by names rather than host addresses. As a result, NDN supports consumer-side mobility out of the box.

NDN provides host multi-homing. Switching from Bluetooth to Wi-Fi or 4G LTE in a TCP/IP architecture can present networking difficulties due to the various IP addresses connected with porting to these interfaces. The data name in NDN is unrelated to the interface.

2.3.5 Routing and forwarding

Routing and forwarding are the two parts of the NDN communication process that can be separated. Reachability of the data is propagated and maintained by routing protocols during the routing phase. In order to deliver packets from the source to the destination, the forwarding phase uses routing tables.

Consumers can locate data at the closest source thanks to NDN's multipath, multicast, and stateful data delivery techniques. With a stateful forwarding plane forwarders keep a state for each data request (Interest packet) and erase the state when a corresponding Data packet is received. It allows intelligent forwarding strategies and eliminates loops. These characteristics also increase tolerance by reducing errors in transmissions. NDN optimizes these properties using various forwarding algorithms based on pre-programmed policies.

2.3.6 Security

Security-wise, NDN is resistant to the majority of attacks made against the TCP/IP stack. Authenticity is enabled by NDN's end-to-end security, since named data is signed at creation by its producer. Because the identified data is encrypted when it is signed, secrecy is enabled. Additionally, it makes availability possible thanks to the redundancy that network caching offers.

Public-key cryptography is used for packet signature and verification in NDN security. Every communication participant (application) makes use of name(s) and a public-private key pair(s). Certificates are used to establish a connection between an application, the name(s) it uses, and its key(s). In reality, a certificate is just a regular Data packet that contains information about a public key associated with a certain name and attests that the name and the key belong to the designated user. In other words, the entity that issued the certificate must sign it. A typical scenario is where an entity can grant certificates to other entities that are permitted to create content in its sub-namespaces. Following this methodology, each entity in the system is approved by its superior entity until we reach the system's authority. As a result, in order to enable recursive entity identification verification, each system's authority requires a local trust anchor that proves its identity.

2.3.7 Limitations

Despite all of its benefits, NDN still presents some open questions for research and it has its own vulnerabilities, different from the ones existing in the IP architecture.

One of the components most prone to issues is the caching feature. On the one hand, content caching in NDN creates a significant privacy concern despite its obvious advantages. On the other, when is applied the security of NDN based on public-key cryptography, it causes content chunks to be encrypted for each consumer, which inevitably lowers cache hit rates [24]. Content poisoning is another vulnerability. In this case, a malicious content provider publishes content using another registered provider's name. The content might not be signed at all, or it might be signed with a false signature and in the situations when routers lack the resources to thoroughly check each Data packet, the probability of content poisoning increases [24]. Dealing with unpopular contents (e.g. voice call) may be considered yet another unresolved issue. However, these contents, such as voice calls, can benefit from caching in cases of loss recovery.

In highly dynamic networks where nodes move constantly, NDN struggles to handle mobility. Because Data packets follow the Interest packets' reverse path, if an intermediate node changes locations after the Interest transfer, the Data packet will be discarded [25].

Another open challenge is to develop inter-domain routing protocols and to map proposed routing solutions at the Internet level.

Naming is yet another area that needs further investigation, as there is no proper naming guideline for application development.

2.3.8 NDN implementation and software tools

ndn-cxx

ndn-cxx is a C++14 library implementing Named Data Networking (NDN) primitives that can be used to write NDN applications.

NFD

Named Data Networking Forwarding Daemon (NFD) is a network forwarder that implements and evolves together with the Named Data Networking protocol. NFD is a core component of the NDN Platform. It is a modular framework allowing easy experiments with new protocol features, algorithms, and applications. NFD has the following major modules [21]:

- Core: Provides various common services shared between different NFD modules. These include hash computation routines, DNS resolver, config file, face monitoring, and several other modules.
- Faces: Implements the NDN face abstraction on top of different lower level transport mechanisms.

- Tables: Implements the Content Store (CS), the Pending Interest Table (PIT), the Forwarding Information Base (FIB), and other data structures to support forwarding of NDN Data and Interest packets.

- Forwarding: Implements basic packet processing pathways, which interact with Faces, Tables, and Strategies. Strategy Support, a major part of the forwarding module, implements a framework to support different forwarding strategies. It includes StrategyChoice, Measurements, Strategies, and hooks in the forwarding pipelines. The StrategyChoice records the choice of the strategy for a namespace, and Measurement records are used by strategies to store past performance results for namespaces.

- Management: Implements the NFD Management Protocol, which allows applications to configure NFD and set/query NFD's internal states. Protocol interaction is done via NDN's Interest/Data exchange between applications and NFD.

- RIB Management: Manages the routing information base (RIB). The RIB may be updated by different parties in different ways, including various routing protocols, application's prefix registrations, and command-line manipulation by sysadmins. The RIB management module processes all these requests to generate a consistent forwarding table, and then syncs it up with the NFD's FIB, which contains only the minimal information needed for forwarding decisions.

NLSR

Named Data Link State Routing Protocol (NLSR) is a routing protocol in NDN that populates NDN's Routing Information Base. NLSR will continue to evolve alongside the Named Data Networking protocol. NLSR calculates the routing table using link-state or hyperbolic routing and produces multiple faces for each reachable name prefix in a single authoritative domain [22].

ndnSIM

ndnSIM is an open-source ns-3 module that enables experimentation with the Named-Data Networking architecture in wireless and wired networks. ndnSIM is fully integrated with the real-world NDN prototypes, the ndn-cxx library and the NDN Forwarding Daemon (NFD).

ndn-tools

NDN Essential Tools is a collection of basic tools for Named Data Networking. Tools in this collection include:

peek: transmits a single Interest/Data packet between a consumer and a producer;
chunks: segmented file transferred between a consumer and a producer;
ping: tests reachability between two NDN nodes;
dump: captures and analyzes live traffic on an NDN network;
dissect: inspects the TLV structure of an NDN packet;
dissect-wireshark: Wireshark extension to inspect the TLV structure of NDN packets.

2.4 In-network computation in NDN

NDN makes it possible to overcome some of the challenges of in-network computing that exist in the IP architecture, namely security risks, data authenticity, integrity and access control which are efficiently supported in NDN.

There are some exemplar works that focus on remote execution in NDN and edge computing support.

2.4.1 NFaaS

One of the related works introduces a concept of Named Function as a Service (NFaaS) [18], whereby rich clients, containing most of the application logic, request for named functions through Interests. The authors propose dynamic service instantiation with system adaptation to user demand. It is implemented as an extension to NDN stack and consists in serverless architecture in which nodes have capacity to store and execute named lightweight Virtual Machines support function execution and include input for the execution of function. There is also added Kernel Store, a data structure to store functions code and decide which ones to execute, based on a unikernel score function, whose purpose is to identify and download the most popular functions. As the result, delay-sensitive functions migrate and execute mostly towards the edge of the network, while bandwidth-hungry functions execute further in towards the core of the network, but always staying within the boundaries of the edge-domain.

2.4.2 RICE

[17] is another work focusing on remote method invocation in ICN. It is a general-purpose network-layer framework whose goal is to enable in-network function execution with client authentication and non-trivial parameter passing, to support cases where computation takes longer than PIT expiry time.

It is proposed 4-way handshake for obtaining shared secret derivation, consumer authentication and input parameter passing.

The decoupling of method invocation from the return of results to enable long-running computations is made through the concept of thunks.

2.5 Summary

Due to the prevalence of wireless connectivity and the availability of many smart devices, the Internet of Things is a new reality. The primary objective of IoT is to enhance human life quality through the collection, processing, and provision of sensed and captured data. Edge devices in edge clouds have powerful processing and storage capabilities that can analyze and provide data quickly, so edge computing should be an excellent solution for helping to overcome the resource constraint issues of IoT devices. IP based architecture is inefficient in the IoT scenarios due to the host-to-host data delivery model. NDN, a promising vision of ICN paradigm comes up to improve IoT based communications because of its advantages. The works mentioned above in the area of in-network computing in NDN have done big contributions to the development of edge-computing in ICN. But there is no complete solution, a framework that leverages remote execution to offload functions on the network. Also, there is no effective method of acquiring the execution output in the mentioned projects.

Support for in-network execution	
RICE	Uses a generic naming scheme for remote execution. Employs the concept of Chunks. Distinguishes Chunks and Functions. The first one is a specific function instance already being run on a specific node, while Function is a method that can be executed by any capable node. Does not present any concrete implementation of functions.
NFaaS	Computing is done on lightweight Virtual Machines in the form of unikernels. As they are stateless, most of the logic is on the client side. In order to maximize users' Quality of Experience (QoE), there are different prefixes introduced to indicate application requirements. Kernel Store exists to store functions and discover most popular functions.
Thesis target solution	Uses NDN standard naming scheme and the nodes capable of computing run functions in the form of dynamically loaded plugins. Allows the nodes to perform computation and to offload their functionalities to the network using Docker containers.

Table 2.1: Comparison of existing works, part 1.

Table 2.1 shows the differences in terms of support for in-network execution and Table 2.2 presents differences in parameters passing and data retrieval implementations. Summarizing, all the previous works have focused on different specific topics of in-network computing, from resource discovery system to data authentication. And this thesis' work aims to deliver a complete solution for remote execution and function instantiation in NDN. The solution also allows the client to provide input parameters as named content through Interests. Another particularity is to allow a Client to be notified about the end of execution by the Server without introducing changes at the network layer.

