In-network Computation for IoT in Named Data Networking

Preechai Mekbungwan

Doctor of Science Thesis
Sorbonne Universités

Specialization

Computer Science

presented by

M. Preechai Mekbungwan

Submitted in partial satisfaction of the requirement for the degree of

Doctor of Science of Sorbonne Universités

In-network Computation for IoT in Named Data Networking

Committee in charge:

Silvia Mirri	**Reviewer**	**Associate Professor, University of Bologna**
Marco Zennaro	**Reviewer**	**Research Scientist and Coordinator of the Science, ICTP**
Isabella Annesi-Maesano	**Examiner**	**Research Director, Université de Montpellier**
Serge Fdida	**Examiner**	**Professor, Sorbonne Université**
Kanchana Kanchanasut	**Examiner**	**Professor, Asian Institute of Technology**
Giovanni Pau	**Supervisor**	**Professor, University of Bologna**

Acknowledgement

Foremost, I would express my gratitude to my thesis supervisor Prof. Giovanni Pau for his supervision with immense support, time and patience during all this difficult time. I am truly grateful to Prof. Kanchana Kanchanasut for her patience, and invaluable time that she gave me for my research with valuable guidance and supervision, which were absolutely crucial for the successful completion of this thesis. I would like to thank Dr. Adisorn Lertsinsrubtavee for his insightful advice, encouragement and moral support during hard times of my PhD.

Furthermore, I would like to thank Prof. Serge Fdida and Emilie Mespoulhes so much for endless supports in administrative issues at LIP6. Special thanks to Dr. Adisorn Lertsinsrubtavee and Prof. Sukumal Kitisin for their valuable time for proofreading and corrections of my thesis manuscript. Their invaluable comments and enlightening suggestions have significantly improved the quality of the manuscript. To Nisarat Tunsakul, Dr. Adisorn Lertsinsrubtavee and Nunthaphat Weshsuwannarugs at the IntERLab for the supporting on my testbed deployments in Thailand. And to Dr. Boris Dessimond for the French translation in my thesis.

I would like to acknowledge Prof. Silvia Mirri and Dr. Marco Zennaro for accepting to be the reviewers of this thesis, as well as Prof. Kanchana Kanchanasut, Prof. Serge Fdida and Prof. Isabella Annesi-Maesano for being on my dissertation defence committee. Their invaluable comments and enlightening suggestions have helped improve the quality of this thesis. And, I'd like to express my gratitude again to Prof. Kanchana Kanchanasut, Prof. Giovanni Pau, Prof. Isabella Annesi-Maesano and also the Franco-Thai Scholarship Program for giving me this opportunity to pursue my PhD.

I must thank to friends at LIP6: Luca De Mori, Davide Aguiari, Andrea Ferlini, Emilie Mespoulhes, Boris Dessimond, Furong Yang, Wei Wang, and Albert Su, I wouldn't have enjoyed my life in France without them. Also, I thank to all friends from IntERLab that have been generously supporting me all along many years. Furthermore, I may miss some names, I just wanted to say thank you for all small and big details, specially for the lifetime you all spent on me along these years.

Finally, I am sincerely grateful to my parents and my younger brother for their endless love, patience and encouragement.

Abstract

With intermittent Internet connectivity condition, traditional IoT deployment cannot provide prompt real-time data analysis and responses to on-site users because steady connection to the cloud cannot be achieved. In-network computation allows application functions to be computed within the network directly on raw sensor data, and publish real-time responses or alerts to users in the field. ActiveNDN is proposed to extend Named Data Networking (NDN) with in-network computation by embedding functions in an additional entity called Function Library, which is connected to the NDN forwarder in each NDN router. Function calls can be expressed as part of the Interest names with proper name prefixes for routing, with the results of the computation returned as NDN Data packets, creating an ActiveNDN network. Our main focus is on performing robust distributed computation, such as analysing and filtering raw data in real-time, as close as possible to sensors in an environment with intermittent Internet connectivity and resource-constrained computable IoT nodes. To deploy ActiveNDN for IoT in wireless networks, the simple wireless broadcast with the NDN forwarding mechanism is used to perform recursive function calls and to aggregate results on the return paths. In wireless networks, inaccuracies in the computation results can occur due to packet collisions where many nodes may happen to send packets at the same time, causing packets to lost. In this context, three mechanisms are proposed, which include: randomizing aggregation window sizes to avoid packet collision, retransmission of Interests to overcome the effect of packet loss and using Interest exclude field to reduce traffic congestion. In this thesis, the design of ActiveNDN is illustrated with a small prototype network as a proof of concept. Extensive simulation experiments were conducted to investigate the performance and effectiveness of ActiveNDN in large-scale wireless IoT networks. The real-time processing capability of ActiveNDN is also compared with centralized edge computing approaches. Finally, the ActiveNDN is demonstrated over the wireless sensor network testbed with real-world applications that provide sufficiently accurate hourly PM2.5 predictions using linear regression model. It shows the ability to distribute the computational load across many nodes, which makes ActiveNDN suitable for large-scale IoT deployments.

Résumé

En raison de l'intermittence de la connectivité internet, le déploiement traditionnel des technologies de l'internet des objets (IoT) n'est pas en capacité de fournir une analyse rapide des données en temps réel aux utilisateurs présents sur le terrain lorsqu'une connexion stable au cloud ne peut être réalisée.

Le calcul dit "In-network" permet aux fonctions applicatives d'être calculées au sein du réseau directement sur les données brutes des capteurs, et de publier des réponses ou des alertes en temps réel aux utilisateurs présents sur le terrain.

"ActiveNDN" est proposé pour étendre le réseau de données nommé (NDN) avec le calcul in-network ; en intégrant des fonctions dans une entité supplémentaire appelée bibliothèque de fonctions, qui est connectée au transporteur NDN dans chaque routeur NDN.

Les appels de fonction peuvent être exprimés comme une partie des noms d'Interest avec des préfixes de dénomination propres pour le routage. Les résultats des calculs sont renvoyés comme des paquets de données NDN, créant ainsi un réseau ActiveNDN.

Nous nous focalisons principalement sur l'exécution de calculs décentralisés robustes, tels que l'analyse et le filtrage de données brutes en temps réel, réalisés aussi près que possible des capteurs, dans un environnement avec une connectivité Internet intermittente, avec des nœuds IoT aux ressources limitées.

Pour déployer ActiveNDN pour l'IoT dans les réseaux sans fil, une simple diffusion sans fil avec un mécanisme de transfert NDN est nécessaire. La diffusion avec le mécanisme de transfert NDN est utilisée pour effectuer des appels de fonction récursifs et agréger les résultats sur les circuits de retour.

Dans les réseaux sans fil, des inexactitudes au niveau des résultats de calcul peuvent se produire en raison de collisions de paquets, lorsque de nombreux nœuds peuvent envoyer des paquets en même temps, entraînant la perte de paquets. Dans ce contexte, trois mécanismes sont proposés, à savoir : la randomisation de la taille des fenêtres d'agrégation pour éviter la collision des paquets, la retransmission des requêtes pour pallier à l'effet de la perte des paquets ainsi que l'utilisation du champ d'exclusion des requêtes pour réduire la congestion du trafic. Dans cette thèse, la conception d'ActiveNDN est illustrée par un petit réseau prototype comme preuve de concept.

Des expériences de simulation poussées ont été menées pour étudier performance et l'efficacité d'ActiveNDN dans les réseaux IoT sans fil à grande échelle. Les capacités de

traitement en temps réel d'ActiveNDN sont également comparées aux approches centralisées de calcul périphérique.

Enfin, l'ActiveNDN est appliqué à un banc d'essai de capteurs sans fil connectés, une application au monde réel fournissant des prédictions horaires de particules en suspension dans l'air (désignées comme PM2.5) suffisamment précises en utilisant un modèle de régression linéaire. C'est une démonstration concrète de la capacité à distribuer la charge de calcul sur de nombreux nœuds, rendant donc ActiveNDN adapté aux projets de déploiements à grande échelle d'objets connectés.

Table of Contents

List of Figures

List of Tables

1 Introduction

Many environmental monitoring applications, especially an emergency warning system, are used to provide real-time or advance warning to people in affected areas as well as to the relevant authorities. Real-time data is collected via sensors in these areas. Network connectivity is required for such IoT monitoring applications. However, in many places, Internet connectivity is unavailable or often interrupted when a disaster strikes. This is especially true for monitoring applications in remote areas, such as forests, where sensors are used to monitor fires. In such an environment, even an Internet connection may not be available. Therefore, the deployment of such systems is usually based on wireless sensor networks (WSN).

The main challenges of the current architecture of IoT monitoring applications are interrupted Internet connection, unstable deployment, and limited computing power and power consumption of the devices. Typical IoT systems with Internet infrastructure collect data and upload it to the cloud servers for processing. Later, some systems pre-process the collected data on the local or Fog servers and then send the information to the remote Cloud. This system architecture may not be suitable for emergency alerts or environmental monitoring systems. First, using a centralized cloud-based system may be ineffective in a situation where Internet connectivity is interrupted because the system cannot collect data and thus cannot warn or inform affected individuals in real time. Second, Fog computing uses edge devices to perform computation and storage, which then communicate with the cloud. This moves computing capacity to the edge of the network, closer to the IoT devices, which means lower latency for the service. Therefore, these edge servers need to be placed in a remote area to perform computations locally. However, installing the devices in remote communities may not be sustainable due to the collective power costs, the required IT capabilities, and the responsibility of maintaining the servers that rest on a few people in whose homes the servers are installed. Therefore, it is impractical to deploy such an architecture. Third, without a more powerful edge server, an IoT device with limited computing power cannot perform computationally intensive tasks. Therefore, a lightweight distributed computing model is required.

To address these issues, [Heidemann et al., 2001] describes in-network processing that enables data aggregation at intermediate nodes in the network to provide real-time responses. The model is later referred to as in-network computation in [Giridhar & Kumar, 2006], where nodes in the network not only forward the collected data but also perform

general computations.

Such a network becomes a network of computing nodes where data can be processed closer to the source and users can receive timely, real-time responses and take appropriate action on the network itself. In addition, network traffic can be reduced because raw data is processed as it is routed within the network, and only the aggregated informative results need to be transmitted as needed. In-network data processing has become more important with the advent of time-critical processing applications. One of these is the Extended Reality (XR), which is a combination of Virtual Reality (VR) and Augmented Reality (AR). Real-time processing of localized information from multiple sensors and video feeds from cameras can be achieved using the in-network data processing [Montpetit, 2019].

Currently, embedding computation on nodes in a network using the traditional host-centric Internet protocol is performed as web services or microservices. However, a user's request for a service must be dynamically resolved to the IP address of the node executing the requested services. While the existing DNS (Domain Name System) can resolve domain names to host addresses, but the complexity in mapping service locations and protocol layers makes it infeasible and unsuitable for time-critical applications. In addition, the architecture can neither handle network disruptions nor mobility of the nodes.

On the contrary to the host-centric protocol, Named Data Networking (NDN) was proposed by [Zhang et al., 2014] as an implementation of the Information-Centric Networking (ICN) paradigm to network computing devices, ranging from IoT sensors, [Amadeo et al., 2014a], small single-board computer devices, laptops, PDAs, to cloud servers, using names. Instead of using an IP address to identify a host, NDN uses hierarchical naming to address a piece of content so that it can be retrieved without a concern on an endpoint location or the network topology. This makes NDN an ideal candidate for in-network computation. NDN's hierarchical naming scheme eliminates the complexity and overhead of mapping content to a host's IP address that traditional in-network computation schemes entailed. Thus, NDN satisfies the characteristics and requirements of IoT applications [Shang et al., 2017] by providing communication, in-network caching, mobility functions, and security.

In this work, we propose to augment NDN with sufficient computational capabilities on IoT sensor nodes to perform the in-network computations. The network is assumed to be composed of sensor nodes that have sufficient power supply, adequate local area network, and limited computational and storage capacity, while the internet connectivity may be intermittent or may not be available at all. When deploying IoT-based environmental monitoring applications for rural communities, a fair distribution of the local data processing computing load must be achieved. In such an environment, each IoT node with a computing unit can perform the necessary computations, such as some basic statistical or mathematical functions. Then the units can be orchestrated as building blocks to form a complex application.

1.1 Motivation

As natural disasters have hit the world with increasing frequency recently, the need for real-time environmental monitoring and emergency warning systems is increasing. These real-time IoT applications enable people to prepare and respond appropriately to manage an emergency situation more effectively; thus, they help mitigate the impact of the incidents. However, there are challenges to implementing and deploying such a system, as in many cases the system must be deployed in remote areas where unstable Internet connectivity, maintainability of equipment, and practicality of deployment are issues.

In most remote areas, Internet connectivity is unavailable or may be disrupted due to natural disasters. Using a cloud-based IoT system may be ineffective because the cloud server may not be connected to the IoT sensors, so it cannot send an alarm or immediately notify people in the area of incidents that are occurring. Therefore, the system must be able to process data locally without relying on an Internet connection, and it must allow people in the field to receive the alert information in real time.

The deployment of the system requires the cooperation of local residents. Residents must be made aware of the benefits of the system to the entire community. However, the cost of maintaining the equipment is a problem. This is a major issue for residents in the deployment area, especially in rural communities where household income is minimal. Setting up a data center or server is impractical because it can be costly to maintain, and therefore may not be supported by residents. Ideally, the deployment of system devices in a community needs to be equitably distributed among residents. Then the deployment will be sustainable. To provide a local IoT computing service with minimal maintenance responsibility that is distributed equitably among community residents, homogeneous nodes of low-cost, sensor-equipped computing units can be deployed in each home in the target community. This allows for a distributed computing mechanism among these nodes, and maintenance costs are shared equitably among the residents of the deployment area.

Finally, distributed data processing among low-cost computing units is unlikely to enable high-precision computation or idempotent functions. However, the data processing tasks of a real-time environmental monitoring or emergency warning system do not require intensive computational loads or stateful computations. Therefore, the design of the distributed computing platform for the monitoring system can be sufficiently lightweight for the low-cost computing units.

1.2 Problem Statement

In remote rural areas, the IoT real-time incident warning system in the cloud or centralized architecture is not suitable to perform the calculations for sensor data due to insufficient Internet connectivity and maintenance cost in the community.

Instead, the calculations can be performed in a decentralized manner by the locally connected sensor nodes. There is a need for in-network computing technology that enables sensor nodes to perform real-time distributed processing to analyse sensor measurements within the network. However, due to the complexity of the mapping process between data and host, in-network computation using the conventional host-centric Internet protocol is not efficient. Named data networking (NDN) can be extended with the proposed mechanisms to provide name-based routing for forwarding a function call request to the desired function program on the closest node within a wireless sensor network.

The existing distributed computing platforms in Information-Centric Networking (ICN), such as Named-Function Networking (NFN) [Tschudin & Sifalakis, 2014], RICE [Król et al., 2018], and Compute First Networking (CFN) [Król et al., 2019], are designed for generic computing with deterministic computation model. However, they are not suitable for IoT network computation. NFN includes an expression evaluator in the forwarder mechanism to embed a programming expression in the Interest name. RICE provides naming and mechanisms to allow large program input parameters that may not be able to include in the Interest name, e.g., image or video files, and to support long-running computations. CFN provides a distributed scheduler to offload computations to the inactive node by synchronizing node resources and scheduling information. These features come with additional computation and network overhead tradeoffs.

In the IoT-based alarm systems, where only primitive sensor data is collected and processed with simple numerical calculations. The complex functions mentioned above, such as name expression evaluation, large function input processing, and job schedulers, are not required for the systems and are too cumbersome to operate on these resource-constrained IoT devices.

In summary, this thesis addresses a design of an IoT network that processes local data and provides prompt, real-time responses to requests from users in the field while relying on the limited computing power of the network.

1.3 Contributions

In this study, ActiveNDN is proposed to enable real-time distributed computation on IoT devices with a simple NDN forwarding mechanism extended with an in-network computation mechanism to share computations among IoT nodes. ActiveNDN is designed to provide computation programs as functions with minimal changes to the native NDN. Each function can be called from the network to execute it and return the result via primitive NDN messages. A running function can call another function locally or remotely on other nodes. The subsequent remote function call allows the computation to be distributed across nodes, spreading the computation across the network. With this lightweight design makes ActiveNDN backward compatible with native NDN while providing computation capabilities.

In this work, several key challenges have been addressed to enable computation in IoT WSN. Here, we focus on three of the above challenges: 1) intermittent Internet connectivity, 2) the fair distribution of computational load. 3) the limited computational power of IoT devices. Based on these constraints and requirements, ActiveNDN is proposed to provide a locally distributed computing platform with in-network computation over the NDN protocol for the network of IoT devices.

ActiveNDN is designed to use basic NDN protocol primitives to forward function calls and results and distribute computation tasks in an IoT network. To enable communication, the Function Library is introduced to store and execute function programs in ActiveNDN nodes. To deal with wireless network issues, three mechanisms are proposed, including using a random aggregation window to avoid packet collisions, using retransmission to recover from packet loss, and using Exclude field to reduce network congestion. As opposed to a centralized system, ActiveNDN is proposed as an in-network computing framework for the IoT in WSN to process data within the network itself, reduce network traffic, and decrease response latency.

ActiveNDN has been implemented and demonstrated to enable in-network computing in physical IoT networks. It is demonstrated with real-world applications to perform real-time data analysis for air quality monitoring in a real system deployment in a rural village.