Parameters passing and data retrieval	
RICE	Parameters can be represented directly in Interest packets when very small in size or as a hash when larger. Uses 4-way handshake to fetch the input parameters. For result retrieval has 3 different methods: 1)Upon reception of Interest the server starts execution. Client polls results sending Interests, if the execution takes long, the PIT entry expires. In this case, the client resends the Interest using the same name in an attempt to recover. 2) Client sets Interest Lifetime long enough to cover the entire computation time. 3)Upon reception of the Client's Interest, the server starts the computations and responds with an ACK message. This acknowledgement is a separate type of packet and must be recognised by all the forwarders. The ACK does not consume the pending PIT entry in the forwarders of the original Interest message, but increases its expiry time by enough to cover the expected response time.
NFaaS	Function input goes as the part of the Interest's name. Other required parameters are sent as TLV elements. There is no support for large input parameters. Does not have any special result retrieval methods, Data packet with result follows the path established by Interest and can increase expiry time of Interests.
Thesis target solution	Input can be sent in Interest or retrieved by the producer from the client or another node by sending Interest to them. For the result retrieval there are two methods. One driven by the Producer of the result, another is conducted by the Client.

Table 2.2: Comparison of existing works, part 2.

Chapter 3

Proposed Framework

This chapter presents the developed framework for the remote execution and function instantiation in NDN.

3.1 Framework for remote execution of functions in NDN

To allow remote execution in NDN a framework was developed based on a protocol that specifies the sequence of exchanged packets and the structure of elements within the packets during the process of remote function execution. It also identifies rules for passing input arguments and two different ways to fetch the execution result, one driven by the Server and the other driven by the Client. The use cases are:

- Remote execution of functions: Nodes that need to do certain data processing or computation and do not have the required function or enough computing power to perform it can abstractly invoke functions instantiated remotely via prefix;
- Instantiation of functions on the edge: enables servers to offload its functions to routers in order to reduce latency and move computing to the edge.

3.1.1 Overview

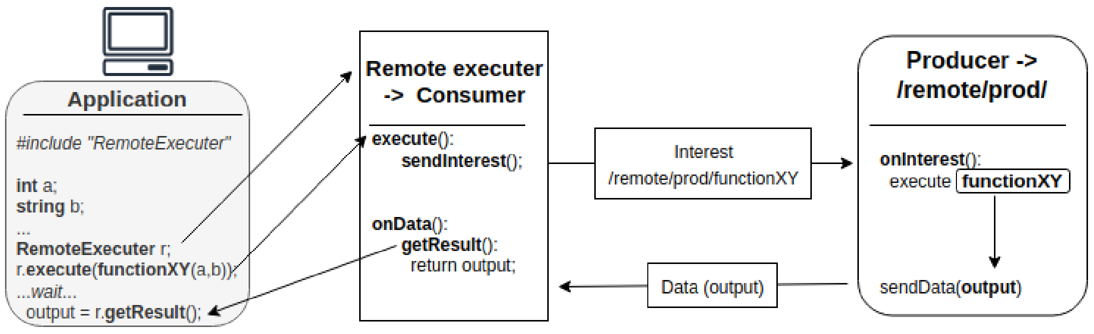
The application that wants to use the framework for remote execution of a function needs to instantiate the module that contains all the logic for the remote execution using NDN and then it only has to call correct methods to pass function arguments, function name, to start execution and to get the result.

The module with the logic is an implementation of the protocol that performs all the operations at the packet exchange level in order to perform remote execution and provide execution results to the application.

Figure 3.1 demonstrates remote execution in a more abstract manner with snippets of pseudo code.

3.1.2 Operating principles

The protocol relies on the exchange of Interest and Data to request execution, pass input arguments, and get the result of remote execution, as normally happens in NDN. The use of the protocol requires that the information contained in the packets complies


Application
Remote executer -> Consumer
Producer -> /remote/prod/
Interest /remote/prod/functionXY
Data (output)

```
#include "RemoteExecuter"

int a;
string b;
...
RemoteExecuter r;
r.execute(functionXY(a,b));
...wait...
output = r.getResult();
```

```
execute():
   sendInterest();

onData():
   getResult():
     return output;
```

```
onInterest():
  execute functionXY

sendData(output)
```

Figure 3.1: Packet processing inside a node

with the formatting and encoding in TLV format, according to the rules defined by the protocol.

Interest packets carry the protocol TLV elements in the Application Parameters, and in Data packets, the protocol TLV elements are placed in the Content.

Requests to execution are expressed by sending Interest to the prefix associated with the function. The NDN default naming scheme is used.

3.1.3 Operating modes

The protocol has two operating modes. The first is the Notify mode - obtaining the execution result is triggered by the Producer which notifies the Consumer when the execution result is available and then the Consumer fetches the result. The second method is the Polling Mode. In this mode, the Consumer periodically sends Interest(s) to the Producer, which responds with Data containing ACK (acknowledge) until the result is ready. When the result is ready, the Producer sends the execution result.

Notify mode has better performance in some applications, namely in the case of offloading. In this mode the Consumer after requesting the execution, can do other tasks, while the execution is done in the Producer. Also the number of packets exchanged between Producer and Consumer is predictable and reduced compared to the Polling mode. There is no overhead in terms of packets, only packets directly related to the remote execution and protocol operations are sent. The only temporal overhead is the time taken by the Producer to send the notification and the Consumer's acknowledge response before fetching the result. It also has some limitations. Sending a notification from the Producer to the Consumer about the end of execution requires the Consumer to have the prefix registered, in order to make this communication possible. This fact makes Consumers exposed to the Producer.

In the Polling mode, the interval at which the result poll occurs is configurable and to define its value there are some conditions discussed in the Chapter 4. The Polling mode is designed for cases when Consumers do not have a registered prefix and therefore do not have the capacity to process Interest packets, or when a Consumer wants to remain anonymous to the Producer. Another case is when a Consumer does not need to know the execution result immediately, for example in the case of low-power IoT devices that only run occasionally.

Figure 3.2 shows a packet flow in the Polling mode. First Interest sent by the

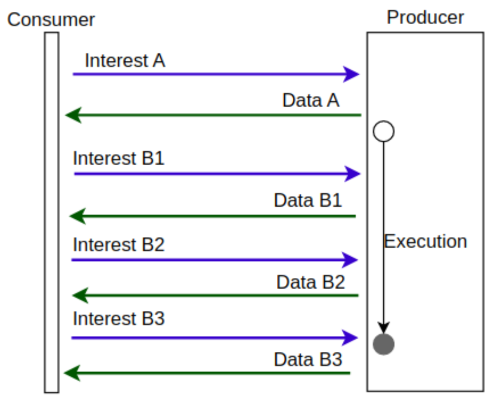


Figure 3.2: Polling mode

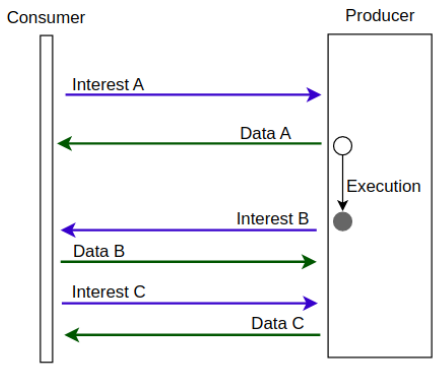


Figure 3.3: Notify mode

Consumer, Interest A, is the execution request and contains information about operating mode, process identification and possibly input parameters. The corresponding Data, Data A, is acknowledge and contains identifications of processes. After sending Data A, the Producer starts the execution. Meanwhile, the Consumer begins to polling execution result by sending Interests, Interests B1 to B3. While the Producer has not finished the execution and thus can not provide the results, it responds to the Consumer's Interests with Data containing just an id of execution process, Datas B1 to B2. Since the moment when the execution is finished and the result is available, the Data sent by the Producer contains execution results, Data B3. The Consumer, after receiving the Data containing the results, Data B3, terminates the polling. The Figure 3.3 presents a packet flow in the Notify mode. The first Interest sent by the Consumer, Interest A, and the corresponding Data, Data A, are similar to the ones exchanged in the Polling mode. After sending Data A, the Producer starts the execution while the Consumer waits for the availability of the results. When the execution is done, the Producer sends an Interest, Interest B, to the Consumer notifying the readiness of the execution results. It responds with an acknowledge, Data B. Then it sends an Interest, Interest C, to the Producer to retrieve the results, which are sent in the Data C.

3.1.4 Types of input

The protocol classifies functions into two classes: functions with large input and functions with no input or small input. In the case of functions with a large input, there is a process for obtaining input parameters.Thus, these are considered implicit input functions, while functions of another class have explicit input, as they do not require an additional process for passing it to Producer.

The input size from which it is considered implicit is configurable. It cannot be too large to not overload the Interest packet size in order to keep them small in size. By the principles of NDN, Interests should be compact and carry the prefix name and little else.

The support for implicit input is not only related to size, but also to allow input parameters to be extracted from a third party as a named content. An example is when a client wants to remain anonymous, it sends the prefix of the node that has input data stored there for the Producer to fetch data from there.

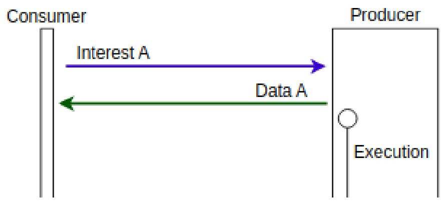


Figure 3.4: Explicit input

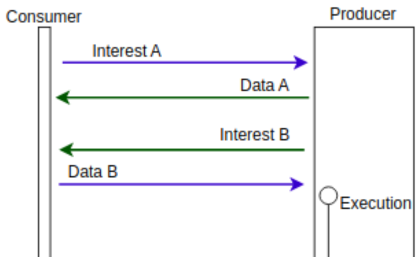


Figure 3.5: Implicit input

Summing up, as the Notify mode requires the Consumer to have a registered prefix, in this mode the implicit input is indicated for cases when the input data is not on the Consumer or when its size is large. In the Polling mode, in cases when the Consumer has no prefix or intends to remain anonymous to the Producer, the implicit input only makes sense if the input data is stored in another node as a named content.

Figure 3.4 shows a packet exchange in case of explicit input. The first Interest sent by Consumer, Interest A, carries the input parameters for function execution and another necessary data. The Producer responds with the Data A, which is an acknowledge, then starts the execution. Figure 3.5 demonstrates the situation in case of implicit input. The first Interest A does not transport any input parameters, but it must contain the prefix from which the input parameters should be retrieved. The Producer sends Data A as acknowledge. Then it sends Interest B to the prefix received in the Interest A, which is the Consumer prefix on this picture, in order to fetch the input parameters. And then, the input parameters are sent in the Data B.

3.1.5 Packet specification

Packet processing is done by parsing TLV elements in Application Parameters in Interests and Content in Data packets. TLV processing is done at two levels. Each packet has a TLV element that contains other TLV elements inside. The outside TLV element will be called wrapper in order to facilitate the explanation. When the packet is received, the first level of processing analyzes what type of wrapper it is. According to this type, it proceeds to TLV elements processing inside the wrapper. For each type of wrapper, specific TLVs are expected, according to the protocol's logic.

The protocol uses the range of TLV Types reserved in the NDN for future assignments, [253, 32767], according to the NDN TLV Registry. The wrappers TLV element types are 4 digits long and use the range from 2000 to 3000. Table 3.1 shows the types of wrappers and their meaning in the protocol. The light blue color means wrappers inside Interests and the light green color means wrappers inside Datas.

First, the TLV element wrapper is processed, then its value is decoded and searched for specified TLVs for the next processing level.

TLV elements of the next level can be either elementary (have actual data already in the Value) or composite (have other TLV elements in the value).

Table 3.2 presents types of elementary elements and how they are interpreted. They use types of range [400 - 499].

Table 3.3 presents types of composite elements and their meaning. They use types in the range [300-399].

Type	Description
2000	Wrapper on Interest to request execution with explicit input.
2001	Wrapper on Interest to request execution with implicit input.
2002	Wrapper on Interest to get execution result after receiving Interest in notify mode from the Producer .
2003	Wrapper on Interest to get execution result in polling mode.
2100	Wrapper on Data response to Interest with explicit input (2000). It works like an ACK.
2101	Wrapper on Data responds to Interest with implicit input(2001). It works like an ACK.
2102	Wrapper on Data response to Interest with result request of execution(2002) in notify mode. Sends the execution result.
2103	Wrapper on Data to respond to Interest with request for execution result in polling mode.
2200	Wrapper on Interest sent by Producer to get implicit input data from Consumer.
2201	Wrapper on Interest sent by the Producer indicating that the result of execution is available.
2300	Wrapper on the Data sent by Consumer containing implicit input parameters.
2301	Wrapper on Data sent by Consumer when it has received Data with type 2201 wrapper. It works like an ACK.

Table 3.1: Wrappers TLV types.

Type	Value
400	input/output
401	length of input/output
402	process id on Consumer
403	Consumer prefix
404	operating mode
405	process id in Producer
498	Acknowledge
499	error code

Table 3.2: Types of TLV elements inside wrappers and meaning of their value.

Type	Value
300	“container” for input
301	”container” for output

Table 3.3: Types of TLV container elements.

3.1.6 The protocol specification for each operating mode and different input types

In this section is described how the processing of the TLV elements in the Interest and Data packets is made in different situations.

Explicit input in notify mode

In the case of explicit input, the Consumer sends an Interest to the prefix associated with the function and waits for the Producer to notify when the result is ready. Before sending the initial Interest in this mode, the Consumer registers its prefix and generates an id for this execution.

Figure 3.6 shows an example of this type of execution.

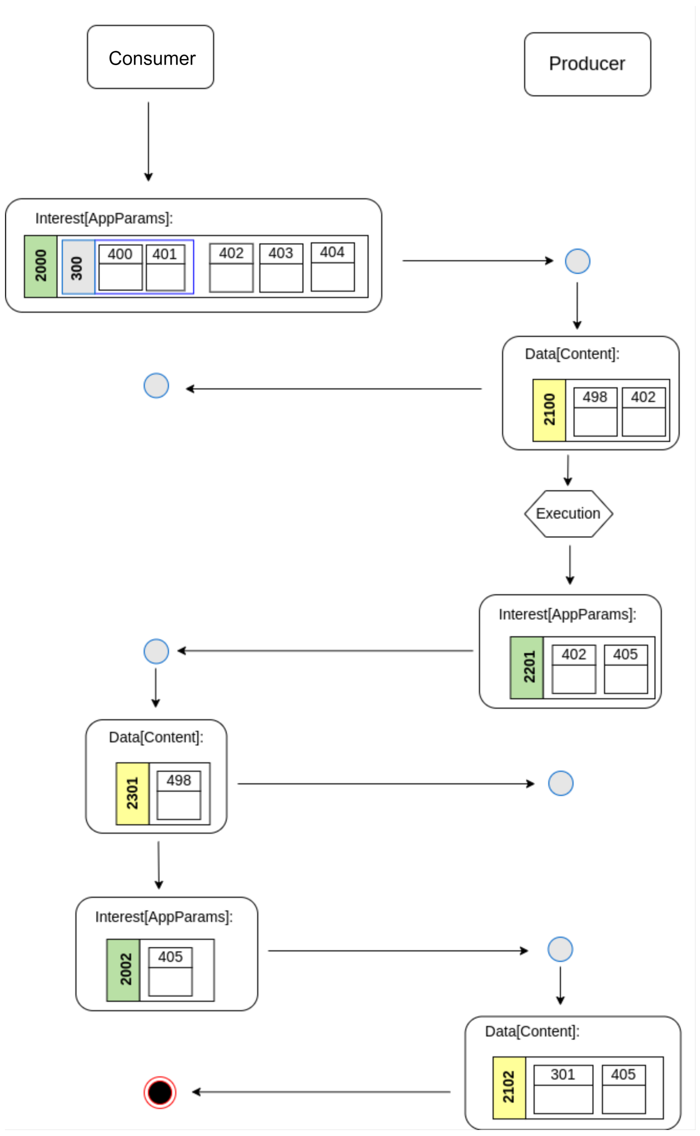


Figure 3.6: Execution with explicit input in Notify mode

The initial Interest in the Application Parameters has TLV element of type 2000, that tells to Producer that it is an execution request and the input is explicit. Inside it has the following elements, as illustrated in Figure 3.6:

```
2000 - wrapper:
|_____ 300 - input container:
        |_____ 400 - input data
        |_____ 401 - length of parameters (optional)
|_____ 402 - Consumer process id.
|_____ 403 - consumer prefix.
|_____ 404 - mode of operation. In this case it has value '1' that means Notify mode.
```

The Producer, after processing the Interest received and if it complies with the protocol rules and has the necessary data, it generates a process ID on its side and sends a Data with TLV element of type 2100 that contains:

```
2100 - wrapper:
|_____ 498 - ACK
|_____ 402 - Consumer process id received in the Interest, in order for the Consumer to be able to map the acknowledgment to the right execution process.
```

After sending a Data packet, the Producer starts the execution.

The Consumer, after receiving the Data, simply waits for the Producer's Interest for the registered prefix indicating the end of execution.

When the Producer finishes the execution it saves the result under process id and sends an Interest with TLV of type 2201 that contains the following elements:

```
2201 - wrapper
|_____ 402 - process id of Consumer
|_____ 405 - process id of Producer
```

The Consumer responds with a Data containing only an acknowledge. The data has in its Content:

```
2301 - wrapper:
|_____ 498 - ACK
```

Then the Consumer sends an Interest to the Producer requesting the execution result with the process id of the Producer. The Interest has the following elements inside its Application Parameters:

```
2002 - wrapper:
|_____ 405 - Producer's process id
```

The Producer sends a Data with 2102 type TLV element, containing the following components:

```
2102:
|_____ 301 - output
|_____ 405 - process id of Producer
```

Explicit input in Polling mode

In this scenario, the Consumer sends an Interest with input data requesting for execution and obtains the result by polling. That is, sending Interests to the Producer until it receives a response with the result. The Producer provides the id that the Consumer polls with.

Figure 3.7 shows an example of this type of execution.