1.4 Thesis Outline

The remaining chapters of this thesis are organized as follows: Chapter 2 provides a comprehensive literature review of concepts and technologies essential to ActiveNDN, as well as a detailed review of related work.

Then, Chapter 3 explains the design, architecture, and mechanisms of ActiveNDN with the Function Library component to provide computation capability on NDN network. As an example, a simple function program called *FindMax* is used to find the maximum sensor value in the entire network.

Chapter 4 addresses the issues affecting computational accuracy in wireless sensor networks and describes the proposed ActiveNDN mechanisms for wireless networks. Then, the proof of concept is illustrated and the experiment and analysis results are presented.

In Chapter 5, the performance of the in-network computation model of ActiveNDN and the centralized computation model are investigated and compared with simulations and testbed experiments. A complex ActiveNDN calculation is also demonstrated with a PM2.5 prediction application using linear regression analysis. The prediction accuracy is validated in a real wireless sensor network.

Finally, a summary of the main conclusions of this thesis is provided in Chapter 6, which lists the most relevant parts of this thesis that serve as motivation for future work in this research area.

2 Literature Review

Our main goal in this thesis is to develop a computing platform for the Internet of Things that allows local users to process sensor data in real time without relying solely on Internet connectivity. In-network computation is a distributed computing technique that allows the computation to be performed in the network. Instead of only forwards the data, the network nodes can also be programmed to process the data before forwarding and thus the network traffic and network latency can be reduced. However, implementing this technique on the traditional internet protocol is not practical as the communication is still relying on host addresses. A dynamic naming resolution is needed to tell the address of the nearest node where the related service is available; however, it will create more complexity and overhead to the system.

On the other hand, an Information-centric networking (ICN) is introduced to address the naming issue in the conventional protocol. The network does not use an address of the host where the resource is provided, but it uses a naming to identify the resource itself. The ICN is independent of the endpoint location or network topology and thus make it a potential candidate networking protocol for the in-network computation.

The goal of this chapter is to provide a comprehensive review of Internet of things architectures, In-network computation and Information-Centric Networking. The methodology we adopt in this chapter is to first discuss the architecture in Internet of Things in section 2.1. Consequently, In-network computation is introduced and the comprehensive details of Information-Centric Networking is provided in section 2.2 and 2.3, respectively. Then, the research works on distributed computation in ICN is presented and discussed in section 2.4. Finally, all related works are synthesized, and the thesis approach is pointed out in section 2.5.

2.1 Internet of Things

Internet of Things (IoT) is a computing paradigm where IoT devices, i.e. physical objects embedded with sensors, actuators and processing capability, are connected to the Internet, allowing them to exchange information and coordinate among each others to provide services to the user. A typical IoT system can be consisted of three parts: devices, network and processing.

IoT devices are not just sensors but are growing into any objects in our daily life embedded with computing unit and connected to the Internet, covering from phones, wearables, office equipments to industrial machines. In 2017, Ericsson predicted that globally, around 29 billion of these devices will be connected to the Internet by 2022 [Collela, 2017]. The data from these massive number of devices is need to be transferred, stored, processed, and analysed efficiently.

To transfer data from these IoT devices to the place where the data can be collected and processed, different IoT devices are needed to be connected to the Internet. For enabling this connectivity, a network communication protocol, either wired or wireless, is used. It is obvious that wireless-based protocols are more suitable for IoT as they give mobility and flexibility to the IoT devices than wired connection. In general, for a communication protocol to operate in IoT environments, it is usually required to be lightweight despite to the device's performance, to use low-energy as IoT devices may have energy limitation, and to be robust for lossy and noisy network environment. Different types of IoT system have some specific requirements that the protocols need to be followed, or some requirements can be ignored. Common communication protocols used in IoT are such as Wi-Fi, Bluetooth, IEEE 802.15.4, Z-Wave, LTE-Advanced, LoRa and cellular networks.

On the higher level, the application protocol creates an abstraction layer to allow IoT components connected through different communication networks to communicate to each other. While the popular internet application protocol such as HTTP can be used for IoT, but it is not lightweight, and so not suitable for the IoT devices. There are many protocols that have been proposed for IoT, those include CoAP, MQTT, AMQP and DDS. Different protocols can provide different features and have requirements. However, they are all based on the conventional internet protocol.

In emergency warning or environmental monitoring IoT systems, such as [Arbib et al., 2019] and [Poslad et al., 2015], IoT sensors are deployed in an area to collect data and detect a hazardous situation. As the on-site users should be promptly notified with the important information in an emergency, the ability to provide real-time information is an essential requirement for this type of system. In forest fire monitoring scenario [Dubey et al., 2019], IoT sensors can be deployed in the forest to collect data to the cloud server. Then the data can be analysed using a complex computation model to detect forest fire, and notification can be sent to people in the area. However, in such remote area, the internet connectivity can be frequently disrupted or unavailable. The sensors or user's devices in the area may not be able to communicate with the internet server. Thus, making the system unable to timely inform the users when the incident is detected.

In cloud-based IoT system, collected information from sensor nodes is sent to store and process in the central cloud servers. This allows a number of IoT nodes and its data to be managed from a single location with a well-established security parameters. The system can provide powerful computation service as all the sensor data is available at

one place. However, it may not be able to handle the massive growth in the number of IoT devices, and more importantly, the processing service will be unavailable when the connection to Internet is disrupted.

Opposed to the centralized cloud-based, the computation service can be pushed closer to end-devices location to reduce the network traffic and latency. *Edge Computing* [Satyanarayanan, 2017] is a broad term to describe a distributed computing architecture in which computation is served closer to the network's edge, i.e. closer to the user's or IoT device's location. The computation task is performed at edge infrastructures, which are typically implemented by deploying *Edge servers* on the network edge, resulting in lower latency and network traffic where the server is deployed. However, using Edge computing does not mean that it is required to work without the core cloud. In the most cases, the edge server is used for supporting the cloud infrastructure by leveraging the locality of computation to reduce latency and increase overall system capacity. *Fog computing* [Bonomi et al., 2012] [Luan et al., 2016] is a type of computing in which the edge nodes become an extension of the cloud server to support the scalability of IoT devices. Instead of only performing computation in the cloud, the smaller computation nodes placed at the network edge, such as an IoT gateway, can pre-process the data before sending the result to the cloud. For example, ThingNet [Qiao et al., 2018] which is a demonstration platform that uses application-level function chaining to distribute IoT data processing to the Edge nodes.

Mist computing [Preden et al., 2015] further pushes the computation to the extreme edge of the network, i.e. on to the actual sensor devices. The computation can be performed locally by utilizing the built-in lightweight processing unit or microcontroller on the device itself. But since the device's processing power is limited, implementing the system for it can be challenging.

As the sensor nodes are computation capable, it can pre-process its own sensor's measurement before deliver the result to a central server or a consumer. Furthermore, the devices can communicate and perform simple aggregations locally in the device's local network without relying on to the external server. This technique can be referring to *In-network computation*, in which the computation is distributed among devices in the network.

2.2 In-network Computation

The in-network computation is a technique to perform distributed computing in a network, aiming at reducing the cost and latency of network communication. At the beginning, the ideas of enabling the network routers or network switches to be programmable to perform customized computation on the packet was firstly defined as *Active network* [Tennenhouse et al., 1997]. Later, a similar paradigm was widely applied in *Wireless Sensor Networks* (WSN), the networks of resource-constrained sensor devices. The attempts to retrieve

information from WSN intelligently was earlier appeared in [Bonnet et al., 2000] and [Govindan et al., 2002]. In these works, sensor network are treated as a distributed database of sensor data, where a user can send through the gateway an SQL-like command to filter and aggregate the data collected from the sensors. The processes are performed in the network while the result data is forwarding back to the user. A study from [Heidemann et al., 2001] described in-network processing for WSN, that allows data aggregation to be performed at intermediate nodes of the network and forwarding the result data back to the gateway. TinyDB [Madden et al., 2005] was one of the very first platform which demonstrated the in-network processing to query sensor data from a WSN as a distributed database. Later, StonesDB [Diao et al., 2007] utilized the storage in WSN devices with data management to provide historical data querying in the WSN distributed database.

The generic computation model was then added and referred to as in-network computation in [Giridhar & Kumar, 2006] where the network not only forwards data but also able to perform generic computations. The network becomes a network of computing nodes where data can be processed nearer to the source and real-time responses or actions can be taken timely and appropriately within the network itself. Moreover, the data traffic is reduced as raw data is processed, and only meaningful results are transmitted when needed.

Recently, in-network computation technique is also applied to the next-generation application where localized information processing is time-critical. For example, in Extended Reality(XR) which is a combination of virtual reality and augmented reality, the environment information and image processing can be performed in-network which significantly reduces the latency [Montpetit, 2019]. Or, in industrial applications such as [Kunze et al., 2021] where the computation of coordinate transformations for assembly robots can be offloaded to the network devices.

Because the communication inside a WSN is usually self-contained with a fairly low-level communication protocol, the in-network computation can be implemented in a particular network. However, being self-contained, an IoT gateway is required to allow the sensors to send data to the Internet. Though, it is possible to embed the distributed computation on IoT/WSN devices while using well-known IoT application protocol, it is not efficient. This is due to the complexity in the mapping process, where a user's request needs to be dynamically resolved to an address of a host in which the related service is available. While the existing DNS can resolve the domain-name to host address, it can neither handle network disruptions nor address changes due to mobility.

2.3 Information-Centric Networking

During the past decade, the majority of internet usage has been shifting from end-to-end communication into accessing information and service. As the conventional IP protocol was designed for the end-to-end communication, the Information-Centric Networking

(ICN) is introduced as a future network paradigm to cope with new emerging usage requirements. Opposing to traditional host-centric networking where the endpoint address is used in communication such as Internet Protocol, ICN moves the current networking perspective from determining the location where a content is kept, to describing what is the content itself. The network is designed around the name of the content to allow the consumer, e.g. user or application, to retrieve the information efficiently from the network. During the past decades, there are several research projects and practical implementations on ICN which are reviewed as following:

2.3.1 DONA

DONA (Data-Oriented Network Architecture) [Koponen et al., 2007] was the first completed ICN, designed as a clean-slate naming and name resolution architecture to adapt to the new requirements. The design uses flat naming, which is consisted of a hash of the content producer's public key and a label to identify a content. A hierarchical network of name resolution nodes called Resolution Handler (RH) is introduced. The RH allows a producer to register a name and allows a consumer to resolve the name into the producer's address.

2.3.2 PURSUIT

PURSUIT (Publish-Subscribe Internet Technologies) [Fotiou et al., 2012] was a project attempted to research and develop an ICN architecture based on a publish-subscribe protocol called Publish-Subscribe Internet Routing Paradigm (PSIRP). PSIRP uses a rendezvous network to handle the content publishing, name resolution, and provide a forwarding identifier for content transfer in a path-label forwarding network.

2.3.3 Networking of Information

Networking of Information (NetInf) [Dannewitz et al., 2013] uses a flat naming scheme and a Name Resolution Service (NRS) for name registration and resolution. A hash of the data or a public key of the content producer is used as a name of the content, allowing the integrity or the ownership of the content to be verified using the name. The producer can register the content names to the NRS. Then, a consumer can send a query to the NRS to get a routing hint for the producer. The routing hint allows the consumer to directly communicate with the producer on the lower-layer network for requesting and transferring the content.

DONA, PURSUIT and NetInf proposed the name resolution layer where a named request is resolved into traditional forwarding information such as a path or an endpoint address, while the content transfer between producer to consumer is separately left to the lower-layer protocol's forwarding mechanism. This separation can increase system

overhead to communicate between layers, so the designs are not suitable for the networks of limited performance IoT devices. On the other hand, another clean-slate ICN approach, called CCN, has merged the name resolution and packet forwarding into one layer.

2.3.4 Content-Centric Networking

Content-Centric Networking (CCN) [Jacobson et al., 2009] is an ICN protocol stack with name-based forwarding and human-friendly hierarchical naming. Two types of packets were introduced: *Interest* and *Data* packets, of which the Interest is a request packet and the Data is a response packet containing the content that can satisfy a matching Interest. Both packet types contain a hierarchical name, which allow them to be forwarded and matched using the longest prefix matching. The Interest forwarding is decided by information in the *Forwarding Information Base(FIB)* and the last-hop direction is recorded in the *Pending Interest Table(PIT)* in each forwarding node. When the Interest meets a matching Data, the Data will be forwarded back in the reverse path of the Interest using the information recorded in the PIT and also being cached in the *Content Store(CS)* in each node in the path to satisfy the future Interests. The CCN architecture has been derived into many implementations using the similar mechanism. The major CCN implementations are such as CCNx, CCN-lite and NDN (Named Data Networking).

CCNx [Mosko et al., 2019] is a reference implementation from the original CCN specification.

CCN-lite [CCN-lite,] is a lightweight implementation of CCN which can be run on a microcontroller based on RIOT operating system.

NDN [Zhang et al., 2014] extends the CCN specification with extensible packets format and has modular design implementation, allowing functionality to be added without breaking the architecture.

These three projects share the same core mechanism which is based on CCN; however, they are architecturally designed with different goals. The CCNx is implemented as protocol reference, but the development has not been continued for many years at the time of writing of this thesis. The CCN-lite is specifically designed to be lightweight to fit into a microcontroller. While it also supports NDN packet format, only the core CCN/NDN functionalities are implemented. On the other hand, NDN's platform is still evolving and has customizable design. The platform also provides programming libraries and tools for development and testing, such as PyNDN [NDN,] for development in Python language and ndnSIM [Mastorakis et al., 2017] simulator, which is based on the NS3 [NS3, a] simulator. With these advantages, the extensible NDN is chosen as the base platform

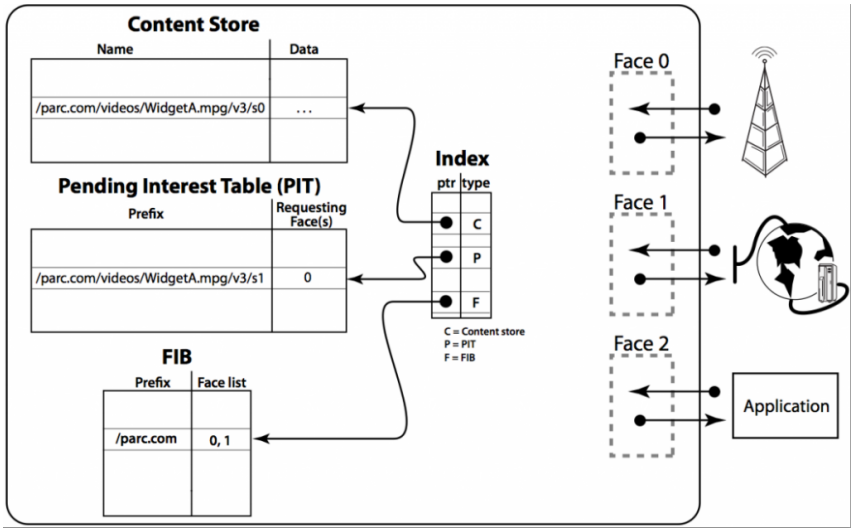


Figure 2.1 – NDN node architecture [NDN,]

for developing a new in-network computation architecture for IoT and WSN. The full detail of NDN is explained as follows:

2.3.5 Named Data Networking

Named Data Networking (NDN) [Zhang et al., 2014] is one of the ICN implementations which uses hierarchical naming and named-based packet forwarding, extended from the CCN protocol. A typical NDN node is shown in the Figure. 2.1 [NDN,]. It contains internal data tables for packets forwarding decision and an abstract communication layer consisted of multiple *Face*s (i.e. interface), each for communicating with another component such as a wireless link, an internet connection or an application.

A consumer can make a request to the network by sending an *Interest* packet containing the name of the content through a forwarding *Face* of an NDN node. The *Face*s are abstractions of communication channels that NDN forwarder exchanges NDN packets with other components, nodes, or networks. NDN routes the Interest through nodes in the network until a name-matched data object is found either in a cache or directly from the producer. The matched data object will be forwarded back to the consumer as a *Data* packet, while on the return path, NDN caches the Data in the intermediate nodes for possible Interest resolutions in the future.

NDN Packet Forwarding

The Figure.2.2 [Zhang et al., 2014] shows the forwarding decision of NDN packets. When an NDN node receives an Interest packet, the node's *NDN forwarder* will search in the *Content Store (CS)* for a cached Data packet. If the Interest name matches a cached

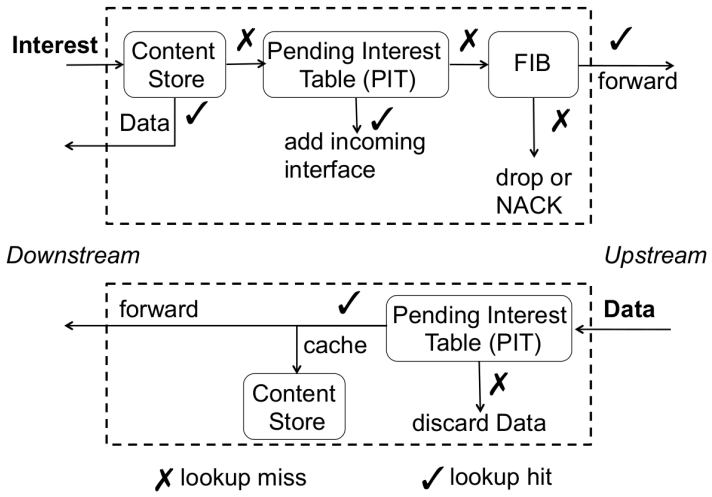


Figure 2.2 – NDN packets forwarding [Zhang et al., 2014]

Data packet name, the forwarder can immediately reply with the cached Data by sending the Data back to the same Face through which the Interest came in, without further processing. In case there is no matching object in the CS, the NDN forwarder will continue to forward the Interest packet by, first, it checks in the *Pending Interest Table (PIT)* for an identical Interest packet; if found, it only records the Face which the packet came from and then the redundant Interest can be discarded from further forwarding. If there is no matching, then the NDN forwarder will record the Interest in its PIT together with its incoming Face and, then, it will choose the nexthop Faces from the *Forwarding Information Base (FIB)*. The FIB is a table of records containing a hierarchical prefix together with its corresponding nexthop Face, where the FIB entry with the longest prefix match to the Interest will be chosen by the NDN forwarder. Each entry may have multiple Faces for the NDN forwarder to select from; whether a Face is selected or not is depending on the *Forwarding Strategy*. The Interest will be forwarded to the selected nexthop Face. Such forwarding continues through each NDN node, until the Interest could locate the requested object.

After locating the requested object in an NDN node, the node will send the Data back to the consumer via the incoming Face of the particular Interest. Each node who received the Data will look up for a PIT entry belonging to the matching Interest and forward the Data via the corresponding Face, and thus the Data is forwarded in a reversed path of the Interest back to the consumer. Additionally, on the return path, the forwarder in each NDN node also caches the Data in its CS. This process continues until the consumer is reached.

Apart from the name prefix, the NDN's Interest packet can contain optional fields, allowing different forwarding configuration per packet. Some important fields are the

following: The *Lifetime* field defines a time duration of how long a PIT record for the Interest will be valid. The *Exclude* field contains a list of name suffix of the Data which is excluded from matching to the Interest. The *CanBePrefix* field allows choosing which Data matching algorithm to be performed. The default value is true in which *prefix matching* is selected, or if it is set to false, the *exact matching* is selected. The *MustBeFresh* field holds a boolean value which used when searching for Data in CS, If true, the Interest will only match the Data that is in the *fresh* state.

The Data packet also have its optional fields, such as: *FreshnessPeriod* which specifies how long the Data will be in the *fresh* state when it is cached in the CS.

The Interest forwarding uses the longest prefix matching in the FIB table to decide the nexthop Face, the matching can results in multiple Faces. The *Forwarding Strategy* takes a decision on how an Interest will be forwarded among the Faces. NDN provides built-in forwarding strategies such as Best-route, Multicast, Access Router, ASF (Adaptive Smoothed RTT-based Forwarding), NCC (Content-Centric Networking, CCN in reverse) and Client Control. The *Best-route strategy* forwards an Interest to only one Face that has the lowest cost from the longest prefix match, except the incoming Face which the Interest is received. Particularly, this strategy is the default configuration of NDN stack. The *Multicast strategy* forwards the Interest to all Faces given from the longest prefix matching, except the incoming Face. The *Access Router strategy* improves Multicast strategies to forward more effectively. The first Interest is forwarded to every Face given from the longest prefix matching like in Multicast strategy. Then, it remembers the Face that the first Data came in and will forward the subsequent Interest only to that Face. *Adaptive Smoothed RTT-based Forwarding (ASF) strategy* selects a nexthop Face that has the smallest SRTT (Smoothed Round-Trip Time), i.e. the round-trip time in which the measurement's anomaly is removed. *NCC strategy* is an implementation of the default strategy of CCNx. This strategy is only provided as a backward compatibility to the original CCN. The *Client Control strategy* is a strategy that allows the consumer application to choose the forwarding nexthop of its Interest packet.

The NDN forwarding mechanism will work in the assumption that the Faces and the prefix in FIB are configured in each node. The configuration can be manually for static network. For dynamic network, NDN has routing protocol designed specially for it.

Named Data Link State Routing Protocol (NLSR) [Hoque et al., 2013] is a proactive link-state routing protocol specially designed for NDN. It uses synchronization protocol to propagate Link State Advertisement (LSA), calculates the shortest forwarding path of each prefix and maintain FIB table. However, the NLSR is designed for an infrastructure or a wired network where each Face is known to be connected to a specific neighbour node. In ad-hoc wireless networks, the neighbour information is unavailable, thus NLSR cannot be used in wireless network.

NDN Forwarding in Wireless Networks

Unlike wired networks where the direction of forwarding can be determined because each NDN Face is typically bounded to a network interface which dedicated to a point-to-point connection to a neighbour node, the wireless networks have different properties: an interface is multi-access, the communication medium is shared with multiple nodes, and the network has dynamic topology. Firstly, a typical wireless node setup has one wireless interface which is multi-access, providing broadcast transmission to multiple neighbours. Without a discovery protocol, the information of the nearby neighbour nodes are usually unknown, so the next hop neighbours could not be determined by a wireless NDN Face. The second concern is that the wireless networks use shared-medium communication, where there is only one node can take control over the medium channel to transmit data. Having multiple nodes transmitting simultaneously is subjected to cause significant interference, and collisions lead to packet loss. Some wireless protocols, such as Wi-Fi, provide a collision avoidance mechanism, but they are not efficient for broadcast traffic [Torrent-Moreno et al., 2006]. According to the native NDN design, reliable data transfer is not provided in the forwarding layer, but it is forced to be handled in application layer. Furthermore, with the mobility, the wireless network topology can be dynamically changing. A connectivity between nodes can be created or loss all the time. From these challenges, the primitive NDN forwarding will not be suitable to be used in wireless network.

To handle NDN packet forwarding in wireless network, many works have been proposed different forwarding techniques. According to a survey paper [Amadeo et al., 2015], the techniques can be categorized into three types: Blind forwarding, Next-hop aware forwarding and Provider aware forwarding. The review of NDN forwarding techniques is summarized as follows:

Blind forwarding is a strategy, that simply floods the Interest to all nodes in the network to find the desired Data. Every node relays the Interest until the matching Data is found, then the Data will be forwarded back on the reverse path. This type of forwarding does not need additional control traffic to maintain the FIB table. The loop prevention is already provided by the NDN forwarding using PIT table, and thus a node will drop the packet that is already seen. However, the flooding generates a lot of traffic and packet collisions, which reduces the successful packet delivery rate. A number of techniques are invented to avoid the packet collisions such as transmission deferring which randomly defer a transmission, packet suppression which drop the packet if the same packet is overheard by a number of time and packet retransmissions [Wang et al., 2012]. These techniques are also practised in the other type of forwarding.

Next-hop aware forwarding is a type of broadcast forwarding scheme in which each forwarder have to decide whether an Interest packet will be forwarded or not, based on

some supplementary information, such as location, distance, previous forwarders, etc., to reduce the flooding traffic. The forwarding continues until the matching Data is found, then the Data will be forwarded back on the reverse path.

In BlooGO [Angius et al., 2012], the sender attaches the Interest packet with a bloom filter containing the IDs of its neighbours. Any receiver will relay the packet only when some of its neighbour are not included in the bloom filter. However, to maintain the list of neighbour, each node is required to broadcast the beacon periodically to advertise its node identifier to its neighbours.

Direction-Selective Dissemination [Lu et al., 2013] uses GPS location to divide the neighbours into geographical quadrants, and the farthest neighbour from each quadrant is selected as a next forwarder. In this scheme, a sender attaches the GPS location in the Interest packet which the node will broadcast to the neighbours, then each receiver reply with an acknowledgment packet with its GPS location back to the sender. The sender selects the farthest neighbour from each quadrant as relay node and broadcast the selection result to the neighbours, then the selected relay can broadcast the Interest attached with its GPS location, and the process is repeated.

In Neighbourhood-Aware Interest Forwarding(NAIF) [Yu et al., 2013], an Interest receiver node decides to forward or drop the packet based on probability calculated from the number of overhearing the same Data packets which are broadcasted by its neighbours. The more packet the node overhears, the lower the rate of forwarding, thus suppressing unnecessary Interest flooding traffic.

V-NDN [Grassi et al., 2014] uses a deferring timer to delay the forwarding based on the distance between the previous sender and the node. The closer to the sender, the longer the delay, thus the farthest node will relay the packet first and so suppressing the other waiting forwarder that overhears the packet. Each forwarder puts its GPS location in the Interest packet to allow the other node to calculate distance.

Provider-aware forwarding is a type of forwarding scheme that incorporates self-learning or routing protocol to maintain nexthop information, so the path to a data provider is determined for each prefix.

Listen-First, Broadcast Later (LFBL) [Meisel et al., 2010] uses the distance-based self-learning with next-hop aware forwarding technique to decide how to forward or drop the Interest, allowing the Interest to forward to only in the direction to provider. Instead of FIB, each node maintains an additional distance table using the hop counts information piggybacked in the Interest and Data packets to track to the shortest path to consumer and producer. Then, the distance table can be used in forwarding decision whenever the information is available.

E-CHANET [Amadeo et al., 2013b] takes the distance-based learning approach, like in LFBL, and extends it to handle dynamic mobility and to control transmission rate. By adding provider node ID, hop count and node's potential transmission rate to the Interest

and Data packets before broadcasting, the receiver can maintain its own forwarding table and decides the whether to forward the packet or not and defer each transmission based on the collected information.

dd-NDN [Amadeo et al., 2013a] uses directed-diffusion technique to establish a path from the consumer to the data source. Staring with a consumer node floods Interest packet to the network until a matching Data is found. Then, while the Data is being forwarded back to the consumer, the intermediate node maintains a next hop table which collecting the node where the Data came from. From information in the table, a path from a consumer to a data source is established for the future Interest.

Reactive Optimistic Name-based Routing(RONR) [Baccelli et al., 2014] simply floods the Interest when there is no matching FIB entry and when the first Data is received. Then, a FIB entry will be created to the previous Data sender, so the subsequent Interest can match the FIB entry and will be sent to the nexthop by unicast.

Navigo [Grassi et al., 2015] further extends V-NDN with geographical forwarding on vehicular network, using the Geographic Face (GeoFace), i.e. Face that represents a geographical area, and road mapping information. Starting with no matching record in FIB, a consumer's Interest is broadcasted and relayed to all direction by the farthest neighbour, as in V-NDN. When an Interest reaches the matching provider, the provider attaches its geographical area information to the Data packet, which will be forwarded back to the consumer. Along the way back, each intermediate vehicle node learns the provider's geographical area by mapping it with a GeoFace, and then the Data's prefix with the GeoFace is recorded to the FIB table. After the Data is delivered to the consumer, a subsequent Interest is issued. The prefix of the Interest will match a FIB record, pointed to a GeoFace which is mapped to the provider's geographical area. The geographical shortest path to the provider's area is calculated using road map information. Then, the Interest can be forwarded along the vehicles in the path, until the provider is reached.

In Dual-mode Interest forwarding (DMIF) [Gao et al., 2016], two forwarding modes: flooding mode and directive mode, are chosen by the forwarding node whether the FIB lookup result is miss or hit respectively. The FIB is also updated with information from the returning Data, the nexthop is chosen by the lowest cost calculated from hop count and available energy of the next hop node. A deferring timer and overhearing is also used for suppressing broadcast packet transmission, resulting in lower network traffic.

HoPP(HoP-and-Pull) [Gündoğan et al., 2018] is a publish-subscribe forwarding scheme for IoT network. HoPP uses Content Proxy nodes as rendezvous point to handle producer's prefix registration and packets relaying between consumer and producer. However, by relying on the proxy nodes, the traffic will not go through the shortest path between consumer and producer, but the path will be depended on the placement location of the proxy nodes.

[Liang et al., 2020] proposes self-learning forwarding for wireless networks by adding the announcing prefix in Data packet to populate FIB entry, and adding the unicast Face

creation for better collision handling in MAC layer in wireless network. When an Interest could not match any FIB record, a consumer will send the Interest with a discovery tag to the network which the Interest will be broadcast and flooded to all nodes. A producer who has the matching Data will return the Data attached with a prefix announcement field. The intermediate nodes that forwards the Data back to the consumer will create an unicast Face which is mapped to the address of the previous hop and add the announced prefix in its FIB with a record points to the new Face, building a path to the producer for the next Interest with the same prefix.

Though unicast may have better collision handling, however on sensor data gathering use case, the sensor data is usually small and does not need very high throughput. To collect data from multiple nodes, a broadcast Interest forwarding will work more efficient as a node needs to be transmitted only once, but using unicast, the network traffic will be significantly increased since one unicast Interest is sent and forwarded to each individual node.

Learning-based Adaptive Forwarding Strategy(LAFS) [Djama et al., 2021] uses the self-learning and adaptive forwarding mechanism for constrained wireless network. LAFS has two mode Interest forwarding: deferred forwarding when FIB lookup miss and learned-based forwarding when FIB lookup hit. A next-hop field is added to the Interest packet to contain the node ID. By checking the next-hop with its ID, the receiver can process the Interest if the ID is matched. When forwarding an Interest, if a FIB lookup miss, the same deferring timer method is used in which the transmission of Interest is deferred before transmission to all members or will be suppressed if the node receive the same Interest during the deferring time. If the next-hop is found in the FIB, the ID of the next-hop node is put in the Interest and transmit immediately. The Data packet is also attached with the hop count and the sender fields. Each node in the Data forwarding path update the matching FIB record's nexthop if the new hop count ID lower and then increase the hop count and update the sender to itself before forward it.

NOLSR [Guo et al., 2021] is a proactive link-state routing protocol for NDN mobile ad-hoc network which maintains the forwarding prefixes in FIB table, so the normal NDN Interest packet can be successfully forwarded to the producer. The NOLSR utilizes MPR technique from OLSR [Jacquet et al., 2001], which limit the flooding relay nodes when disseminating routing control information to avoid broadcast storm. By periodically broadcast HELLO and LSA packets, the NOLSR nodes can discover their neighbours and exchange prefix information to maintain the FIB records.

All the existing NDN forwarding design for the wireless network was designed to forward an Interest to retrieve Data from one provider node. However, for the data harvesting application, using these approaches, one Interest must be created for each provider node. And to defer the forwarding, the NDN forwarder must be modified to a great extent. In this work, we aim for the lightweight approach, so the simple broadcast forwarding is used with the collision avoidance mechanism implemented in the application

level.

NDN in IoT

NDN has been demonstrated to provide decentralized IoT infrastructure without relying on cloud [Shang et al., 2016] to provide common IoT features, such as name-based communication, device provision using NDN schematized trust and name-based rendezvous point, allowing cooperative computation between the applications and devices [Shang et al., 2017].

Recently, NDN has been studied and integrated into the existing IoT systems such as, vehicular networks [Grassi et al., 2015], smart homes [De Silva et al., 2016], robot networks [Dauphin et al., 2018], and healthcare monitoring [Saxena & Raychoudhury, 2019].

NDN has been compared to traditional host-centric IoT protocols. The work of [Baccelli et al., 2014] shows that CCN based protocol can have lower energy consumption footprint than IPv6, RPL and 6LoWPAN protocol. In [Gündoğran et al., 2018], the NDN's in-network caching allows NDN to outperform CoAP and MQTT in successful delivery rate on disruptive wireless multi-hop environment.

However, its design has difficulties on some IoT deployments, such as at the network edge [Liang et al., 2020], where there are co-existing multiple types of communication, cost of running traditional routing protocol, and dynamic network environment issues are needed to be concerned.

One important process of IoT for environment monitoring system is data harvesting, in which the data from many sensors are being gathered to be processed.

Data Harvesting with NDN

Wide-area environment monitoring system takes data from sensors deployed in the wide-area to process and provide insights on the monitoring environment. To collect the data from numerous sensors, data harvesting techniques is used. Typically, data can be collected from the sensors using either push or pull method. NDN is natively support pull method by sending an Interest to retrieve the Data, while the push method is also supported by additional synchronization protocol such as Psync [Zhang et al., 2017].

However, one downside of using NDN on data harvesting is: it was designed with a 'one Interest per one Data object' principle to control the amount of data transfer. An Interest name can match and retrieve exactly one Data, as the name of a Data is uniquely defined. This leaves the responsibility to the application to send multiple Interest packets to retrieve multiple Data. It makes data harvesting in traditional NDN inefficient regarding scalability, concerning that the network and device resources requirement will be increased with the network size.

In wireless network, multiple Data with the same prefix can be retrieved by a single Interest using a mechanism proposed by [Amadeo et al., 2014b] which employing the broadcast nature of radio transmission. By modifying NDN forwarding to not deleting the PIT record after the satisfying Data is received, multiple Data can be forwarded to the consumer until the PIT record expires. The mechanism was also designed to handle communication problems in wireless networks by deferring Data transmission with a random duration to avoid packet collisions, including packet re-transmissions to recover from a packet loss, and utilizing the Exclude selector feature of NDN protocol to reduce the redundant Data in the retransmission of Interest. However, the work only concerned the data collection in one-hop communication.