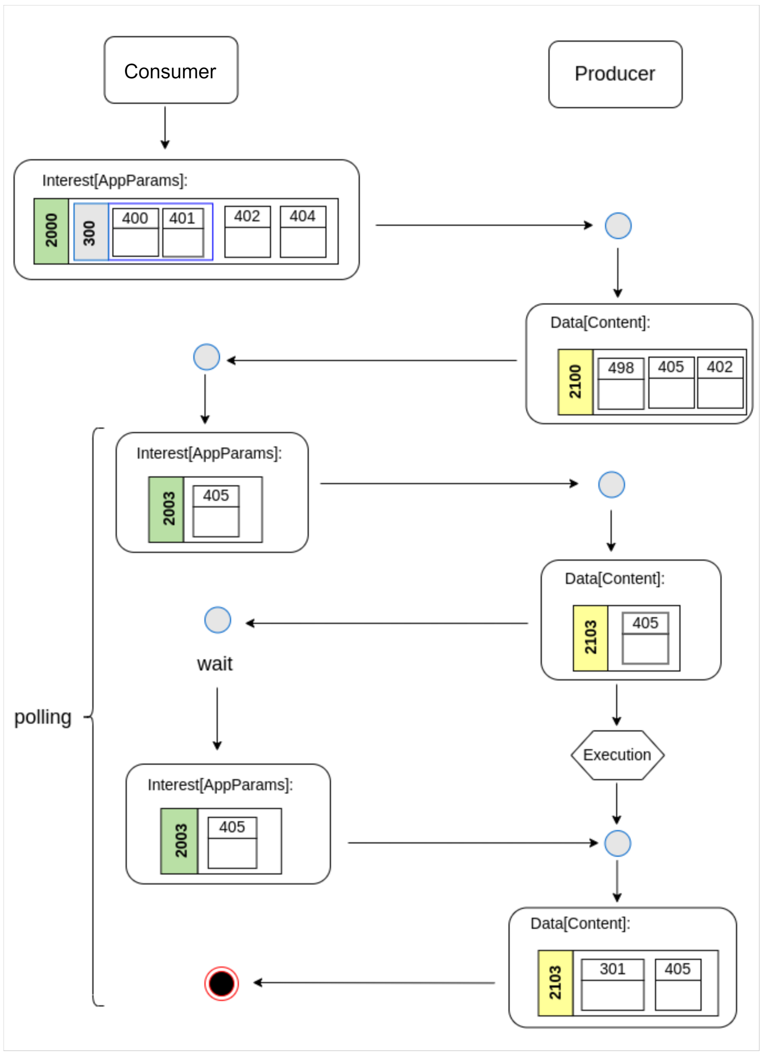


Figure 3.7: Execution with explicit input in Polling mode

The first Interest has the following structure of TLV elements:
2000 - wrapper:
|______ 300 - input container:
    |______ 400 - input data
    |______ 401 - length of input parameters (optional)
|______ 404 - operating mode, its value is '0' - Polling mode.
|______ 402 - process id of Consumer

After processing the execution request received in the Interest, assuming everything is in order, the producer sends a Data with acknowledgement and starts execution right away. Data content has the following structure:

2100 - wrapper:
|_____ 498 - ACK
|_____ 405 - process id of Producer
|_____ 402 - process id of Consumer

After getting Data with the acknowledgement, the Consumer waits the predetermined amount of time for polling to be completed before sending an Interest requesting the result. The Interest in the Application Parameters has the following elements:
2003 - wrapper
|_____ 405 - id in Producer

If the execution has not finished when the Interest is received, the Producer sends Data that only contains the element with the ID information, i.e.,
2103 - wrapper:
|_____ 405 - id

Alternatively, after the execution is complete, the data also includes the element containing the outcome, i.e.,
2103 - wrapper:
|_____ 301 - output
|_____ 405 - id

Implicit input in Notify mode

In this case, the Consumer sends Interest with an execution request, then the Producer fetches input parameters by sending Interest to the Consumer. When the Producer finishes running, it sends an Interest to the Consumer with a notification. The Consumer fetches the result by sending an Interest to the Producer. Figure 3.8 shows an example of this type of execution.

Before sending the first Interest the Consumer registers the prefix and generates an id. The first Interest has an id to associate Data containing acknowledge to the right process, the operating mode information and Consumer's prefix, i.e.,

2001 - wrapper
|_____ 402 - process id of Consumer
|_____ 403 - Consumer prefix
|_____ 404 - operating mode, has value '1'.

After processing the Interest, the Producer first sends an acknowledge and then fetches input from the Consumer by sending an Interest to the Consumer prefix. The Data's structure is as follows:
2101 - wrapper:
|_____ 498 - ACK

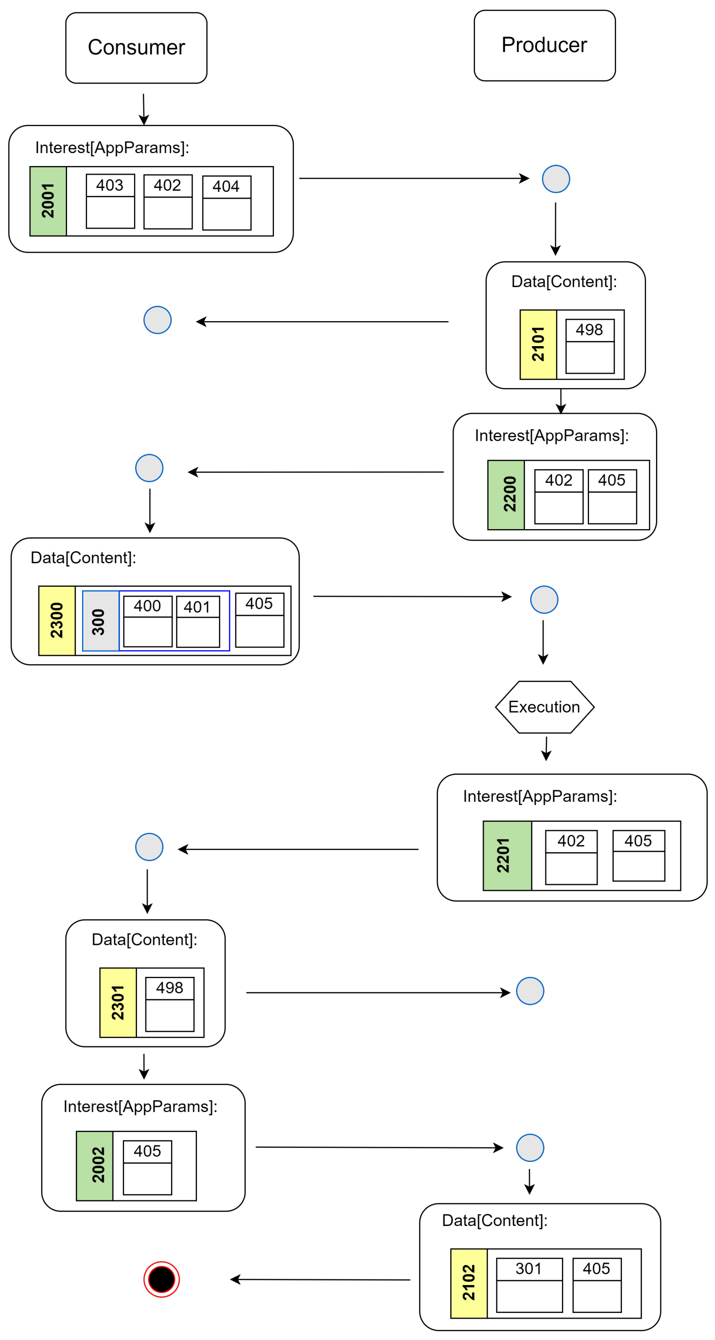


Figure 3.8: Execution with implicit input in Notify mode

The structure of the Application Parameters of the Interest sent by Producer to request Consumer input is:

2200 - wrapper:

|______ 402 - id on Consumer

|______ 405 - id on Producer

The structure of the Data sent by the Consumer with input parameters is:

```
2300 - wrapper:
|_____ 300 - input container:
        |_____ 400 - input data
        |_____ 401 - length of input parameters
|_____ 405 - Producer id
```

After receiving a Data with input from the Consumer, the Producer begins to perform the execution of the function. Consumer waits for the Interest from the Producer. The Application Parameters' content sent by the Producer when execution is done is:

```
2201 - wrapper:
|_____ 402 - id in Consumer
|_____ 405 - id in Producer
```

The Data sent by Consumer with acknowledge is the following:

```
2301 - wrapper:
|_____ 498 - ACK
```

Then, the Consumer sends an Interest with the following elements to fetch the result:

```
2002 - wrapper:
|_____ 405 - id in Producer
```

The structure of the Data sent by Producer with the execution result is the following:

```
2102:
|_____ 301 - output
|_____ 405 - id in Producer
```

Implicit input in Polling mode

In this case, the Consumer sends an Interest with an execution request, then the Producer fetches input parameters by sending an Interest to the Consumer and provides an id to poll the result. The Consumer starts polling by sending Interest(s) with id received from Producer and the Producer responds with Data(s). When it has the result of the execution, the Data will contain the element with the result.

With the exception of the initial step of acquiring the input parameter, the operating concept is the same as the Polling mode with explicit input.

The presence of a prefix from which the input will be taken is necessary for implicit input. Consumers can provide the prefix of another node that can supply input parameters in the element of type 403, if they do not have a prefix or want to stay unnamed. Figure 3.9 shows an example of this type of execution.

The first Interest sent by the Consumer is structured as follows:

```
2001 - wrapper
|_____ 402 - id in Consumer
|_____ 403 - prefix
|_____ 404 - operating mode
```