The efficiency of data harvesting can be further improved by using the in-network computation techniques, instead of just transferring all the raw data from the sensors to the collector node, the data can be pre-processed or aggregated during the forwarding by the relay nodes, so the size of data to be transferred is reduced. The idea of applying in-network data aggregation in NDN was demonstrated in [Amadeo et al., 2018] and [Liao et al., 2019]. In these work, a node in the network is allowed to send a request Interest packet to all neighbours, similar to broadcast transmission. Each neighbour will continue to forward it, until the Interest is flooded throughout the network and nodes at the end return data back to the previous Interest sender. Each node will collect the data returned from their neighbours and aggregate them before return it to their previous Interest sender or parent node. This process repeats until the final result are back to the first node. However, both of the proposed systems assume that the neighbour information is available, allowing them to detect if a node has received data from all neighbours before processing and returning the result. This may not be applicable for the wireless network without discovery or routing protocol.

2.4 Computations in ICN

Performing additional processing of packets of data flow in the network before it is delivered to the destination has emerged as a new trend for developing future internet architecture. The term of in-network computation has widely studied in the literature with several domains such as network management in SDN [Arumaithurai et al., 2014], data processing in IoT networks [Tschudin & Sifalakis, 2014] or service delivery in edge computing [Braun et al., 2011]. Even though ICN was originally designed for content retrieval, there are many attempts to expand the ICN functionalities to support in-network computation. The concept of ICN is applicable for decoupling the location of a particular network function instance from the identity of the function it provides.

In [Arumaithurai et al., 2014], the Function-Centric Service Chaining (FCSC) has been proposed by taking function identifiers as route-able prefixes. A named-based network management protocol was introduced by using NDN-like hierarchical name forwarding

to forward traffic packets through a chain of functions in the network. Traffic packets that come into the network can be tagged by the border nodes by adding a header of a hierarchical name which is a series of a chain of functions that the packet has to go through, defined by network admin. For example: a packet tagged with a name "/Firewall/funcA/funcB" means this packet is designated to a "Firewall" function then function "funcA" and then function "funcB".

Named Function Networking(NFN) [Sifalakis et al., 2014] extends ICN/NDN architecture by adding Lambda expression to the naming to orchestrate distributed data computations in the network with call-by-name; if fails, the positions in the Interest names can be switched and call-by-value can be applied. With Lambda expressions, NFN can manipulate the forwarding by changing Interest prefix, allowing the Interest to be forwarded to a different direction to search for function codes or data objects separately and whenever both the required code and object are found, an execution can take place according to the *resolution strategy*; an application layer with a lambda expression resolver, pusher and evaluator is added. The NFN's basic resolution strategy is called *Find-or-Execute (FoX)* in which it first tries to find the cached result or later, if not found, execute the computation. However, this strategy is not suitable for the IoT network as there is no mechanism to prevent the computation to be performed in less powerful nodes, which is not efficient.

For IoT edge network, PIOT [Ye et al., 2016] uses a specific name prefix to enable any performance network node to publish its computation service and capture appropriate Interests, execute and return the result, simplifying the general NFN computation model to satisfy the computation requirements in IoT scenarios.

The functional chaining has been demonstrated to be working in NDN [Liu et al., 2016]. They apply on-the-fly processing, allowing an ordered set of functions to be executed seamlessly. However, the ordered set of function chain is prespecified and static once they are called, and so it can be disrupted in dynamic network environments.

Later, NFN also introduced three additional strategies for IoT [Scherb et al., 2018]. Firstly, *EdgeFoX* is a strategy for stationary network, like PIOT, prioritizing on pushing computations to the edge nodes that have higher computing resource by utilizing a dedicated name prefix. The other two additional resolution strategies: *Find-and-Execute (FaX)* and *Find-or-Pull-and-Execute ((FoP)aX)* are introduced for dynamic mobile IoT networks. *FaX* is designed for short running computation in mobile IoT networks. Instead of only searching for the data stored in network cache, it also starts a computation simultaneously and then take the result from whichever returns first. In this essence, the node can still complete the task, even it is disconnected from the network. *(FoP)aX* is extended from FaX with an ability to fetch the intermediate result from another running computation using their proposed R2C(Request-to-Computation) protocol. The R2C protocol is useful in a long computational task, as it allows transferring the intermediate result of a running computation from another node without having to wait for the whole computa-

tion to finish. In this way, the function can start the processing from where the existing intermediate result is found; thus, it does not have to redo the whole computation.

NDN uses reverse path forwarding to return the desired Data packet back to the consumer, this requires the information in PIT record to maintain the Interest inquiry until the Data is successfully delivered to the consumer. The Interest lifetime is used for determining if a PIT entry is still valid or expired. However, different functions take different amount of execution time for the computation and return the result Data. Instead of using the fixed Interest lifetime, different timing techniques were proposed to maintain the PIT record. The work of [Król et al., 2018] has categorized the dynamic timing techniques for computation in ICN into 3 groups.

1) Network timescale: a consumer sends Interest with a short lifetime which is long enough for network forwarding time or RTT and keep retransmit when timeout, until the data arrives. If the computation takes longer time, the retransmission can create unnecessary network traffic.

2) Application timescale: the Interest lifetime is set longer, as it is considered to cover the expected computation time. If computation takes longer than expected, the consumer needs to retransmit the Interest. However, if the requesting Interest is lost, it will cause unnecessary waiting before the retransmission is triggered by a timeout.

3) Using Acknowledgement: this refers to a process of adding an Acknowledgment message to the NDN protocol. A consumer can send Interest with specific lifetime and then the producer/compute node will immediately respond with the Acknowledge message. The Acknowledgment will extend the expiry time of Interest which is holding in the PIT of nodes along the forwarding path until the computation is done and the response Data is returned. However, with mobility scenario, the path between consumer and producer may abruptly change during computing time.

The approach applied in this thesis is close to the Application timescale, in which the Interest lifetime is set to be long enough to cover the computation. However, in this scheme, the lifetime can be used as a deadline for telling a computation to terminate earlier before the caller's lifetime gets expired. It is useful for data harvesting in a dynamic multi-hop wireless networks to specifying how far the computation request will be covered without additional traffic overhead.

RICE [Król et al., 2018], an ICN computation scheme that minimizes time overhead of retrieving the computed data, proposes by using the 4-ways handshake mechanism. This allows a consumer to pass large-size parameters using call-by-name and the thunk name, which is a routable prefix to allow retrieving a result from a specific computing node. The 4-way handshake works as following: First, a consumer sends Interest containing functions prefix and the name of parameters to the network. While an Interest is forwarded, a temporary entry containing a parameter's prefix pointed to the Interest's incoming Face is appended to the FIB table in the intermediate nodes, creating a path for the computing node to the consumer. After receiving the Interest, the computing node sends an Interest

to fetch the parameters published by the consumer. Consequently, the consumer returns the parameter as a Data packet back to the computing node. Finally, after receiving the parameters, the computing node can immediately respond to the request with a Data message containing the thunk name of the computation result and the estimated time to complete the computation. The consumer can then send an Interest with the thunk name to retrieve the result Data from the computing node.

Proposed by the same authors, Compute first Networking (CFN) [Król et al., 2019] has a distributed scheduling mechanism to load-balance the computing resource between nodes. Each computing node advertises free system resource information periodically through broadcast mechanism. The distributed scheduler maintains and synchronizes system resources information and the computation graph. Whenever a scheduler receives a request, it can dispatch the computation to available nodes by looking at the amount of free system resources.

RICE and CFN are suitable for in-network computation with support of both stateless and stateful computation by using call-by-name function. However, for computing in a sensor network, the data size is typically much smaller and the computation is lighter, so maintaining the thunk name and computation graph can be too costly for a small task.

IoT-NCN [Amadeo et al., 2018] extends NDN to support IoT data processing at the edge network. The computation requests can be resolved close to the IoT devices, to limit the raw data traffic and reduce the latency. By adding a Service Table to NDN architecture to store the name of available service in a node and using the priori known network topology information, the Interest can be forwarded and computation will be performed in the branching node in the network. This allows the data to be aggregated in the network between the IoT devices and the requester. [Amadeo et al., 2019a] then extends the IoT-NCN with a mechanism to allocate the computation to the optimal node using the Service Execution Cost (SEC) calculated from the network cost distance from the node to the data source and the availability of the processing capability of the node. The SEC is piggybacked in the Interest and using acknowledgement messages to signal the selection of the node with the lowest cost to preform execution. Further, NDN-Fog [Amadeo et al., 2019b] proposed the allocation of the data processing to be performed either at the edge network or remotely in the cloud to minimize the service latency. Using the similar mechanism as the SEC, the estimated Service Processing Time (SPT) is embedded in the Interest and propagated to find the optimal node to perform the execution.

ICedge [Mastorakis et al., 2020] proposed mechanisms for Edge computing on NDN where service discovery is provided. A dynamic self-learning forwarding is used to efficiently forward the Interest to a computing node. In this regard, a naming abstraction with Thunk allows a new Interest to be forwarded to the previous execution result.

IceFlow [Kutscher et al., 2021] is an ICN framework design of Dataflow-based computation, which coordinates the set of computing functions and data stream processing in a

distributed Dataflow pipeline. However, the processing graph for Dataflow is needed to be synchronized between nodes in the pipeline. The synchronization adds more overhead and complexity to the system; thus it may not be suitable for IoT application.

2.5 Synthesis

In this thesis, we propose a computational platform for the Internet of Things (IoT) that processes sensor data collected from IoT devices in real time to provide prompt responses to local users without the Internet connection. A common approach to achieve this is to deploy powerful nodes or edge servers in the local environment to centrally collect and compute results from IoT sensors. Our approach is to distribute the computations among the sensor nodes themselves by leveraging in-network computation. While the sensor data is forwarded to the user, the intervening sensor nodes can compute and aggregate part of the data and forward the partial result to the node near the user. In this way, the data generated by the sensors can be processed in the network, and the user can get the result in real time.

Computations in different nodes in the network can be invoked by function calls forwarded through name-based forwarding in the NDN. The requesting Interest is forwarded to the nearest computationally capable node, and the result data can be sent back in the other direction. Several methods have been proposed in the NDN for NDN forwarding in wireless sensor networks, such as neighbour-aware forwarding or provider-aware forwarding, to efficiently retrieve data from a single node. To retrieve data from multiple nodes, simple hop-by-hop broadcast forwarding is very effective and does not generate additional computational and traffic overhead compared to the other methods. Interests can be forwarded from one node to multiple neighbours, and all neighbour responses can be collected on return. However, the native NDN is designed to receive only one record per Interest. The Interest record is deleted from PIT as soon as the first record has satisfied it, so any data arriving later is discarded. To collect multiple data, the multi-source technique can be used by keeping the Interest record on PIT until it expires, even if a date is received. Another problem is the wireless communication environment with packet collisions, unreliability, and congestion issues. To solve these problems, techniques such as random transmission postponement, retransmission, and suppression of unnecessary transmissions can be used, which were considered in the development of our proposed architecture.

For orchestrating computations in the network, NFN has proposed the use of lambda expressions as part of the names with resolvers for lambda expressions. We intend to keep any constrained IoT device lightweight and low overhead while still being able to process IoT data in real time. Instead of adding lambda expressions as a shim layer to NDN, we use a simple function call-by-value in our proposed scheme, which can be implemented as name prefixes in NDN. Since the function pusher and evaluator is done in the function

program at the application level, the normal NDN naming scheme can be maintained, so no separate lambda expression resolver is required.

The works closed to our proposal are RICE and CFN, which apply the call-by-name and mechanism to provide both stateful and stateless computation. However, the idempotent functions are less of concern for our target applications than the ability to share computation among resource constrained nodes with a non-deterministic distributed computation model. In our proposed scheme, we keep our design as NDN backward-compatible, with minimal change to be deployed in resource constrained conditions targeting on processing basic sensor's data which are small and simple enough, e.g. a number or a string, on mobile nodes and unknown network topology.

3 ActiveNDN Architecture

IoT for Real-time incident warning system which requires immediate data and some computational responses to assess the situation, some may propose edge computing or centralized computing using cloud-based infrastructure. However, the area affected by natural disaster may not have internet infrastructure deployed or have poor internet connectivity. Many IoT for Real-time incident warning systems in those remote or rural areas are deployed in an ad-hoc manner, hence facing several challenges on intermittent connectivity and unstable electricity supply. These brought a huge roadblock to apply the cloud-based solution for such deployments. Not only those tiny IoT devices cannot send data out from the affected area, but the devices are also eventually offline due to the electricity shortage. Edge computing server cannot be deployed in the area either due to the same problem. The lack of connectivity, power and maintenance, so the only applicable approach is to distribute computational function into the device itself.

In traditional IP networks, to collect data or request computational service on a device, the device's address is required. However, managing the devices' addresses in an ad-hoc network of real-time IoT-based incident warning system is complicated. Researchers have worked on Information-Centric Networking (ICN) for content delivery problem. In particular, Named Data Networking (NDN), one of the ICN research projects, introduces the name-based routing that decouples the location of the content sources. Each NDN node can query the desired content or data directly by expressing the Interest message without knowing the IP address of destination node. This turns the network of IoT devices to be the distributed database. The integration of in-network computation and NDN allows the user's queries to be processed while giving the responses directly within the local network configured at the remote edge. An NDN agent is assigned to collect data from all relevant neighbour nodes while computing some useful functions. However, the existing approaches of in-network computation in NDN are either lack of backward compatibility with the native NDN implementation or requires extra modification for NDN integration which is not lightweight enough for tiny IoT devices.

In this regard, the lightweight ActiveNDN architecture is proposed to enable the embedded functions to be called and executed on IoT devices. This allows real-time in-network computations which can provide timely responses suitable for an incident warning system. The design principles of ActiveNDN architecture and the mechanism of the function call are explained in this chapter.

3.1 Design Principle

The goal of ActiveNDN is to build a programmable framework by adding the computation capability inside the IoT devices. The framework will allow the user to create their IoT application service that controls and orchestrates activities across the devices in the IoT network deployed in the challenged environment. ActiveNDN aims to provide a computation capability to process stream of sensor data inside IoT devices in real-time and allowing the in-network computation. The design principles of ActiveNDN are summarized as following.

Design principle I: chain of functions Each ActiveNDN node is capable of storing function programs and has the ability to locally execute or distribute the requested computation to other nodes. ActiveNDN provides two types of function call: single function and chain of functions.

In single function, the computation request (i.e. a function call) is executed locally inside the node while computation results are returned to the requester. In case that the request function call is not stored in the local repository, the node is responsible for forwarding the request to other nodes who have the requested function in local repository.

To have a chain of functions, during the execution phase, a function can make data requests and sub- computation requests (i.e. sub-function calls) to the internal function stored in local repository or calling the external functions stored in other nodes. The chain of calling creates a chain of function calls to build up the in-network computation capability across several ActiveNDN nodes in the network.

The function execution capability set ActiveNDN to the ultimate design goal, which is, allowing the user to program the network to orchestrate computation among the IoT devices.

Design principle II: lightweight and fully distributed ActiveNDN is designed to work in IoT sensors devices with limited computation resource, memory capacity and low bandwidth ad-hoc connection. Connection from these IoT sensors to the Internet can be intermittent or may not be available at all.

To allow computation in limited computation resources node, the ActiveNDN's chain of functions with in-network computation technique can distribute the computation tasks evenly among the nodes in the network. Each ActiveNDN node can perform the necessary computations, such as some basic statistical or mathematical functions. ActiveNDN works with wireless ad-hoc network by taking advantage of broadcast forwarding and overhearing in wireless network with the loop prevention in NDN forwarding. The function call can traverse thought the network without additional discovery and routing protocol. With NDN forwarding in wireless network and the in-network computation technique, each node can do data aggregation, so the amount of data is decreased, and thus saving the

storage space and network bandwidth.

Design principle III: NDN compatibility While there are existing works that has already put in-network computation in ICN, such as NFN and RICE, however, their designs modify the NDN forwarding mechanism in some greater extent, making them to be unable to work seamlessly with native NDN. The NFN requires an additional layer to process the lambda expression. In RICE, the mechanism to create the path from a producer back to consumer is not existed in the native NDN mechanism.

ActiveNDN is designed to be backward compatible with the native NDN while introducing the computational capabilities with minimal modification to existing NDN. The function identifiers for a function call are used as NDN prefix, making it routable by NDN. To add computational capability to NDN, a Function Library (FL) is attached as an external module to the native NDN. The FL module is a local repository for function codes implemented on top of native NDN forwarder. It is responsible for function storage and executing the function when it is requested. The detail of function library will be explained in section 3.4.3.

3.2 Proposed ActiveNDN Architecture

The architecture of ActiveNDN is illustrated in Figure 3.1 that consists of three main components: the *NDN forwarder*, the *Data Repository (Repo)*, and the *Function Library (FL)*. Each component communicates to each other through the abstraction interface, called *Face*s, inherited from native NDN architecture.

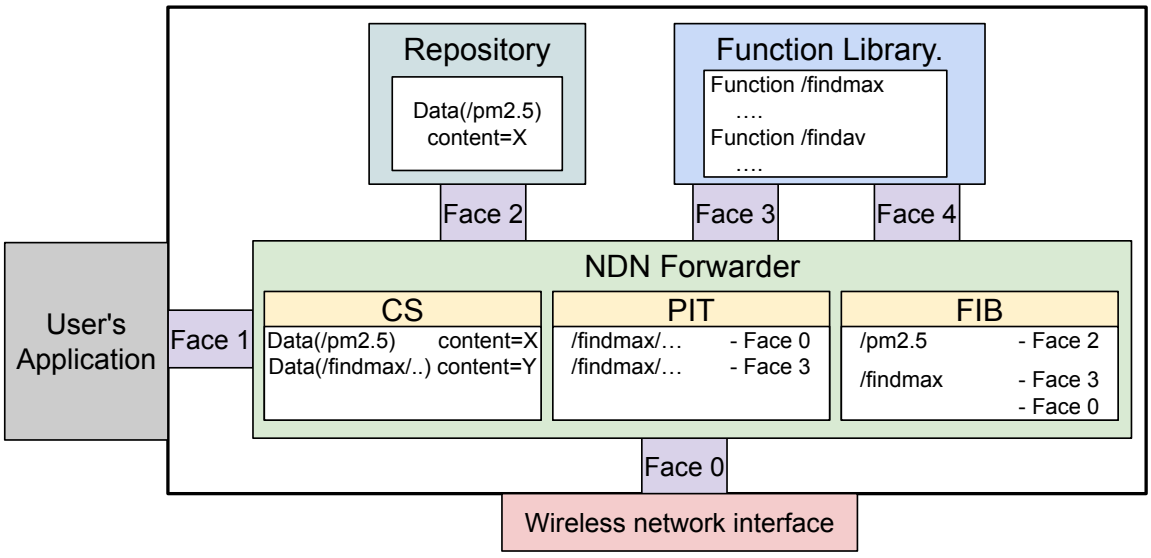


Figure 3.1 – ActiveNDN Node and components

To integrate the in-network computation capability over native NDN architecture, the *Function Library (FL)* is proposed to be responsible for storing and executing function codes. The FL is implemented on top of the native NDN forwarder where the interconnection is handled by *Face*, an abstraction of the communication channel managed by the native NDN Forwarder.

The operation of ActiveNDN is typically triggered from the users' application, where function requests are expressed following the NDN naming scheme (hierarchical naming) and passed to NDN forwarder through an internal Face (e.g., Face 1). Inside the NDN forwarder, it consists of three tables including Content Store (CS), Pending Interest Table (PIT) and Forwarding Interest Table which are used for forwarding mechanism as mentioned in Chapter 2.3.5. In addition, the ActiveNDN is also integrated with a *Repository (Repo)*, a persistence storage to store sensor-produced data and to provide the data for the functions in FL.

To describe ActiveNDN, a simplified scenario is chosen with one face per component category. As shown in Figure 3.1, *Face 2* is used for internal communication between Repo and NDN forwarder while *Faces 3,4* are configured for communication between FL and NDN forwarder. The *Face 1* is configured for communication with the user's applications, such as a web browser or scripting programming, and *Face 0* is connected with the physical network interface (e.g., Wi-Fi) to allow a node to transmit and receive packets from/to other nodes in the network. In general, there can be a different number of Faces in a node, such as multiple application Faces which each connects the NDN Forwarder to a different application, since multiple different applications for different tasks can be run simultaneously on a node, or multiple network Faces which each allows the NDN Forwarder to forward packets to a different physical network interface.

On the startup of a node, the names of the embedded functions in the FL are registered in FIB with associated interface numbers pointing to the Face of the FL or pointing to a network interface Face to enable calling the function across nodes or both. For example, in Figure 3.1), a function *FindMax* has configured a FIB entry: */findmax* prefix, associated with Face 3 and 0 which pointed to FL and to the network interface respectively, while a Data name */pm2.5* associated with Face 2 and pointing to the Repo is registered in FIB. In the current design, the decision of where functions or data reside is left to network administrators to configure the network.

3.3 Operation of ActiveNDN

To request a computation from the network, an Interest with an embedded function call is expressed by the consumer, e.g. the user, and sent to any ActiveNDN node in the network, but preferably the nearest node from the consumer. An example of expressing a computation request is illustrated in Figure 3.2 (1) where a user's application takes a request Interest from the consumer and forward the Interest to the NDN Forwarder via

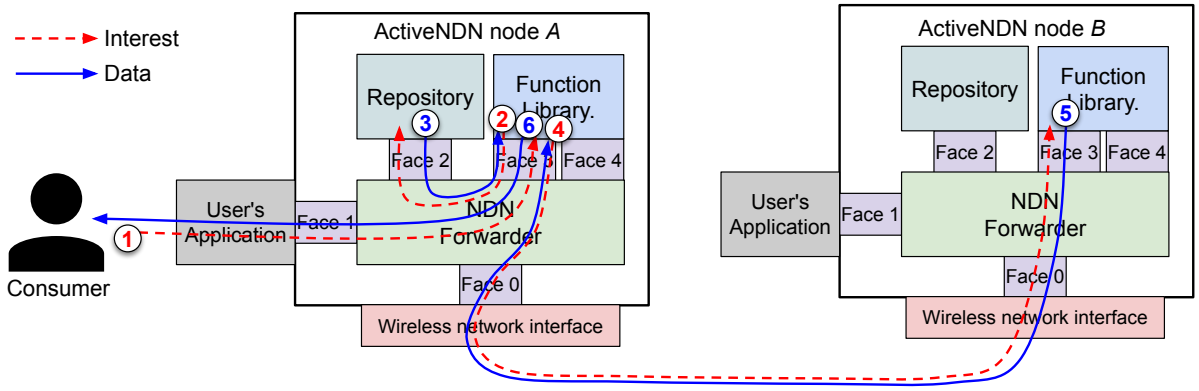


Figure 3.2 – ActiveNDN Operation

Face 1. This Interest is treated by the NDN forwarder as a normal NDN Interest packet. First, the Interest is checked against the records in node's CS and PIT, if it could not find the matching record, then it is checked against the FIB. If the Interest's name prefix matches a function prefix entry in the FIB table, indicating that the requested function is in the FL, the NDN forwarder will record the Interest in PIT, with Face 1 as the incoming Face and Face 3 as the outgoing Face, and then forwards the Interest via Face 3 to the FL to execute the function.

The FL extracts the function call parameters from the name in the Interest packet and performs respective function initiation and execution, where all parameters are treated as call-by-value to the called function. Each instance of a function execution is assigned a *nonce*, which is a random number generated by the FL as an execution process identifier. The nonce has a role for matching the *caller*s and *callee*s in a part of a function chain, which will be described in more details in Section 3.5.2.

During its execution, the called function may initiate a request for a data object that can be propagated over the network or from its Repo for the case of local sensor values. To request data from the Repo, the function sends a new Interest packet requesting data to the NDN forwarder then the Interest is forwarded to the Repo via Face 2, as shown in Figure 3.2 (2). Then (3), the Repo can send the requested sensor readings back to the NDN forwarder as Data packets, which are forwarded to the FL via Face 3. After receiving the data packet, the requesting function in FL can continue its execution.

A function in an ActiveNDN can call functions in other nodes, as well as be applied to sensor data in other nodes. The call can also be a call to the function itself, recursively within the same node or distributed throughout the network. Similar to requesting data from the Repo, an executing function can issue a function call by creating an Interest packet and forwarding it to the NDN forwarder to perform further computation, either by the same node or by other nodes in the network. The Figure 3.2 (4) shows an Interest which is sent to initiate a function call on the other node. Then (5), the results returned

by the function calls are forwarded to the requesting function or caller in FL to continue its execution.

Each Interest has its lifetime, or *Time-to-Live (TTL)*, and is only valid within a lifetime period specified by the caller functions or by the consumer. The first time the function execution is initiated by the consumer, it is called the *anchor* of the function call, where the associated lifetime or time-to-live TTL_0 is the *scope* of the consumer function call.

The computation initiated at the anchor is expected to return results within the validity range TTL_0, i.e. TTL_0 spans all subsequent function calls, including recursive calls, and any response or data packet returned to the consumer after TTL_0 expires is considered invalid and discarded. A function execution in FL terminates when its associated TTL expires, whereupon it either returns a result as a data packet to its calling function downstream, or does not respond in the case of an incomplete computation. When the function completes and returns a result, its data packet is sent to the NDN forwarder via Face 3, through which the function call was made, as shown in Figure 3.2 (6). From there, the NDN forwarder removes the corresponding PIT entry, stores the result as data in the CS, and forwards the data packet to the caller via Face 1, through which the user request entered the node.

3.4 ActiveNDN Components

An ActiveNDN node consists of three main components: the NDN forwarder, the Data Repository (Repo), and the Function Library (FL).

3.4.1 NDN Forwarder

The NDN forwarder is a core component that controls the forwarding of Interest and Data packets between multiple Faces as specified by the NDN protocol [Zhang et al., 2014]. Using the information from the PIT and FIB, the NDN forwarder can direct the incoming Interest and Data packets between multiple NDN Faces, where each Face is connecting to the other components to the NDN forwarder. In this way, the Interest packet can be forwarded throughout the network, while the Data packet is returned via a reverse path forwarding mechanism.

In ActiveNDN, the NDN forwarder can forward an Interest to the FL with a function call if there is a pointer to the FL in its FIB record. A network administrator has a role to add the function in FL and configure the function name prefixes with pointers to the FL's Face into the FIB. The *Best-route forwarding strategy*, which is a default configuration in NDN setting [NDN, c], is adopted. The Best-route strategy selects one upstream Face which has the lowest routing cost and forwards the Interest messages to the upstream.

3.4.2 Data Repository

The Repository (Repo) was introduced in NDN as a persistent storage [Zhang et al., 2014]. In many IoT use cases, Repo can be used to store readings from electronic sensors, e.g. temperature, humidity, or PM2.5, CO, CO2 concentrations, etc. To allow data retrieval from the Repo, the name prefixes of sensor readings must be registered with the Repo's Face to the FIB. The Interest messages are directly forwarded to the Repo, in return, the Repo will respond with matching Data messages embedded with sensor readings value.

For IoT applications, to obtain the latest sensor data from the Repo, the *Psync* [Zhang et al., 2017] library is used, which enables synchronizations where the next reading from the data stream is fetched for each data request. This allows any consumer to subscribe for Data under a specific prefix from the Repo. By synchronizing state with Repo, the subscribed consumer will receive the latest Data name and will be notified whenever the producer publishes a new Data to the Repo. After the Repo received an Interest packet from the NDN forwarder, it will prepare a Data packet corresponding to the request and return it to the NDN forwarder. Apart from producing the recent/real-time data, the Repo can also publish the historical data if it has storage capacity.

For example, when a consumer who wants to subscribe to a series of Data with prefix */data/pm2.5* using Psync, it will send a *Sync Interest* in which name contains the Psync prefix and two bloom filters, first one is *Subscription List* containing the prefixes which it wants to subscribe, such as */data/pm2.5*, and the latter is *Producer State* which is blank for the first subscription. The Interest will be forwarded to the Repo, which will compare the received *Producer State* with the internal one. If the consumer's state is behind, it immediately sends a *Sync Reply* Data, containing the updated names with the sequence number of all subscribed prefixes, such as: */data/pm2.5/123* and the updated *Producer State*. But if the consumer's state is already up-to-date, it can keep the *Sync Interest* in a pending list and so the *Sync Reply* will be sent later whenever a new Data under the subscribed prefix is published. When the subscriber received the *Sync Reply* as a notification, it can send an Interest with the latest sequence number */data/pm2.5/123* to retrieve the latest Data from the Repo. It can then use the received *Producer State* to update the subscription for the next data notification.

3.4.3 Function Library

The FL unit is introduced in ActiveNDN to provide a library of functions for in-network computations. The network administrator is responsible for providing functions in FL to serve its specific IoT applications. In a heterogeneous network, the network administrator may consider different design criteria when assigning functions to different nodes in the network. For example, if computational power is important, computationally intensive functions should be placed in powerful nodes, while less powerful nodes can have lighter functions. On the other hand, some applications running on multi-hops network can

generate heavy traffic due to extensive data forwarding. Reducing the redundant traffic is necessary by placing the data aggregation function on nodes located near the data sources.

When FL receives an Interest packet with a request for a function in its library, it initiates a local execution of the function and then attaches the execution results in a Data packet that is returned to the caller. The executed function can make further function calls by creating a new Interest packet, which can be an *external call* to any upstream external nodes or an *internal call* to any functions within the same FL. For example, the *average* function can call the functions *sum* and *count* within the same ActiveNDN node and take the result, or the call can be sent to other nodes in the network. The anchor of a function call will aggregate computation results of all subsequent calls into a final Data packet and sends it back to the NDN forwarder to be forwarded to the consumer via a Face specified in the corresponding PIT entry.

3.5 In-network Computation in ActiveNDN

ActiveNDN node provides computation by allowing the functions in FL to be called and executed within the node. To call a function in ActiveNDN, the caller must create a function-calling Interest and send it to any ActiveNDN node The name of the function to be called and the input parameters of the call are embedded as the name of the Interest, allowing NDN to forward the Interest to the FL where the function code is stored, then the FL can execute the corresponding function.

The FL allows a function to be requested for computing Data or making sub a subsequent request to call other related functions, which are stored in both internally and externally.However, in native NDN, a unique name is required to identify a static content. Multiple Interests from different consumer which are sharing the same name will be forwarded and matched to the same content, which is useful in reducing the latency and traffic in content delivery application (e.g., CDNs). However, this feature is not appropriate with the dynamic content (e.g., real-time IoT applications), as the computation results are spatially and temporally updated depending on the location of ActiveNDN node and execution time. With native NDN architecture, the ActiveNDN node will not be able to identify the state of multiple function-calling Interests at the same name. To enable the in-network computation capability, the caller and callee mapping is crucial for a recursive function to make the returning function output delivered back to the correct caller and to allow the caller to identify the output from different callees. This mapping also prevents the data redundant in the computation, which is important for some type of functions in which the data uniqueness is crucial. However, without identifiable state, the mapping of the caller and callee are not available.

To map the caller and callee pair in ActiveNDN, we propose the use of *nonce*, a unique random number appended to the Interest's name and used as the identifier of the caller

and callee. The range of random number is assumed to be large enough for a computation period which can be assured that any request pair is unique. The FL generates a nonce for each function call and will append it to the Interest name when the function makes a sub-function call and to the output Data name. The ActiveNDN naming scheme utilizes function name as a routable prefix with nonce to make the function-calling Interest and the returning Data to be forwarded correctly.

3.5.1 Expressing Function Call in ActiveNDN

The function call in ActiveNDN is embedded in the Interest packet, and the returning result from the function will be embedded in the Data packet.

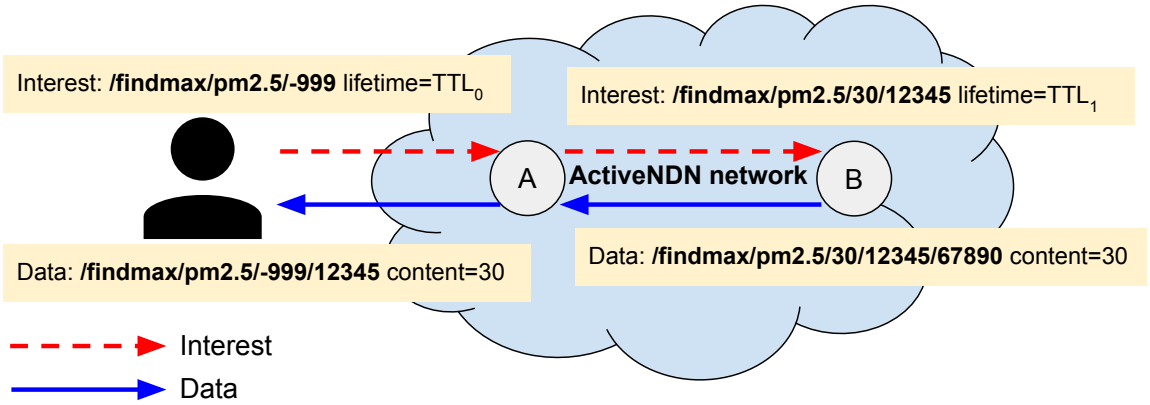


Figure 3.3 – Function call in ActiveNDN

For example, in Figure 3.3, node *A* makes a call to a function *FindMax* to find the maximum *PM2.5* with value higher than *30* and a caller in which nonce is set as *12345*, can be expressed into the name of an Interest as */findmax/pm2.5/30/12345*, and a Data packet which containing the result for the corresponding to this call, will have a name */findmax/pm2.5/30/12345/67890* where the prefix is taken from a full Interest name and the suffix *67890* is the nonce generated by the callee.

The name of function-calling Interest will have three components: the function identifier, calling parameters and nonce, embedded into this pattern:
/<function-identifier>/<parameters>/<caller-nonce>

From the example, the Interest */findmax/pm2.5/30/12345* has function identifier *findmax* as its prefix, following with two parameters: *pm2.5* and *30*, and ending a caller-nonce: *12345*.

The name prefix is important information for NDN to correctly forward the Interest, so the function-calling Interest must take a function identifier as prefix, so NDN can forward the Interest to an FL where the function is available to be executed. As an

Interest can only have one prefix, ActiveNDN allows one function call per one Interest. In this example, the function identifier is *findmax*, so the NDN will forward it to the FL who registered the prefix */findmax* in the FIB to itself for the *FindMax* function.

The function call parameters are embedded as a list of values, which are separated by the hierarchical separator *'/'*. The parameters are pass by value to the function, where the number and type of parameters are conforms to the function specification. In this example, the parameter specification of the *FindMax* function is defined in the program code, it has two variables: ($sensor\_name$:string, max_value:integer). From the Interest */findmax/pm2.5/30/12345*, the two parameters are extracted in the same sequence order: *pm2.5* is the name of the sensor from which the data is to be obtained from which is expressed by variable $sensor\_name$, and 30 is the value for the variable max_value.