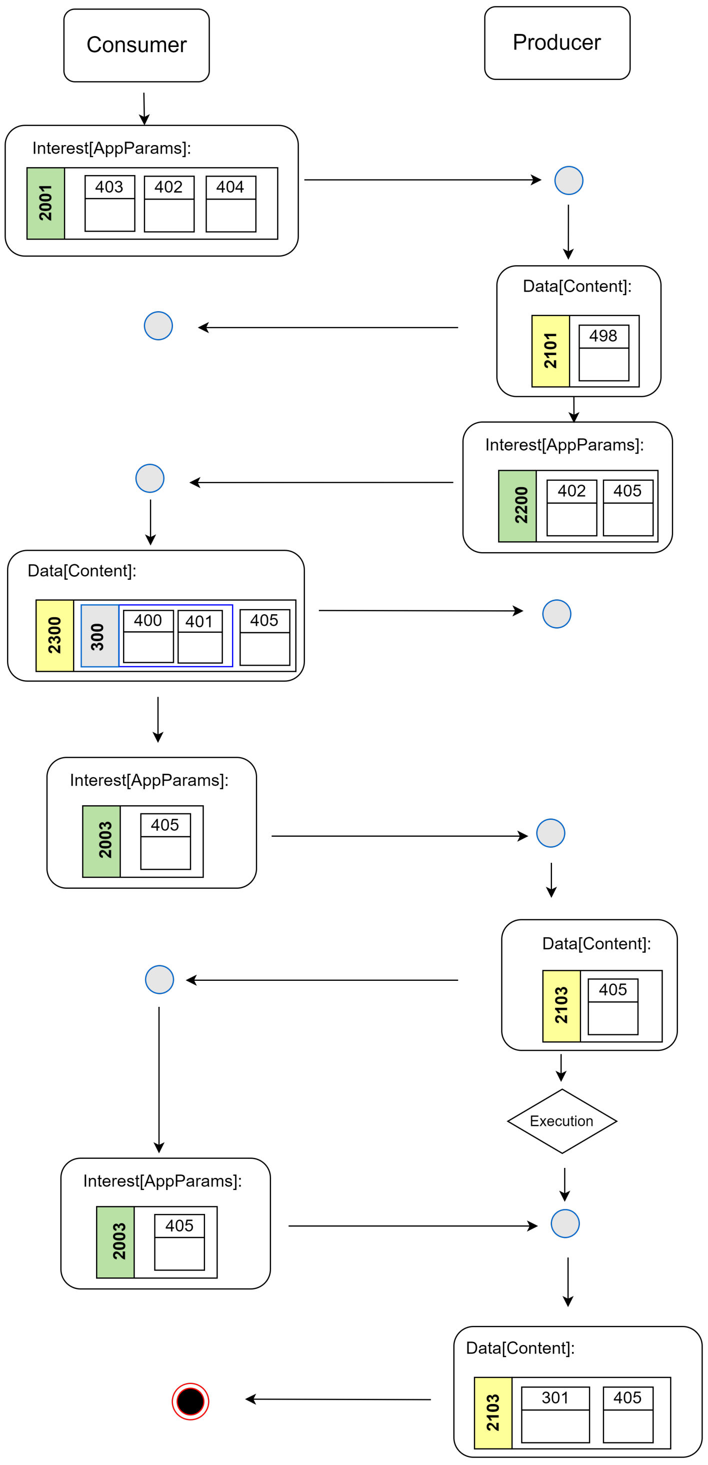


Figure 3.9: Execution with implicit input in Polling mode

The Producer's Data sent after accepting the execution request is defined as:
2101 - wrapper:

```
|_____ 498 - ACK
```

The Interest's application parameters are structured as follows by the Producer in order to solicit input from the Consumer or a node identified by the prefix received previously:

```
2200 - wrapper:
|_____ 402 - Consumer id
|_____ 405 - Producer id
```

The structure of the Data with input parameters sent by Consumer is defined as:

```
2300 - wrapper:
|_____ 300 - input container:
        |_____ 400 - input data
        |_____ 401 - length of input parameters (optional)
|_____ 405 - Producer id
```

The Interest sent by the Consumer during polling is defined as:

```
2003 - wrapper
|_____ 405 - id
```

The Producer transmits Data that just includes the element with the ID information if the execution is not completed when the Interest is received, i.e.,

```
2103 - wrapper:
|_____ 405 - id
```

Otherwise, the element containing the result is included in the Data when the execution is finished as follows:

```
2103 - wrapper:
|_____ 301 - output
|_____ 405 - id
```

3.2 Function execution on routers

Another part of the solution includes offloading of functions to the network. It is assumed that the routers can carry out function execution and perform computations. The objective is to allow the nodes that perform computation to offload part or all of the processing to the network using the developed framework. For this, nodes performing computation request instantiation of their functions on routers. This instantiation is nothing more than executing a function on the router that instantiates the function.

The instantiation of functions is done based on Docker images. Producers who want to instantiate their functions must call the installation function on the router and pass the Docker image as input parameter. This image should allow instantiating a container with corresponding function on the router. In other words, instantiation of functions on the router by Producers is nothing more than remote execution of a function on the router by them.

Figure 3.10 shows an offloading process. Initially, requests to execute Producer functions are forwarded to the Producer, because in the router's FIB the Producer is registered under the prefix /producer. The Producer intends to offload its functionA to the router. For that, it executes remotely the function that is running on the router under prefix /router/install and passes as input parameter the name of the Docker image that simulates the Producer application and contains functionA. The function on the router instantiates the container that runs the Producer application with support for the execution of function A. The application registers the prefix /producer/functionA in the router's local forwarder. From that moment on, all requests to execute function A will be forwarded to the container and not to the Producer, as the forwarder will match the longest prefix in the FIB, which is now the prefix of the container.

More implementation details are described in Section 4.2.

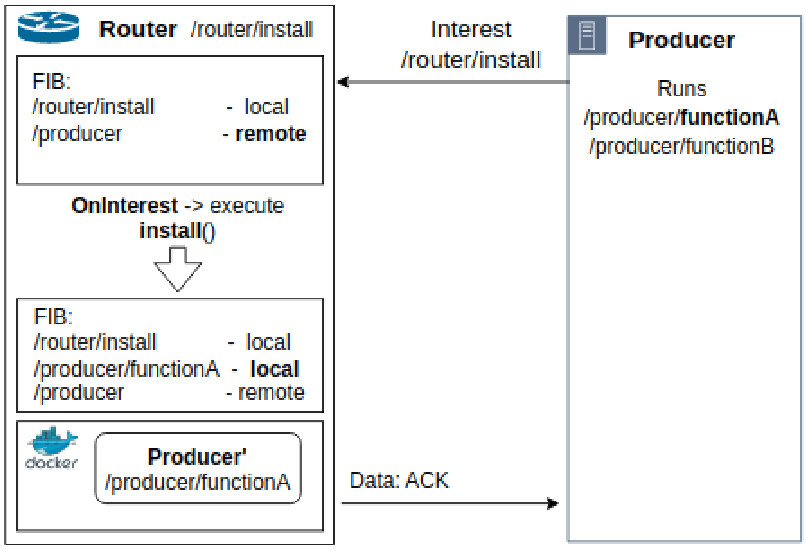


Figure 3.10: Function offloading

3.3 Summary

The framework that was created was discussed in this chapter. The operational principles, including the TLV components used, the operating modes, and the passing of input parameters were presented. For each type of execution, the specification of packet elements was illustrated. Furthermore the mechanism used for the offloading was explained.

Chapter 4

Architecture implementation and validation

In this chapter will be described the implementation of the developed framework. Also the tests carried out and results obtained will be presented.

To test the solution, a scenario with a NDN network, a Consumer application, a Producer application, a router, and four temperature sensors deployed in a particular geographical area was developed. The Consumer application wants to know the average temperature. For this, it sends an Interest to the Producer, which then gets the temperatures of the four sensors and runs a function to get the average temperature. In order to test the protocol, different operating modes, Notify and Polling, for execution of remote function will be examined. Then to validate the offloading of functions to the network, the Producer will instantiate its function on the router.

The router is the Ubuntu 18.04 VM with NDN Producer application and running NFD. The Consumer is running on a host with Ubuntu 18.04 and Producer is running in a Docker container. Both, Consumer and the Producer, same as the router, have their local NFD. In order to interconnect all the nodes, static routes were defined between them.

Figure 4.1 shows the test scenario.

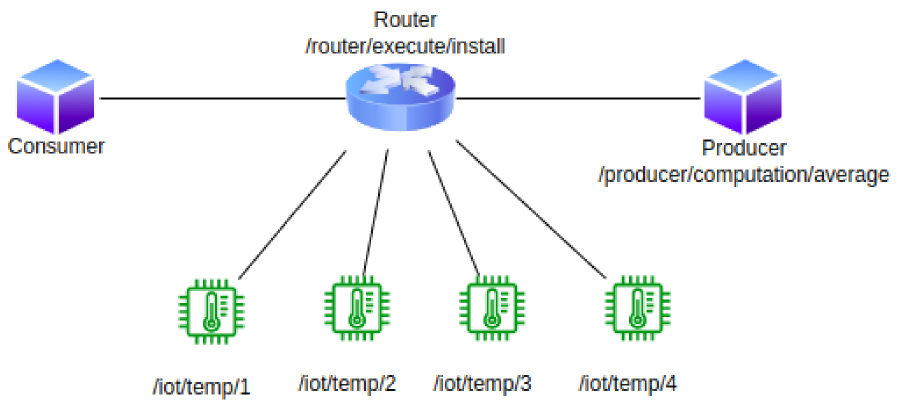


Figure 4.1: Test scenario

4.1 Protocol implementation

The framework is developed using ndn-cxx libraries[1] in version 0.8.0 . In the context of the protocol there are two parties that communicate, Producer and Consumer. The Producer is the entity that has executable functions and produces the result. The Consumer is the entity that requires remote execution.

The implementation is based on the simple C++ applications existing in ndn-cxx that implement NDN Consumer and Producer. The code to process protocol-defined TLVs was added to these apps. Also the Consumer application has been extended to send multiple Interests in the right sequence, register prefix, receive Data packets. The Producer application received support for sending Interest packets and running plugins. A NDN application that wants to use the protocol, has to include the developed Consumer code, create an instance of it and call methods to execute functions remotely. For example:

```
Consumer consumer;
string id = consumer.functionPrefix("/producer/computation/average");
consumer.setArguments(args);
consumer.execute(id);
```

All the code of the developed work can be found in the Github repository[2].

4.2 Functions implementation

The functions are implemented via a plugin system that makes use of C++ shared libraries that are dynamically loaded.

4.2.1 Plugin system

Common shared libraries are the initial step toward dynamically loadable libraries.