Lastly, a field called *caller-nonce* is used as an identifier for each function caller instance and is explained in more detail in the Section 3.5.2. A nonce is generated by FL each time an Interest received, before the function is called. The nonce is assigned and appended to the name of the Interest created by the function process to distinguish different instances of the function caller so that the results can be correctly returned to the respective callers. However, as shown in Figure 3.3, the first anchor call from consumer */findmax/pm2.5/-999*, does not have the caller-nonce as the call is not made by the function in FL.

Apart from the name, the *InterestLifetime* [NDN, a] is a field that can be included in the Interest to specify how long the Interest is valid. Consequently, in ActiveNDN, the *scope* or TTL is used to determine when a function computation is expired or terminates; a small TTL cause an immediate termination of a computation process. The TTL is embedded in the Interest as the Interest lifetime. It is set to TTL_0 by the consumer for the first call or set by an anchor node, which means that the total time consumed by subsequent function calls is within TTL_0. It provides a scope for subsequent function calls, namely, all the computations initiated by this function call must be completed within the specified scope TTL for the results to be valid.

After the function finished its execution, the returning result from the function will be embedded in a Data packet to be forwarded back to the caller. To allow NDN to correctly forward the Data back to the caller, the Data must be matched to its corresponding Interest. To match them, the Data packet returned from the function will have the Interest name as its name prefix. Since some function call may have multiple callees, to allow the caller to be able to differentiate Data between different callee, each callee appends its nonce to the Data name. So, the name of each Data packet will be following this pattern:
/<function-prefix>/<parameters>/<caller-nonce>/<callee-nonce>.

As shown in Figure 3.3, a Data packet which containing the result for the corresponding to the Interest */findmax/pm2.5/30/12345*, will have a name */findmax/pm2.5/30/12345 /67890* where the prefix is taken from a full Interest name and the suffix *67890* is the nonce generated by the callee.

This data name format, with the two nonce numbers, is adopted for all Data packets in the ActiveNDN, whether it is being forwarded by the NDN forwarder or kept in the cache in the CS.

3.5.2 Forwarding Function Calls in ActiveNDN

Functions can be invoked by referring to its name prefix, as described in 3.5.1. When the FL receives an Interest with a function call, it takes the function, assigns a random nonce to the call, and activates its execution. The function execution may initiate further function calls to functions in other nodes as an *external call* or in the same node or FL as an *internal call*. As shown in Figure 3.4, an external function call is forwarded to other ActiveNDN nodes for execution through the Face 0, while an internal function call initiates function execution by the FL. The Face is an abstraction interface of NDN layer to communicate with other modules. As for this example, the Face 0 is set for communication with the physical network interfaces (e.g., Wi-Fi), the Face 1 is used for interacting with application layer, while both Face 3 and Face 4 are configured for FL.

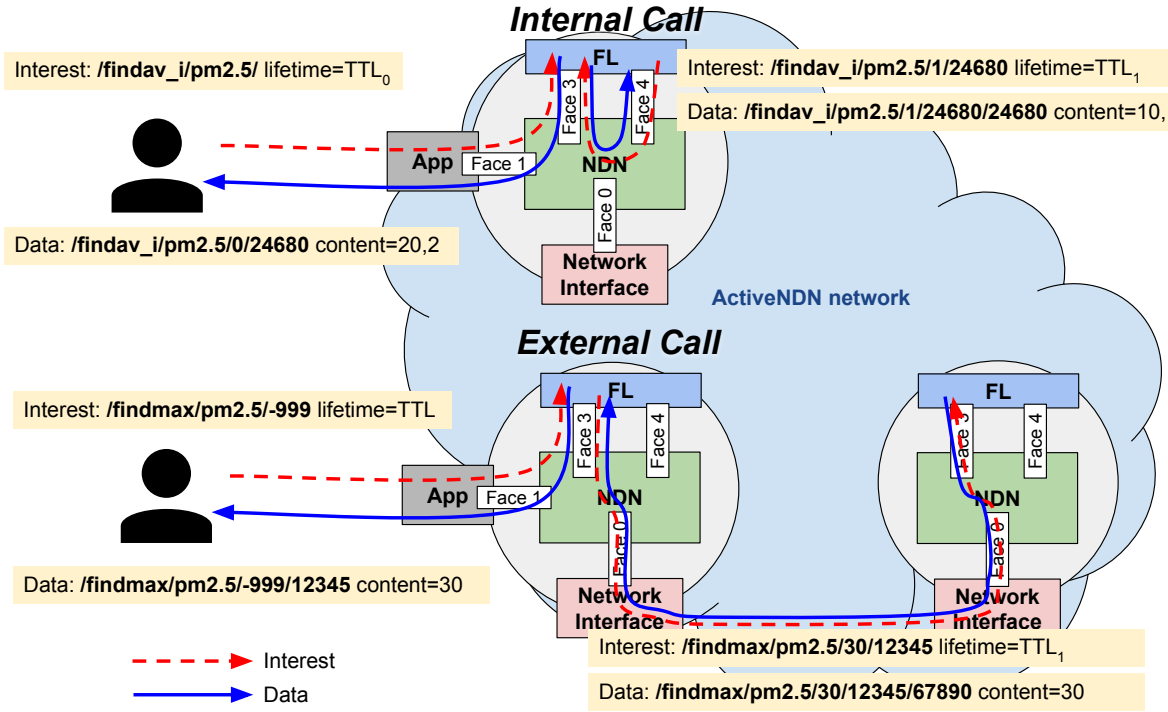


Figure 3.4 – Internal and External function call

For an external call, an Interest is created with an external function call which the prefix is either a function that is not existed in the local FL or the same function, appended with the new nonce of the calling function and forwarded to the NDN forwarder via Face

3. Then, with the longest prefix match on a FIB entry, the Interest is sent to the network interface via Face 0. An entry is also created in the PIT with an incoming Face 3 and an outgoing Face 0. Upon receiving the response from the network, the results or Data are entered into the node via Face 0. If a matching PIT entry of the Data name is found, it will be stored in CS and forwarded to the calling function within FL via Face 3.

For an internal call, similar actions take place, where a new Interest is created for the call along with the calling function's nonce and forwarded to the NDN forwarder. The call should return to the same FL; therefore, the function prefix must be registered in the FIB and point to *Face 3*, not *Face 0*. But with the mechanism in the *Best-Route Strategy* [NDN, c] the forwarder does not route an Interest back to the incoming or downstream Face, i.e. if a call is made from FL, the Interest cannot be routed back to the FL. Therefore, to handle internal calls, a pseudo Face is introduced for FL so that when an internal function call is made, it is sent to the forwarder via the pseudo Face, e.g. *Face 4* in order that the forwarder can forward the Interest back to FL via *Face 3*.

As mentioned in section 3.5.1, the nonce is used to identify function calls and to represent a caller and callee pair in the Data packet. To pair them, FL assigns a nonce for each activation of a function in its library, and FL keeps records of all nonces it generates. For internal calls, the same nonce is kept, while for external calls, a new nonce is generated and appended to the Interest name. When an Interest is forwarded to the FL, its nonce is compared with those in the FL's records. If it is found, it is an internal call and the same nonce can be used throughout. Otherwise, it means it is an external call and the FL generates a new nonce for the new activation for the called function.

From Figure 3.4, a Data packet returned from an external call, *FindMax*, can be: */findmax/pm2.5/30/12345/67890* where 12345 is the caller and 67890 is the callee nonces, while a Data packet from an internal call, *FindAv_i*, will return a data name: */findav_i/pm2.5/1/24680/24680* with two identical nonces.

3.5.3 Recursive Functions in ActiveNDN

The computation in ActiveNDN is considered as recursive network by nature where an ActiveNDN node communicates with several nodes in the network to discover the computing results and return to a requesting user. For instance, if all nodes in the network have a common function (e.g., *FindMax*) installed in the FL, a recursive program can be distributedly executed throughout the network. With such power, the collection, aggregation, and filtering of data can be expressed simply as recursive functions whose execution systematically traverses all nodes in the network. ActiveNDN allows recursion to be performed internally within the same node or externally involving other neighbouring nodes or the entire network, where recursive calls are expressed as an Interest name with a new nonce and a new TTL smaller than the scope TTL_0.

To use recursion over the network, the external call is performed in recursive. Using the same external call mechanism illustrated in Figure 3.4, the Interest is forwarded by

the NDN forwarder to the network interface (Face 0) for broadcasting to the neighbouring nodes. However, a normal FIB configuration may not applicable for external recursive forwarding because the FIB record for the external recursive function has to be mapped to two NDN Faces to FL (Face 3) and network interface (Face 0) which are needed for incoming call and outgoing call respectively. For this, the nexthop cost for Face 3 is set lower than Face 0. With the *Best-Route Strategy* [NDN, c] the NDN forwarder will try to forward an incoming Interest to Face 3 first, as Face 3 has lower cost, except the Interest generated by an executing function in the FL that came to the NDN Forwarder via Face 3, will not be sent back to the same Face, thus Face 0 is automatically chosen. The forwarding is repeated until the scope TTL timeout, each neighbour sends a Data packet back to the calling function node, or if the TTL is exceeded, the computation on that node is failed to complete. If successful, the Data is returned via the network interface (Face 0), checked against the PIT, cached in the CS, and forwarded to the calling function via Face 3 by the normal NDN forwarder; function execution can then continue with the received Data packet. This call distribution enables all reachable nodes to be traversed from the anchor, as long as the total execution time of all subsequent recursive calls is within the initial scope TTL_0 set in the user's request. This mechanism allows us to forward a request as Interest and perform simple recursive function calls within the network without prior knowledge of routing information.

Algorithm 1 FindMax Algorithm

```
    G ← (V, E)
    v ←initial vertex
 1: function findmax(G, v, max_value)
 2:     if v.visited then
 3:         return max_value
 4:     end if
 5:     v.visited ← True
 6:     if v.value ← max_value then
 7:         max_value ← v.value
 8:     end if
 9:     G′ = G − v
10:     for each w adjacent to v do
11:         u ← findmax(G′, w, max_value)
12:         if u > max_value then
13:             max_value ← u
14:         end if
15:     end for
16:     return max_value
17: end function
```

An Algorithm 1 shows how recursion is performed on ActiveNDN with a *FindMax* function on a wireless IoT network. The network is considered as a graph G with $|V|$ nodes and $|E|$ edges, $G = (V, E)$. This function has three inputs: G is the graph that the function is working on, v is the *current node* that is going to check, and max_value is the current maximum value found. Adapted from graph search algorithm, the *FindMax* can be separated into three parts. First, line 2-4, it checks whether the current node v is visited or not, if it is visited, the function terminates and returns current the max_value. Second part, line 5-8, if the node has not been visited yet, it marks the node as visited, then compares the node's value with the current max_value and updates the value if the new result is higher. In the last part, line 9-16, is used where the function recursively call itself on every neighbour nodes. The function call itself with the *current node* parameter changed to each neighbours of v (line 10-11). Each call on a neighbour will repeat the same process which includes checking whether it is visited, comparing the node's value to the max_value, and calling its own neighbours, and thus the function is recursively called into a deeper level until the graph is fully traversed. After the last node is called, the result max_value will be returned to its caller, which it then compares all the results it has received and return the result back to upper level (line 12-16). Finally, the first function will take the results from all returning sub-function call and max_value, compare them, and the maximum value is returned as the function result.

The Algorithm 1 can be transformed to an external recursive function in ActiveNDN, as shown in Algorithm 2, where *FindMax* traverses an air quality IoT network to obtain the maximum PM2.5 reading among all nodes in the network within a fixed time slice TTL.

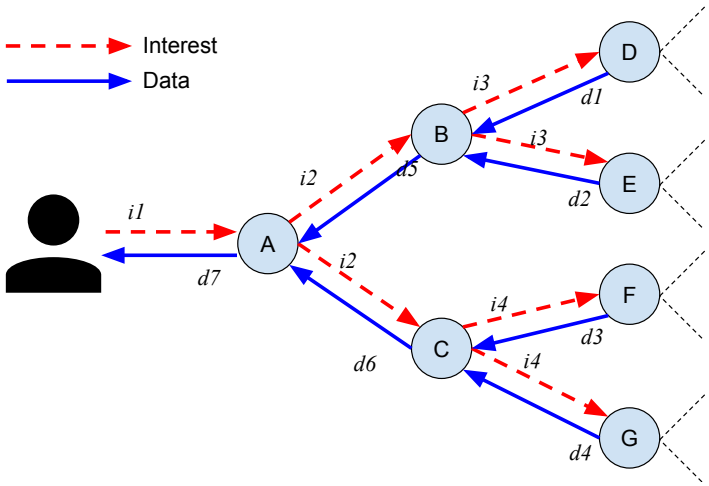


Figure 3.5 – Recursion over network in ActiveNDN

The Algorithm 2 will be explained along with the scenario shown in Figure 3.5. Starting from the first ActiveNDN node (node A in Figure 3.5) that receives the *FindMax*

Algorithm 2 ActiveNDN FindMax Algorithm

Input to NDN forwarder:

`Interest`($/findmax/$pm2.5$/max_v/nonce$ [InterestLifetime = 1000])

Output from NDN forwarder:

`Data`($/findmax/$pm2.5$/max_v/nonce/callee_nonce$)

A: Check if node has been visited by NDN forwarder:

1: **if** Data D with $/findmax/pm2.5$ prefix in CS **then**
2: **return** `Data` from CS $/nonce/callee\_nonce$
3: **end if**
4: **if** Interest with $/findmax/pm2.5$ prefix in PIT **then**
5: **return** Nothing
6: **end if**

B: FindMax in the Function Library retrieves local data and do external function calls:

7: $new\_nonce \leftarrow$ GenerateRandomNumber()
8: $TTL \leftarrow$ ExtractLifetimeFromInterest(I)
9: $Interest\_expire \leftarrow$ CURRENT_TIME + TTL
10: Send $I_{local} \leftarrow$ `Interest`(/data/pm2.5)
11: Receive $D_{local} \leftarrow$ `Data`(I_{local})
12: **if** D_{local}.payload > max_v **then**
13: $max_v \leftarrow D_{local}$.payload
14: **end if**
15: $new\_TTL \leftarrow TTL * reduction\_rate$
16: Broadcast `Interest`(
$/findmax/$pm2.5$/max_v/new_nonce$ [InterestLifetime = new_TTL])

C: FindMax in the Function Library aggregates the returned data:

17: $new\_max_v \leftarrow max_v$
18: **while** in new_TTL period **do**
19: Receive $D \leftarrow$ `Data`(
$/findmax/$pm2.5$/max_v/new_nonce/callee_nounce$ payload=x)
20: **if** D.payload > new_max_v **then**
21: $new\_max_v \leftarrow D$.payload
22: **end if**
23: **end while**
24: **return** `Data`(
$/findmax/$pm2.5$/max_v/nonce/new_nonce$ payload=new_max_v)

Interest (*i1*) issued by a consumer, the function will recursively propagate to all reachable and not yet visited nodes or the entire connected graph or network. To check whether a node has been visited (Algorithm 2 Part A), the incoming Interest is checked against the CS and the PIT. If the requested data is found in the CS, the data value from the CS is returned as a Data packet (line 1-3), and if an entry is found in the PIT, the Interest is discarded (line 4-6). Otherwise, (Algorithm 2 Part B), it means that the node has never been visited by the function, so the forwarder forwards the Interest to the FL and initiates an execution of the requested function, in this case *FindMax*. The function retrieves the data from local repository, compares and updates to the current max value (line 10-14). Then (line 15-16), the function will propagate the next recursion level Interest (*i2* for node A node in Figure 3.5) to neighbour nodes (node B and node C in Figure 3.5). Then during the TTL period (Algorithm 2 Part C), *FindMax* waits and receives the response from the neighbours, and the final result is calculated and returned to consumer when the duration ends. This is shown in Figure 3.5, where the node A receives the response Data *d5* and *d6* from its neighbour nodes B and C, and returns result Data *d7* to the consumer.

As shown in Figure 3.5, multiple nodes can receive an Interest which is broadcasted from a caller node, the caller node must collect multiple Data packets from multiple receivers. At each recursion level, the calling node's TTL or aggregation window is the time in which returning Data packets from subsequent calls are aggregated. This window is supposed to be large to have enough time for collecting the Data packets from all nodes. The lifetime of the PIT entry is set to be equal to the TTL value to allow multiple Data packets to be delivered from all callees, which is a modification to the original design of NDN [Zhang et al., 2014].

To ensure that all callees return the Data packets within the scope of the anchor call, the size of the aggregation window is reduced at each recursion level by multiplying the incoming TTL_{i-1} by a $reduction\_rate$ of less than 1: $TTL_i = TTL_{i-1} \times reduction\_rate$ (line 14 of Algorithm 2). The simplified scale of TTL reduction is shown in Figure 3.6 where the TTL_0 is the scope of the anchor call, and the TTL is reduced in each recursion level. The total time of all calls must fall within the outermost range of the function call. When the TTL becomes too small, the node can no longer forward a new Interest, which means that the termination of the recursion is applied.

3.5.4 Function Scope and Aggregation Window

After an Interest which is sent by the anchor reaches a terminal node with no further call forwarding, the Data will be returned to the caller of the terminal node. Then the caller will perform a data aggregation computation, put the outcome into a Data packet and forward it back to its parent caller; the process is repeated in the reverse Interest path until the Data packet reaches the anchor which it then gives respond to consumer.

Before a node can return the result to its caller, it has to make sure that the returning

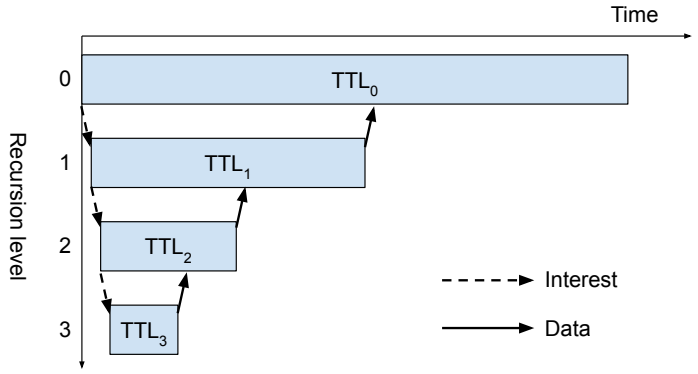


Figure 3.6 – TTL in different recursion level

result is actually computed with most, if not all, data from the callees. However, when network topology is unknown, it is not possible to know if the node itself is a terminal node or how many callees that receive a call and if all callee has returned the result or not.

As stated in 3.3, each Interest has its lifetime, TTL, the termination condition is reached when the TTL of the Interest is insufficient for more computation thus TTL is set to provide a time window for data aggregation at each level of the call.

The scope TTL is used as the time window where a function will wait for the data to return, we call it the Aggregation Window. Aggregation window is particularly useful when deploying ActiveNDN on a wireless network where the information of the network topology is not available.

During the aggregation window, the node will accept the returning results from its callees. When the window of any particular call ends, the caller node will immediately finalize the remaining steps within the caller function, aggregate all the received data and then send the result of its function computation to the upper-level caller. The outermost duration, or the scope of the call, starts from the anchor of the function calls to specify the top-level aggregation window that is applied to cover all computations generated by the anchor.

Through the Interest lifetime, the node i tells its aggregation window to its callees $i+1$ to return a result before a deadline, TTL_i, while each callee set a new aggregation window TTL_{i+1} for its outgoing function calls to be smaller than the incoming Interest lifetime: $TTL_i > TTL_{i+1} + Time_{call} + Time_{return}$ where $Time_{call}$ is the time from when the Interest enters to the node until the time that the function is making a call and $Time_{return}$ for the time taken in the data aggregation process. When the $Time_{call}$ and $Time_{return}$ are uncertain, the aggregation window sizes at each level of the recursive call can also be reduced by multiplying the incoming TTL with a *reduction rate* of less than 1; the total time taken by all calls must fall within the outermost scope of the function call.

When the aggregation window become too small the node will not be able to forward new Interest signifying the termination condition of the recursion.

3.6 Conclusion

In conclusion, ActiveNDN extends NDN to provide in-network computation for processing IoT sensor data. ActiveNDN is designed with three principles: to have ability to make chain of functions, to be lightweight and fully distributed, and to be backward compatibility with native NDN. Using NDN forwarding mechanism, a function can be called by Interest with the function name as routable prefix, and the computation result is returned as a Data packet. With using nonce to identify caller and callee and using TTL mechanisms to specify the scope of the computation, a function program can make a recursive function call to dispatch the execution to other nodes, allowing computation to be fairly distributed to the sensors on the network. Thus, ActiveNDN allows the data collection, aggregation and filtering in IoT application to be distributed and process directly on the IoT sensor devices.

4 ActiveNDN in Wireless Sensor Networks

IoT application scenarios for environmental monitoring, smart metering or incident warning usually require a wireless networking with multi-hop communication. These deployments are commonly located in the rural or remote area where fixed network infrastructure is less available. This specific environment is also called Wireless Sensor Networks (WSN), where the IoT devices or sensor nodes establish wireless connection among themselves.

For ActiveNDN to serve in this kind of environment, each node in the network can utilize the wireless broadcast transmission channel to distribute function calls over the network and receives the computed results on the reverse path forwarding. However, the communication in wireless network is not always reliable because of the nature of the shared-medium communication channel, the effects from network congestion, interference and packet collision lead to packet loss during the transmission. This can significantly impact to the correctness of computation results from data processing because some important data collected from sensors are lost and excluded from the computation function.

In wireless environment, due to shared-medium communication channel, packet collisions can become problematic when simultaneous broadcasting occur. Therefore, providing reliability of packet transmission is vital to improve the completeness of data collection in wireless network conditions. In this essence, ActiveNDN requires additional mechanisms to reduce packet collision, recover from packet loss and reduce network congestion.

In this chapter, we demonstrate the use of ActiveNDN on IoT wireless sensors network with an example of air quality monitoring network. The adjustment of function scope and three mechanisms for improving data collection are proposed as follows; *Random Aggregation Window* to reduce packet collision, *Interest Retransmission* to recover from packet loss and *Interest Exclude Selector* to reduce traffic congestion. To prove that ActiveNDN is applicable for the deployment in wireless sensor network, the prototype of ActiveNDN is developed on IoT devices deployed in a laboratory setting with two simple functions; *FindAv_i* and *FindMax* demonstrating the internal and external function calls respectively. The performance evaluation with large-scale network are explored through extensive simulations aiming at improving the correctness of data processing.

4.1 ActiveNDN Mechanisms for Wireless Communication

To distribute computation in ActiveNDN, the *function scope* is an important parameter which limits the function's execution time. For a chain of function calls, the function scope becomes the first *aggregation window* of the particular function call. It also defines the coverage area of the wireless network that the broadcasting function call can reach. To cover all nodes in the network, the function scope is supposed to be large, which can induce unnecessary delay. However, if the function scope is too small, some nodes would be excluded from the computation. This trade-off needs further investigation to optimize the setting of function call.

On wireless sensor networks, packet loss, caused by packet collision, can happen regularly as multiple nodes may transmit packets at the same time. Furthermore, overhearing induced by shared wireless medium can be the cause of network congestion, as an unintended node can receive and process the packet then transmit the result, which would ultimately lead to unnecessary retransmissions and a high number of packet loss. In ActiveNDN, the reliability of data transmission is crucial as the in-network computation process requires collaborative data from several nodes in the network. Therefore, missing information from data transmission is a serious problem in ActiveNDN, as it can produce incorrect computation results as an outcome. To overcome this challenge, three mechanisms are proposed to support ActiveNDN in wireless sensor network scenario. To prevent the packet collision, the *Random aggregation window* is introduced. Then, *Interest retransmission* aims to recover from the impact of packet loss. Lastly, since ActiveNDN is based on broadcasting, the *Interest Exclude Selector* is applied to reduce unnecessary transmission and minimize the chance of network congestion.

4.1.1 Random Aggregation Window

In ActiveNDN, collisions can happen when all neighbouring nodes make the recursive function call or return their computation results from the recursion and deliver to the caller node at the same time. To mitigate such incidence, a *Random aggregation window* is applied in each function call. Each node individually calculates the new aggregation window (aka. TTL) for a function call. Specifically, the first aggregation window (function scope) will be set by the initiating node, then each subsequent callee node will apply a random reduction rate to reduce the aggregation window. As a result, each ActiveNDN will schedule to return their results at different times and thus naturally avoiding the packet collision. Notice that using randomized reduction may change the network-coverage (i.e. number of hops) of a recursive call. If necessary, the randomness have to be tuned regarding the network density to ensure that the function call can reach to every node in the network.

4.1.2 Interest Retransmission

With packet loss in the network, there is no guarantee that the Interest or a function call sent by a node in the WSN can be received by the intended neighbours, nor is guaranteed that the Data will be received by the caller. To proactively recover from the undetectable packet loss, each ActiveNDN node will re-transmit the Interests or re-call the same function call to retrieve Data that is missing from packet loss.

The retransmission will be scheduled to start after the aggregation windows of all callees expire, with a randomized packet retransmission timer.

In each retransmission, the aggregation window of the retransmitted Interest is also reduced proportional to the time that has passed since the previous transmission.

The Interest retransmission will stop when the aggregation window timer becomes less than 10ms, which is the minimum forwarding interval of NDN forwarder that allows the same Interest to be sent.

With the retransmission, many responses to a retransmitted Interest can be expected because multiple neighbours receive the Interest and respond to it. Some of those responses may have had reached the caller on the previous requests, which can create congestion in the network. Therefore, the next mechanism is required to alleviate such problem.

4.1.3 Interest Exclude Selector

Redundant transmissions caused by overhearing where nodes may receive Interest/Data not intended for them must be reduced. This can be even more critical when there are many Interest retransmissions, a node will respond with a Data that is already received by the retransmitting node. To prevent unnecessary duplicate transmissions of Data packets, an NDN feature called the *Interest exclude selector* [NDN, a] is applied to filter out those Data packets which have already been received by the requester.

As mentioned in Section 3.5.1, the name suffix of each Data packet has the unique random number (i.e., nonce) of the creator. The requester can put all nonces, or suffixes, of the Data packets that have already been received into the Exclude selector field of the retransmitting Interest packet. The Data with its nonce listed in the Exclude selector field will not be transmitted again, thus, suppressing unnecessary transmissions. This is particularly useful when performing a retransmission of the Interest packets; duplicate answers is suppressed at its sources.

4.2 Demonstrating ActiveNDN in Wireless Sensor Network

This section demonstrates how ActiveNDN can be deployed in wireless sensor network environment, with emphasizing the three proposed mechanisms for wireless communica-

tion. The use case of ActiveNDN is illustrated by using a simple function, *FindMax*, to find the maximum PM2.5 readings among sensors nodes. The scenario is divided into two phases, including *Interest traversal* and *Data aggregation*. The two processes are illustrated through an example of wireless sensor network that measures the air quality (e.g., PM2.5 concentration) as shown in Figure 4.1 and 4.2. The topology comprised of 8 sensor nodes ($N20$, $N30$, $N40$, $N50$, $N60$, $N70$, $N80$ and $N90$) and a consumer ($N0$) forming an ad-hoc wireless network, all are integrated with ActiveNDN functionalities. The function *FindMax* is already added in the Function Library (FL) of each sensor node. In addition, the FIB table of each node is also preconfigured with a name prefix */findmax* to route a *FindMax* Interest to the FL.

4.2.1 Interest Traversal

The Interest Traversal is a process to propagate the Interest packets to explore the results from the recursive function call while building a network traversal graph. At the first step, a customer ($N0$) triggers the operation by sending an Interest packet (INT) with a name */findmax/pm2.5/-999* to query a maximum value of PM2.5 from all sensor nodes. The first prefix (*/findmax*) is defined as the name of the function call, while the second prefix (*/pm2.5*) refers to an input parameter of the function. The third component (-999) is a temporal negative value, which is used to compare with PM2.5 values from other nodes. As mentioned in section 3.5.4, the first aggregation window size or function scope (TTL_0) is set by a caller or customer to limit the duration of waiting time. To implement this feature on native NDN structure, the *function scope* is embedded in the Interest packet via the lifetime field. This information informs a callee the time limit for returning the data back within a defined scope. In this example, the function scope (TTL_0) is specified as 1000 ms to ask for the maximum PM2.5 reading from the network within a chosen time slice.

Following in the second step, where the closest ActiveNDN node ($N20$) receives the Interest packet from the consumer to query the maximum PM2.5 reading. The NDN forwarder of $N20$ records an incoming Interest in its PIT table and forwards the prefix */findmax* to its Function Library (FL) while selecting the function *FindMax* for execution. Now, $N20$ becomes the anchor of the function call with a scope of 1000 ms (TTL_0) for the entire computation. It reads a PM2.5 value from its persistent storage (Repo). Let us suppose that the PM2.5 sensor in $N20$ gives a value $20\mu g/m^3$ which is higher than an incoming PM2.5 value (-999) attached in the Interest packet. Thus, the *FindMax* function updates the maximum PM2.5 value to 20. To further propagate the calls, the function *FindMax* in *N20* broadcasts its recursive call to all neighbours with a caller's nonce and a random size of aggregation window. With only the initial name prefix */findmax/pm2.5/value*, the callees (i.e., node receiving the function call) cannot identify who is the caller. Therefore, a ***caller's nonce*** (a randomly generated number, represented as $n20$) is also attached in the name prefix. This allows the callees to identify different

function calls and let the NDN forwarder to deliver the computation result back to the caller correctly. Notice that each caller's nonce must be unique within a duration of the function scope, which can be achieved by having a very large randomize sample space which minimize the possibility of getting the same nonce.

Consequently, the ***aggregation window*** defines how long a function call can wait for a computed result. Having different aggregation window size for a function call in each node will make the function to return result in different time, and so preventing packet collision in the wireless networks. While the node *N20* is an anchor of the call where it is not likely to cause a collision as it is the only node who can return the Data to the consumer. This node can have a constant aggregation window reduction rate. For the sake of simplicity, the same reduction method is applied for every function call throughout the demonstration. The aggregation window is reduced by multiplying the input lifetime (TTL_0) with a random reduction rate, which is randomized between range $[0.3, 0.7]$. Assuming that, the randomization in this call gives a reduction rate of 0.543 as a result and the new TTL becomes 543 ms. A new Interest packet with the name prefix */findmax/pm2.5/20/n20/* appended with the current maximum PM2.5 value together with a caller's nonce ($n20$) and a new lifetime from the aggregation window size ($543ms$) is created and forwarded back from FL to NDN Forwarder. Then the NDN forwarder forwards the Interest packet to the wireless interface and broadcasts it to nearby nodes. The *FindMax* function in $N20$ will wait for incoming Data packets from its neighbours until the timing specified in aggregation window is expired.

The third step is proceeded by nodes located at the second hops away from the consumer, which are $N30$, $N40$ and $N50$. Upon receiving the Interest packet from $N20$, the NDN forwarder of each node records the Interest in their PIT tables and forward the Interest packet to the FL, which it then calls the *FindMax* function. The function reads a PM2.5 value from its persistent storage (Repo) with values of "$30\mu g/m^3$", "$40\mu g/m^3$" and "$50\mu g/m^3$", for nodes $N30$, $N40$ and $N50$ respectively. After updating the current maximum PM2.5 value inside the *FindMax* function, each node creates a new Interest packet with the same prefix, */findmax/pm2.5* while updating a new max value and the new caller's nonce. For this example, the new Interest packet of $N30$ is set as */findmax/pm2.5/30/n30*. As mentioned in section 4.1.1, the packet collision can be occurred, if multiple nodes return the computed results simultaneously. Therefore, the ***Random aggregation window*** is applied in this step by generating a new aggregation window size which is randomly reduced from the incoming Interest's lifetime. As a result, the new aggregation windows of $N30$ (TTL_{N30}), $N40$ (TTL_{N40}) and $N50$ (TTL_{N50}) are assigned as 240 ms, 243 ms and 265 ms respectively. The new Interest packets of $N30$, $N40$ and $N50$ are presented as $3.a$, $3.b$ and $3.c$ while the TTL value is embedded in the lifetime field of the Interest packet as shown in Figure 4.1. The Interest packets are broadcasted by each node and due to the overhearing in wireless communication, all neighbour nodes nearby can receive the Interest. For example, an Interest packet which is broadcasted

from $N50$, will be received by $N20$, $N40$, $N60$, $N70$ and $N80$. However, the nodes where the *FindMax* function is already started and is still waiting for result Data packets, i.e. $N20$ and $N40$, will discard the Interest, as the */findmax/pm2.5* prefix can be found in the PIT of these nodes.

At the fourth step, all three hops away nodes from the consumer ($N10$, $N60$, $N70$ and $N80$) receive the Interest packet from its upstream caller. Note that in this example, there is an unreachable node $N90$ which cannot receive any Interest from any other node on the network; thus it is not included in the computation. The *FindMax* operation is repeated as the previous step while the new Interest packet is updated with new max value, caller's nonce and aggregation window size which is randomly reduced from the upstream. The recursive function call will continue to propagate the Interest packets to all reachable nodes or until the aggregation window becomes too small (lower than 10ms). In other words, the recursion termination condition (no more adjacent node) is reached, and the execution is returned to the upstream callers. The Interest propagation paths are distributively recorded as the PIT entries on the nodes that took part in the recursion can be viewed as an *Interest traversal tree*, as shown in Figure 4.1.

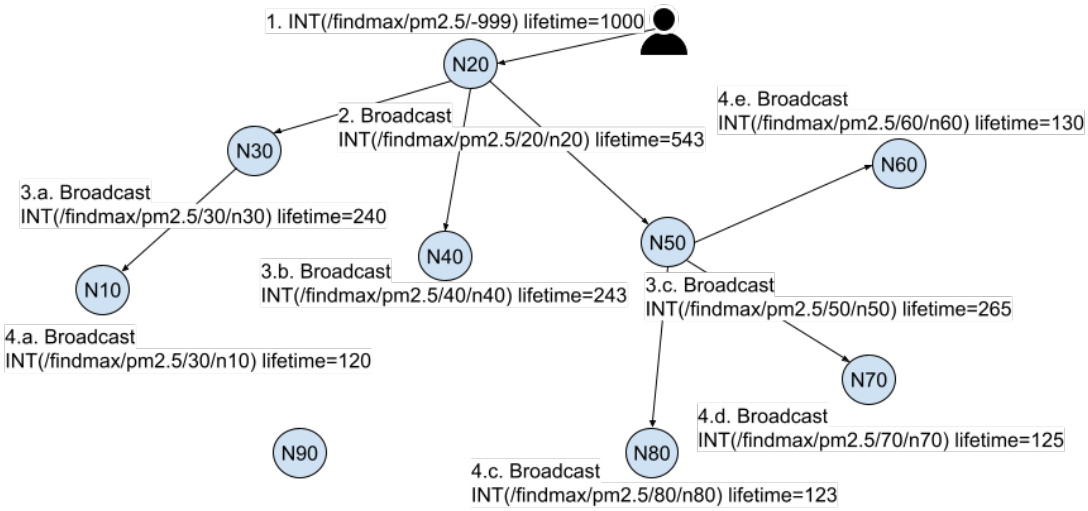


Figure 4.1 – Interest traversal tree of a recursive function call.