A collection of data and software code is called a shared library. Shared libraries are an excellent way to increase the functionality of C++ programs without having to completely rewrite them. They provide a set of exported variables, functions, and other symbols that can be used in another program as if the shared library's contents were directly integrated into it. With the concept of dynamic loading instead of having a predefined dependency on a library, its inclusion is optional, and the functionality of the application can be modified as a result. This eliminates the requirement to add libraries at the time of compilation in favor of loading them as needed during the runtime [12]. Shared libraries are Executable and Linking Format (ELF) files, essentially like a standard executable, with the exception that they typically lack a main() method as an entry point. Additionally, their symbols are organized so that any other program or library can utilize them as necessary in their own context.

As a rule, functions that are intended to be executed remotely must be done as a shared library in order to later allow them to load dynamically. Due to the design of the system, libraries are required to have a method called function() that does all of the processing intended for the function.

[1]`https://github.com/named-data/ndn-cxx`
[2]`https://github.com/AlexkononovV/ndn_remote_execution`

To create a shared library it is necessary to compile the C++ file as a position-independent object file, and then link it as a shared library. For example:

```
g++ -fPIC -shared function.cpp -o function.so
```

The prefix name is used to identify the function that will be executed. The name of the plugin to be loaded is inferred from the last name of the Interest name. After determining the plugin name from the prefix, the Producer loads this shared library dynamically at runtime. Once it has been loaded, it is possible to look for symbols there, extract them to pointers, and utilize them just like if the library had initially been linked. For this, **libdl** is available on Unix platforms.

The **dlopen()** function, which upon successful completion returns a generic pointer handle, is used to load the library. Then it is used **dlsym()** to locate and extract the "function" symbol from the library, passing the handle that was previously returned. dlsym() returns the address of the symbol as void *, if it is detected, that then can be allocated to a pointer type that represents the symbol. If everything goes well, it is conceivable to use the extracted function() just like it had always been there, and the result will be the same.

The following code is the snippet demonstrating dynamic loading of shared libraries:

```
void* handle = dlopen(so.c_str(), RTLD_LAZY);

std::string result;

if (!handle) {
    std::cerr << "Cannot open library: " << dlerror() << '\n';
}
else{
  // load the symbol
  typedef std::string (*function_t)(std::string);
  // reset errors
  dlerror();
  function_t function = (function_t) dlsym(handle, "function");
  const char *dlsym_error = dlerror();
  if (dlsym_error) {
      std::cerr << "Cannot load symbol 'function': " <<
      dlsym_error << '\n';
      dlclose(handle);
  }
  else{
    // use it to do the calculation
    result = function(args);
    // close the library
    std::cout << "Closing library...\n";
    dlclose(handle);
  }
}
```

Because functions are made as executables without the requirement for compilation and because they are loaded dynamically, it is possible to add new functions without having to recompile the Producer application.

4.2.2 Function for offloading

Support for instantiating functions on the network was created to make possible to offload some of the server's functionalities to the network. The primary goal is to push execution to the edge and reduce the latency.

The routers must possess the necessary computational power to carry out execution of functions.

Offloading is based on Docker containers. The creation of a Docker image is required by nodes that want to offload their functions to routers. This image is supposed to be a partial replica of the node's application. Hence, it is the Producer's responsibility to build a Docker image that will enable the instantiation of a container that mirrors the Producer's desired functionality.

The router in turn is supposed to run an application that can execute a function that instantiates containers.

Producers that want to offload their functionality to the router must perform remote execution of this function. The execution outcome is a container running on the router that will handle Interests sent to the Producer's prefix.

In terms of implementation, the instantiation function is another plugin, same as the other functions. The function first processes the input arguments, detects the image name, container instantiation options and possible input to the Docker container. The PStreams library[3] was used to invoke the docker run command by the function in order to instantiate the container. It is a simple library re-implementing popen() in C++. PStreams can read from the process' stderr as well as stdout. [3] Since it is intended to know the output after running the shell command, in this case docker run, it was made a decision to use this library. C++ native System() function for shell command execution does not allow getting output or error message after execution.

The outcome of the execution is a container running and performing Producer function(s).

Because the container assumes support for only part of the Producer's functionalities, the registered prefix will be longer, i.e. specifying supported function. For instance, the original Producer's prefix is /producer/computation, and it supports a number of functions. The prefix /producer/computation/sum will be registered by the container since the producer wishes to offload the sum function to the router. Default forwarding of NDN makes the forwarder receiving an Interest to match the longest prefix that is present in the FIB. Interest for /producer/computation/sum in the presence of the container is forwarded to the container rather than to the original Producer since the container has a longer prefix.

Figures 4.2 and 4.3 show a forwarding of Interest before and after function offloading.

[3]`https://pstreams.sourceforge.net/`

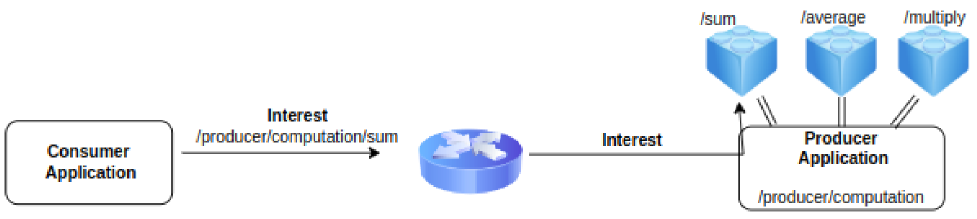


Figure 4.2: Before offloading

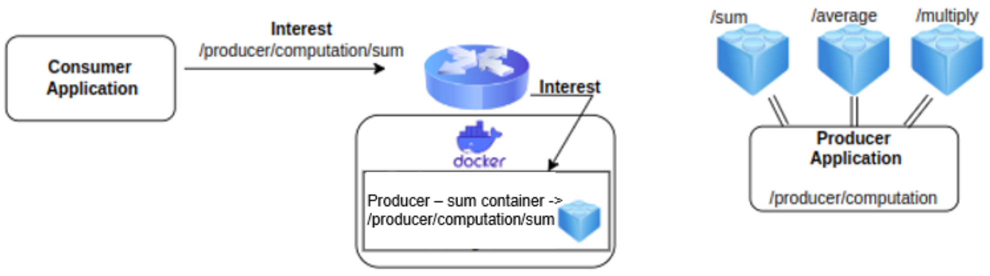


Figure 4.3: After offloading

4.3 Notify mode validation

The Notify operating mode of the protocol (described in Chapter 3 and illustrated by Figure 3.3) in some applications is better than Polling mode, especially in the case of remote execution to perform offloading, since the result of the execution is fetched right after the Producer has done the execution of the function and there is no overhead in terms of numbers of packets.

For the evaluation multiple temporal intervals were measured. The instant t0 is considered the moment at which the remote execution process starts. The first measured time interval, Tboot, is from the start of the process of the remote execution until the Consumer sends the first Interest (t1), i.e. Tboot=t1-t0. It is related to the boot time, which represents the time that Consumer takes to register its prefix and it includes some program overhead. The second interval, Trequest, is connected to the request time that lasts from the moment of sending the first Interest by the Consumer with the request for execution (t1) until the Producer starts executing (t2). In the implemented test scenario, the function tested produces results and implies the existence of input parameters and the Consumer is the bearer of these parameters. Thus, in this phase 2 packets are exchanged between the Consumer and the Producer in the case when input is explicit and 4 packets in the case of implicit input. The third interval, Texec, is related to the execution time, which represents the time that lasts from the start of the execution of the function in the Producer (t2) until the execution ends (t3), i.e. Texec = t3-t2. Immediately after completing the execution of the function, the Producer sends the Interest to the Consumer notifying the availability of the result. The instant at which the Consumer receives the notification, in the Interest, from the Producer is registered (t4). As the request phase is measured in the Consumer, it is difficult to detect the start of execution of the function in the Producer , so the duration of the request phase was indirectly obtained as Trequest=t4-t1-(t3-t2). Thus, it includes round trip times existing in the sending of the notification. The fourth time interval, Tretrieval is associated to the retrieval time. It starts when the Consumer receives the notification from the Producer (t4) and ends when the Consumer receives the final data that contains the result of the execution (t5). Four packets are exchanged in this phase. Therefore, Tretrieval = t5-t4. The fifth measured time is the total completion time, i.e. Tretrieval = t5-t0.

Table 4.1 shows the duration of each phase of the remote execution in Notify mode, represented by the mean with 95% confidence interval

Notify mode				
Explicit input				
Boot time(ms)	Request time(ms)	Execution time(ms)	Result retrieval time (ms)	Total completion time (ms)
mean and 95% confidence interval				
3.14 (3.06, 3.22)	4.61 (4.53, 4.69)	18.65 (18.42, 18.88)	2.41 (2.31, 2.50)	29.86 (29.57, 30.15)
Implicit input				
Boot time(ms)	Request time(ms)	Execution time(ms)	Result retrieval time (ms)	Total completion time (ms)
mean and 95% confidence interval				
3.38 (3.26, 3.50)	7.21 (7.07, 7.34)	39.07 (38.70, 39.44)	2.29 (2.19, 2.39)	52.24 (51.80, 52.67)

Table 4.1: Notify mode evaluation.

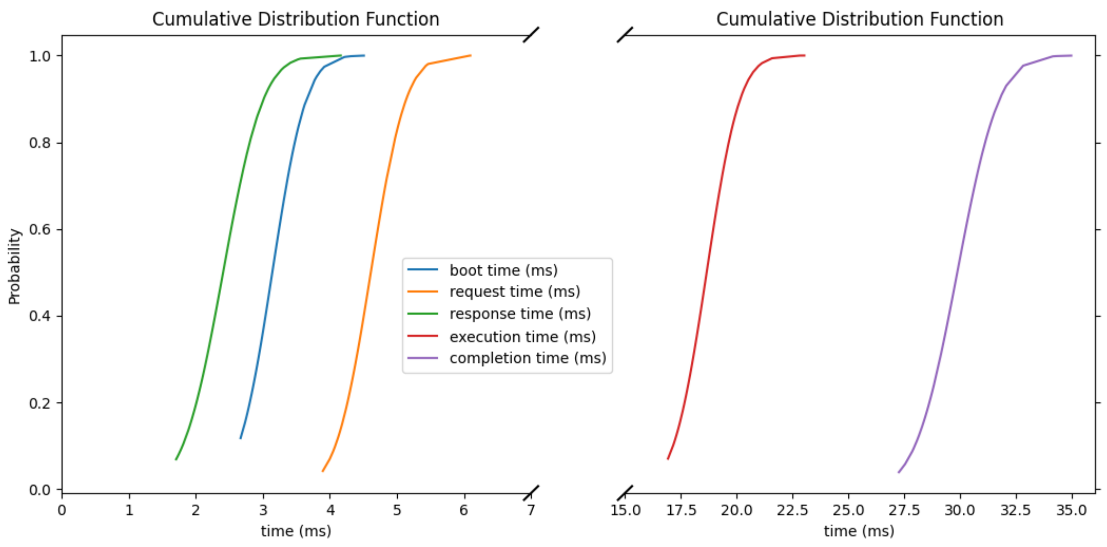


Figure 4.4: CDF of different phases of execution in Notify mode with explicit input.

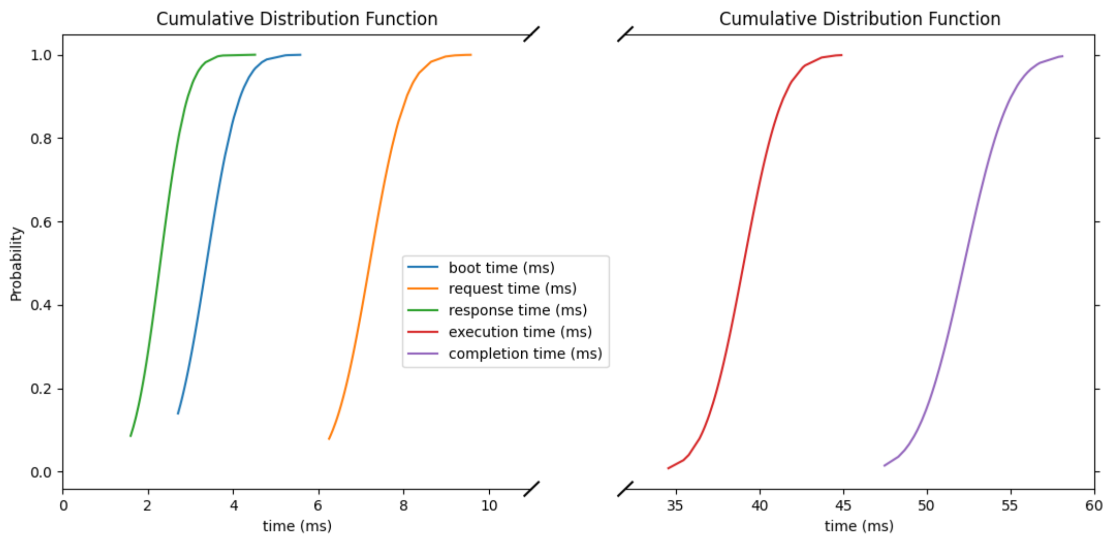


Figure 4.5: CDF of different phases of execution in Notify mode with implicit input.

Figure 4.4 has graph with Distribution Function (CDF) for each step of the execution in Notify mode with explicit input. It is visible that the boot time takes less than 4 milliseconds (ms) in the most cases. The request time takes less than 5 ms with more than 80% probability. The execution time is highly possible to be below 20 ms. The result retrieval time is likely to be between 2 ms to 3 ms. And the completion time is usually below 33 ms. Figure 4.5 has CDF graph in case of implicit input. The boot time and the retrieval time are similar to the times of the execution with explicit input, as expected. The request time is mostly below 8 ms. The execution time value is likely to be around 40 ms. And most of the times the total completion time is less than 55 ms.

From an analysis of the table and the graphs is possible to observe that the initialization phase represents some small overhead, related to registering the Consumer prefix.

The request phase is longer in the case of implicit input which is justified by the fact that there are 2 more packets exchanged for passing input parameters, increasing the request time in more than 50% compared to the executions with explicit input.

The result fetching phase time has similar duration in both cases as the process of result fetching is exactly the same.

The biggest variation is in the execution time of the function. Two input arguments were passed for the testing of explicit input, while seven input arguments were passed for the tests of implicit input.

To summarize, it is inferred that the execution time of the function has the greatest impact on the completion time in Notify mode. This time depends on the complexity of the function, on the size of the task and on the available resources of the hardware where it is executed.

4.4 Polling mode validation

The Polling mode description and use cases for its applications are described in Section 3.1 and its working principle is illustrated in Figure 3.2.

To evaluate this mode tests were made with explicit input.

An attempt was made to find the optimal polling value to obtain the lowest completion time.

The Average function was modified for this tests and a 10ms delay was added. Thus, the average function execution time became 19ms.

The plot on Figure 4.6 shows the function's execution time, the total number of packets exchanged and the total completion time as a function of the pooling interval on the Consumer.

After analyzing can be concluded that there is no ideal polling interval. When the polling interval is very small, there is a big overhead in terms of exchanged packets, but on the other hand the completion time is quite low because the result is collected almost as soon as it is available. In general, when the polling interval is longer, fewer packets are exchanged but it takes longer for the process to finish. This makes sense given that when the poll is done shortly before the execution concludes, the subsequent poll is done only at the next time. Thus, the longer the polling interval is, the longer is the delay at receiving the results.

However, if the polling interval becomes closer to the average execution time, the occurrence of situations when the Consumer gets the result in the first poll starts to happen more frequently. It is explained by the fact that the function can occasionally execute more quickly than usual and the Interests of first poll are able to detect the end of execution and gather the results before the second poll happens. This explains the drop in the total completion time for polling intervals close to the execution time.

It is also not recommended to set the polling interval equal to the execution time. If the execution takes a bit longer than average, there will be sent an additional Interest and there will be a delay that is at least equal to the polling interval. In other words, the fetching of the result in this scenario would take almost twice as long as the execution time. Even though the completion time increases as the polling interval is increased, the number of transferred packets decreases until it becomes constant.

To define the best polling interval it is necessary to know the average execution time and packet propagation delays. One depends on the available resources of the hardware where it is running and another on network delays that depend on network state. Outside the scope of this work, it would be worth developing a mechanism that would be able to calculate the most efficient polling interval value taking into account the mentioned factors. The performance of polling mode would be greatly enhanced with this mechanism.

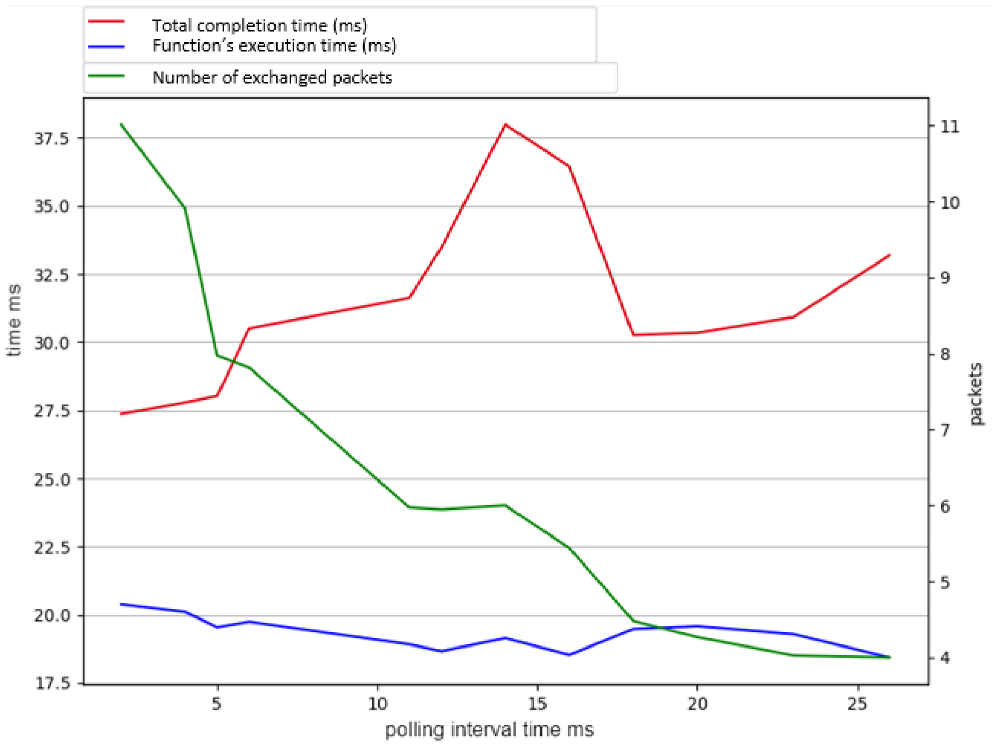


Figure 4.6: Polling mode measurements

4.5 Function offloading evaluation

To validate the offloading of functions on the routers, the Producer application will ask the router to instantiate one of its function, namely the Average function. To do this, a Docker image will be made that replicates Producer's behavior and contains a plugin for the Average function. The router application will create Docker container that will run this Docker image. The local forwarder of the router will receive the prefix registration from the containerized Producer, which will cause the Interests requesting average function to be forwarded to the container. This means that while Interests for /producer/computation/average will be forwarded to the container, Interests with less specific prefixes (such as /producer/computation) or for other functions (like /producer/computation/max) will be forwarded to the remote Producer.

Offloading is done through remote function execution under prefix /router/execute/install by Producer. The implementation of this function is described in Section 4.2.

Making the Producer Docker image is the first thing to do before calling the function. Since this action is not a part of the solution, it is the Producer's responsibility to ensure that the appropriate Docker image is accessible through the Docker Hub repository.

First, the performance of container instantiation was evaluated. It is necessary to pull the docker image from the repository by running the docker pull command whenever the Router does not have required Docker image. The image pulling was not considered in the analysis because the amount of time required to download is dependent on both the Internet speed and the size of the image.

Table 4.2 displays the results of the offloading performance, assuming that the Producer Docker image already exists in the Router.

The offloading function was evaluated using Notify mode, which is considered more efficient for this use case. To address each instantiation phase four time intervals were measured. The first interval, $T^{offloading}{}_{total}$ is the total completion time of remote execution, it is the Completion time measured in the Producer that requested instantiation of its function. It lasts from the moment of sending first Interest requesting execution of instantiation function in the Router until receiving the final Data from the Router. The second interval, $T^{offloading}{}_{exec}$, is the total execution time of the instantiation function in the Router. It begins when the install function in the Router starts executing and ends when the execution of the function terminates. The third interval, $T^{offloading}{}_{instantiation}$, is the container instantiation time, a sub-interval of the previous, and it is the amount of time it takes to perform the docker run command by the instantiation function in the Router. The last interval, $T^{offloading}{}_{register}$, measures time that takes the registering of prefix of the container in the local forwarder, i.e. the time it takes between the container being ready and being accessible by the registered prefix.

The results of the time analysis show that the Container is created on average in 232 milliseconds, with the time depending on the complexity of the container and the available hardware resources. Because the instantiation function parses input parameters to determine container instantiation options, checks for image presence locally and incurs additional procedural-related delays, it has some temporal overhead and in average the total function execution time takes 439 milliseconds.

Then, it was evaluated the gain in total completion time with offloading. Table 4.3 demonstrates the results.

In the case of the notify mode the completion time, which is related to response time, is reduced by about 10% while in polling mode it is decreased more than 20%.

Prefix registration(ms)	Container instantiation time(ms)	Total execution time of the function(ms)	Total completion time(ms)
mean and 95% confidence interval			
3.46 (2.96, 3.96)	232.93 (217.24, 248.62)	439.26 (427.37, 451.16)	584.26 (572.95, 595.57)

Table 4.2: offloading measurements.

Notify mode				Polling mode			
Explicit input		Implicit input		Explicit input		Implicit input	
Before offloading(ms)	After offloading(ms)	Before offloading(ms)	After offloading(ms)	Before offloading(ms)	After offloading(ms)	Before offloading(ms)	After offloading(ms)
mean and 95% confidence interval							
175.8 (173.6, 177.9)	153.9 (139.7, 168.0)	217.7 (214.2, 221.1)	195.5 (193.4, 197.5)	59.6 (57.9, 61.2)	34.0 (33.3, 34.6)	229.4 (221.3, 237.4)	168.3 (165.1, 171.4)

Table 4.3: Offloading advantage .

4.6 Summary

The test scenario for validating the created framework was described in this chapter. The function utilized for offloading was explained, and the implementation of the functions in general was covered in depth. Two operational modes were examined, and the results were discussed. The different stages of the execution process and the factors that have an impact on the overall completion time were identified in the Notify mode. The evaluation of the Polling mode provided the finding that the performance of the mode depends on the value of the polling interval that must be properly defined. The effectiveness of offloading and the improvements to remote execution's response time were demonstrated by the performed evaluation.

Intentionally blank page.

Chapter 5

Conclusions

The Internet of today is dynamic and faces new challenges that call for fresh approaches. The IP architecture currently used, fails in many things, such as lack of efficient congestion control, lack of security at the architectural level, the way the content distribution supported does not correspond to the existing scale, lack of efficient cache, support for problem diagnosis and others. The discussion of these problems leads to examine the present IP architecture and future networking paradigms, such as ICN. One of the ICN paradigms' functional visions is NDN, that includes features like in-network caching, multi-path forwarding, built-in security, named data routing, scalable forwarding, and others. It offers infrastructure support for serverless application design and natively enables data-centric information distribution. All of these characteristics offer solid advantages over the present Internet. NDN, despite being a potential solution that addresses the most of the present Internet's issues, it still has significant drawbacks and challenges that need to be solved.

One of the significant network issues at the moment is the massive amounts of data that need to be processed at the edge given the current IoT boom. In-network computing and remote execution are two needs for this. As ICN does not have the built-in support for this, there is a search for a workable solution to address this gap.

In this work a framework for remote execution and function offloading in NDN was created to help overcome this gap. One of the framework's goals is to enable remote execution in NDN applications in an easy way without requiring network layer changes. Another goal is to employ remote execution to offload network functions by instantiating containers on routers that would emulate the behavior of data publishers, bringing computing closer to the edge. One of the peculiarities of the proposed design is its ability to support new ways for delivering input parameters to functions and new techniques for gathering the execution result. The full implementation of the suggested approach was described. It was created a scenario through which it was demonstrated the framework's feasibility. Moreover, it was showed that function offloading at the edge improves response time, since the computation is moved closer to the clients.

For future work, it will be necessary to address the security and data authentication in the remote execution and instantiation of functions, as well as create a mechanism for servers to be able to determine the best location at the edge to instantiate their functions according to demand. Also, a method for automated docker image creation for producers who want to offload their functions to the network must be developed.

Intentionally blank page.

Bibliography

[1] NDN Protocol Design Principles. `https://named-data.net/project/ndn-design-principles/`.

[2] Network Virtualization, . `https://www.vmware.com/topics/glossary/content/network-virtualization.html`,.

[3] POSIX Process Control in C++. `https://pstreams.sourceforge.net/`,.

[4] Producer Mobility Support for Information Centric Networking Approaches: A Review. `https://www.researchgate.net/figure/IP-and-NDN-CCN-Hourglass-Architecture-32_fig2_323999288`.

[5] What is IoT Edge computing?, 2022. `https://www.redhat.com/en/topics/edge-computing/iot-edge-computing-need-to-work-together`,.

[6] Amar Abane, Paul Muhlethaler, Samia Bouzefrane, Mehammed Daoui, and Abdella Battou. Towards evaluating Named Data Networking for the IoT: A framework for OMNeT++. 09 2018.

[7] Alexander S. Gillis. What are containers (container-based virtualization or containerization)?, 2020. `https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization`.

[8] Andy Patrizio. Edge computing in 2022: Double-digit growth, 2022. `https://www.networkworld.com/article/3648032/edge-computing-in-2022-double-digit-growth.html`,.

[9] Gary Chen. The Role of Virtualization in the Era of Containers and Cloud. Sponsored by : Red Hat. 2018.

[10] Docker Docs. Docker Overview. `https://docs.docker.com/get-started/overview/`.

[11] I.C. IBM Cloud Education. Docker, 6 2022. `https://www.ibm.com/cloud/learn/docker`,.

[12] Gregori, Sven. It's all in the libs – building a Plugin System Using Dynamic Loading, , 2018. `https://hackaday.com/2018/07/12/its-all-in-the-libs-building-a-plugin-system-using-dynamic-loading/`, Last accessed on 2022-10-10.

[13] Muktar Hussaini, Shahrudin Awang Nor, and Amran Ahmad. Producer Mobility Support for Information Centric Networking Approaches: A Review. *International Journal of Applied Engineering Research*, 13, 03 2018.

[14] International Data Corporation (IDC). IDC Multicloud Management Survey, 2019. Special Study, US45020919.

[15] Divya Kapil, Parshant Tyagi, Sonu Kumar, and Vinay Prasad Tamta. Cloud Computing: Overview and Research Issues. In *2017 International Conference on Green Informatics (ICGI)*, pages 71–76, 2017.

[16] Brian Kirsch Kate Brush. Virtualization, 10 2021. `https://www.techtarget.com/searchitoperations/definition/virtualization`.

[17] Michał Król, Karim Habak, David Oran, Dirk Kutscher, and Ioannis Psaras. RICE: Remote Method Invocation in ICN. ICN '18, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[18] Michał Król and Ioannis Psaras. NFaaS: Named Function as a Service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, ICN '17, page 134–144, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Ge Ma, Zhen Chen, Junwei Cao, Zhenhua Guo, Yixin Jiang, and Xiaobin Guo. A tentative comparison on CDN and NDN. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2893–2898, 2014.

[20] Nitin Nagar and Ugrasen Suman. Analyzing Virtualization Vulnerabilities and Design a Secure Cloud Environment to Prevent from XSS Attack. *International Journal of Cloud Applications and Computing*, 6(1):1–14, 1 2016.

[21] NFD. NDN site. `https://named-data.net/doc/NFD/current/overview.html`.

[22] NLSR. NDN site. `https://github.com/named-data/NLSR`.

[23] Noa Zilberman . In-Network Computing, 4 2019. `https://www.sigarch.org/in-network-computing-draft/`.

[24] Benjamin Rainer and Stefan Petscharnig. Challenges and Opportunities of Named Data Networking in Vehicle-To-Everything Communication: A Review. *Inf.*, 9:264, 2018.

[25] Divya Saxena. Named Data Networking: A Survey. *Elsevier Computer Science Review*, 01 2016.

[26] Narendra Rao Tadapaneni. Different Types of Cloud Service Models. In *SSRN*. 2017.

[27] Qi Zhang, Ling Liu, Calton Pu, Qiwei Dou, Liren Wu, and Wei Zhou. A Comparative Study of Containers and Virtual Machines in Big Data Environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 178–185, 2018.