4.2.2 Data Aggregation

The data aggregation process is started when all neighbours nodes have already been visited. The recursive function call starts aggregating from each leaf node which returns a Data packet back to the anchor node ($N20$) following the reverse path of *Interest traversal tree* as shown in Figure 4.2. In this example, node $N80$ which is a leaf node of the tree is considered. When the aggregation window (TTL_{N80}) is expired, $N80$ will transmit a Data

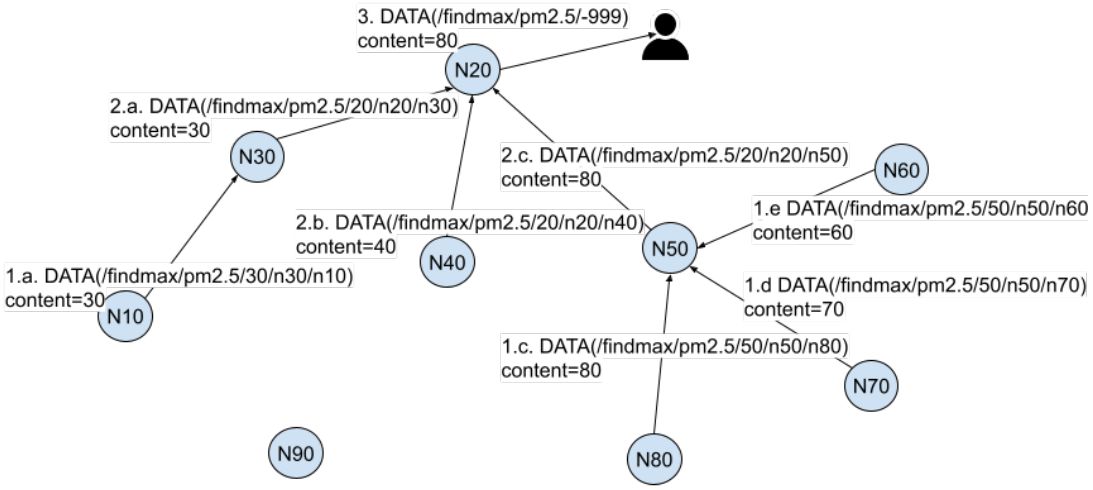


Figure 4.2 – Data aggregation tree of a recursive function call.

packet */findmax/pm2.5/50/n50/n80* with the PM2.5 reading of "$80\mu g/m^3$" to its caller which is $N50$. The name of the Data packet is taken from the Interest and appended by the node's nonce, i.e. callee's nonce (represented as $n80$).

With broadcast transmission, many responses to an Interest are expected from multiple neighbours who are callees. Thus, the PIT entry cannot be removed after the first Data is received, but it has to be kept alive until TTL is expired to receive as many responses as possible. In the original design of NDN [NDN,], it was designed to match one Interest packet per one Data packet (1-to-1 Interest and Data matching) so unsolicited and duplicated Data are avoided. The design is implemented with a mechanism: deleting the matching PIT entry upon receiving an incoming Data, causing the next corresponding incoming Data packets that are matching the same Interest to be discarded by the NDN. Removal of this restriction was introduced in [Amadeo et al., 2014b] for IoT applications for collecting data from multiple data sources. In this example, the caller node $N50$ should receive Data packets from its callees: $N60$, $N70$, and $N80$. In node $N50$, these packets are checked if they are found to correspond to entries in the PIT table of $N50$ before forwarding to the pending *FindMax* function in the FL of $N50$.

Due to unreliability of wireless condition, the returning Data packet can be lost during transmission. To handle the packet loss, ActiveNDN applies the ***Interest Retransmission*** mechanism by sending the Interest packet after the aggregation window of all callee is expired, which can be estimated from the reduction rate. For instance, the node $N50$'s aggregation window is 265ms, the maximum possible aggregation window of the callee can be $265 \times 0.7 = 185.5$ms, where the multiplier 0.7 is the possible maximum value of the random reduction rate. Notice that, network information is unavailable, the node $N50$ itself will not know the actual number of its callees, so the retransmission starts 185.5ms

after the first Interest is sent and keep going until its aggregation window ends.

As the retransmitted Interest is broadcasted, all the callees will response with its own Data packet, this includes the Data that has already receive by the caller which is unnecessary and cause congestion. To suppress this unnecessary Data, the caller put an ***Interest Exclude Selector*** which contains a list of name suffixes, i.e. the callee's nonce, of the already received Data packet, into the retransmitting Interest. In this case, if $N50$ has already received Data from $N60$ and $N70$, it can retransmit an Interest */findmax/pm2.5/50/n50 exclude=n60,n70* which will not match the Data */findmax/pm2.5/50/n50/n60* and */findmax/pm2.5/50/n50/n70*, thus the transmissions from $N60$ and $N70$ are suppressed.

The *FindMax* function in $N50$ ultimately selects the maximum value among all received Data packets and returns the results as Data named */findmax/pm2.5/20/n20/n50* with the value "$80\mu g/m^3$" via its NDN forwarder through the Network Interface to $N20$.

As a consequence, the node $N20$ computes the final *FindMax* and answers to the query from all answers from reachable ActiveNDN nodes in the network. After processing all incoming data in the same manner as in other nodes, the result obtained in $N20$ will be transmitted to the consumer, as shown in Figure 4.2.

4.3 Implementing ActiveNDN Prototype

To prove that the design is applicable, the ActiveNDN proof-of-concept prototype is implemented on real IoT devices and experimented in laboratory setting. An ActiveNDN IoT prototype was developed with two functions: "*FindMax*" and "*FindAv_i*", as described in Section 3.5.2, to demonstrate external and internal calls respectively. We put "*_i*" at the end of a function name to indicate that the function is an internal recursive function.

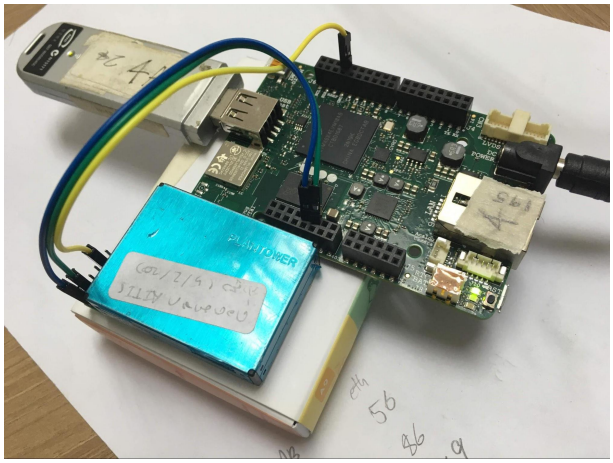


Figure 4.3 – An ActiveNDN node in the testbed

A prototype of ActiveNDN node is developed on UDOO NEO FULL [UDOO,], a single board computer equipped with an NXP i.MX 6SoloX which is embedded with 2 cores CPU: an 1Ghz ARM Cortex-A9 core and a 200Mhz ARM Cortex-M4 core, 1GB of RAM and 16 GB SD card storage. The PM sensor Plantower PMS7003 and external 802.11g Wi-Fi USB dongle Linksys WUSB54GC are also integrated to collect the PM2.5 value and create wireless ad-hoc connection respectively. Figure 4.3 shows the prototype of an ActiveNDN developed on UDOO NEO board.

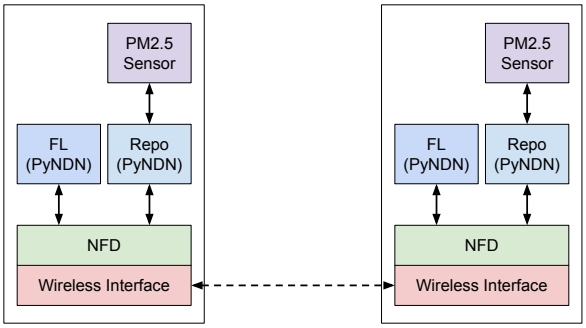


Figure 4.4 – The node architecture of ActiveNDN prototype

We use NFD (Named Data Networking Forwarding Daemon) [NDN, b] as our NDN Forwarder and use PyNDN library [NDN,] to implement the Function Library (FL) and Repository (Repo) as illustrated in Figure 4.4. In this prototype, two functions are implemented: an external recursive function, *FindMax* which finds the maximum data from the network and an internal recursive function, $FindAv\_i$ which calculates the average value within an ActiveNDN node. The Repo is implemented to store the persistent data and with publish-subscribe capability using the PSync (Partial-Synchronization for NDN) [Zhang et al., 2017] from the PyNDN library. The Repo can respond to a matched Data of the incoming Interest as a simple NDN Repository and also accepts Psync-subscribe message and pushes an update to a subscriber whenever a new data is added to the Repo. The PM sensor is configured to capture a PM2.5 value for every one second. The reading value is recorded in the Data packet with name prefix $/data/pm2.5/(timestamp)$ and stored in the Repo.

Four ActiveNDN nodes are placed around a laboratory at 10-15 meters from each other, which is too close for multi-hop experiment as every node can be reached within a single transmission. To achieve a multi-hop network environment in this small area, the Wi-Fi transmission power is lowered to 5 dBm to reduce the transmission range and create a multi-hop scenario in the network, as shown in Figure 4.5.

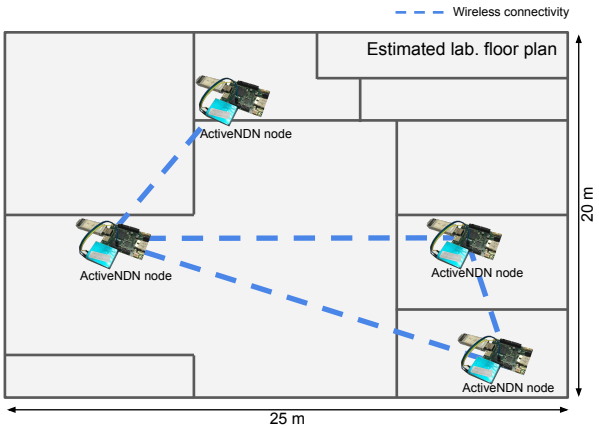


Figure 4.5 – The nodes placement and wireless connectivity in test-bed

4.3.1 FindAv Internal Function

The *FindAv_ i* is a function to find an average of a specific measurement, which is executed locally within an ActiveNDN node. The function library is recursively called itself to get a new update data while computing an average result. The function can be called by an Interest:

```
/findav_i/(value_name)/(sequence_number)/(caller-nonce) lifetime=TTL
```

Two parameters are embedded in the Interest name: the name of the measurement to be averaged and a sequence number of the last data that has been collected. The *caller-nonce* is an optional component which can be either user define or automatically generated by the FL. This caller-nonce will be used throughout the whole computation to return data back to the appropriate caller. Notice that the *FindAv_ i* is able to calculate the average value of any measurement stated in value field, but in this proof-of-concept, only the PM2.5 value is considered.

The function runs recursively within a node to retrieve the latest data from the Repository. The Interest is sent from the function in the FL and bounced back to call the same function again.

In each call initiation, the lifetime of an Interest is decreased by the time that has passed, which includes the time for processing and the time for retrieving the data from the Repository. Since each function call invokes only one function in each local node (only one callee), the aggregation window can be reduced by a constant value. The new Interest can be updated with a new lifetime (TTL_i) as follows:

$$TTL_i = TTL_{i-1} - t_{local} - t_{return} \tag{4.1}$$

where TTL_{i-1} is the lifetime of incoming Interest $i-1$. The t_{local} is the time taken from the arrival of the incoming Interest to the time of sending new Interest i to the NDN

forwarder. This duration includes the time used in retrieving a new Data Repository, which can take up to one second to waiting for a new value from the PM sensor. The t_{return} is a compensation time for the delay of forwarding the Data back to the caller, which is arbitrarily set to 100 ms. When the aggregation window is expired or the first response Data is received, the function call will resume the execution and return the result as Data packet back to its caller.

The function *FindAv_i* is tested on the testbed by sending an Interest packet with name: */findav_i/pm2.5/0 lifetime=10000* from a consumer application to one of the ActiveNDN node located in the laboratory. The *pm2.5* specifies the function to take average the PM2.5 values, and the sequence number, initialized as 0 informs the last sequence of PM2.5 Data that it has collected. From the lifetime field, the function scope is set as 10 seconds (10,000 ms). Thus, the function will recursively collect the average PM2.5 values from an internal sensor over the next 10 seconds duration and then will return a Data packet containing the average PM2.5 values and the number of samples.

```
i1 /findav_i/pm2.5/0 lifetime=10000
RepoData1: /data/pm2.5/%5E%8D%DC%8C content=12
i2 /findav_i/pm2.5/%5E%8D%DC%8C/8%B2%8A%B5 lifetime=9165
RepoData2: /data/pm2.5/%5E%8D%DC%8D content=12
i3 /findav_i/pm2.5/%5E%8D%DC%8D/8%B2%8A%B5 lifetime=8220
RepoData3: /data/pm2.5/%5E%8D%DC%8E content=12
i4 /findav_i/pm2.5/%5E%8D%DC%8E/8%B2%8A%B5 lifetime=6350
RepoData4: /data/pm2.5/%5E%8D%DC%8F content=12
i5 /findav_i/pm2.5/%5E%8D%DC%8F/8%B2%8A%B5 lifetime=5330
RepoData5: /data/pm2.5/%5E%8D%DC%90 content=12
i6 /findav_i/pm2.5/%5E%8D%DC%90/8%B2%8A%B5 lifetime=4262
RepoData6: /data/pm2.5/%5E%8D%DC%91 content=12
i7 /findav_i/pm2.5/%5E%8D%DC%91/8%B2%8A%B5 lifetime=3306
RepoData7: /data/pm2.5/%5E%8D%DC%92 content=12
i8 /findav_i/pm2.5/%5E%8D%DC%92/8%B2%8A%B5 lifetime=2361
RepoData8: /data/pm2.5/%5E%8D%DC%93 content=12
i9 /findav_i/pm2.5/%5E%8D%DC%93/8%B2%8A%B5 lifetime=1317
RepoData9: /data/pm2.5/%5E%8D%DC%94 content=12
i10 /findav_i/pm2.5/%5E%8D%DC%93/8%B2%8A%B5 lifetime=375

--i10 expires--
d9 /findav_i/pm2.5/%5E%8D%DC%93/8%B2%8A%B5/8%B2%8A%B5 content=12.0,1
d8 /findav_i/pm2.5/%5E%8D%DC%92/8%B2%8A%B5/8%B2%8A%B5 content=12.0,2
d7 /findav_i/pm2.5/%5E%8D%DC%91/8%B2%8A%B5/8%B2%8A%B5 content=12.0,3
d6 /findav_i/pm2.5/%5E%8D%DC%90/8%B2%8A%B5/8%B2%8A%B5 content=12.0,4
d5 /findav_i/pm2.5/%5E%8D%DC%8F/8%B2%8A%B5/8%B2%8A%B5 content=12.0,5
d4 /findav_i/pm2.5/%5E%8D%DC%8E/8%B2%8A%B5/8%B2%8A%B5 content=12.0,6
d3 /findav_i/pm2.5/%5E%8D%DC%8D/8%B2%8A%B5/8%B2%8A%B5 content=12.0,7
d2 /findav_i/pm2.5/%5E%8D%DC%8C/8%B2%8A%B5/8%B2%8A%B5 content=12.0,8
d1 /findav_i/pm2.5/0 content=12.0,9
```

Figure 4.6 – Sequence of Interest and Data message of FindAv internal

The experiment was conducted for 10 seconds in which 9 samples of PM2.5 were read. However, the measurement was conducted inside the laboratory, so the PM2.5 values were remained at $12\mu g/m^3$ throughout the experiment. Figure 4.6 presents the sequence of Interests and returning Data messages of the recursive function call from a node in the testbed. The Interests: $i1$, $i2$, $i3$, ..., $i10$ calling the function *FindAv_i* are satisfied by subsequent Data packets: $d1$, $d2$, $d3$, ..., $d9$ in respected order. The shown sequence of the function can be explained as follows: The Interest $i1$, which is sent from the consumer to the closest ActiveNDN node, initiates the call to the *FindAv_i* function in FL. After invoked by the call, the function then retrieves the latest PM2.5 measurement from the Repo using Psync protocol, the received Data is shown as *RepoData1*. Then, the function creates an Interest $i2$ with a name pattern: */findav_i/pm2.5/(sequence_number)/(nonce) to make a recursive call to itself.* The sequence number in $i2$ is taken from the hexadecimal-encoded timestamp of the retrieved *RepoData1* (*%5E%8D%DC%8C*). Since Interest $i1$ does not have a nonce, the FL randomly generated one which is *8%B2%8A%B5* in hexadecimal and associates it to this call. As an internal recursive function, the *FindAv_i* uses this nonce through the whole of its following function call, including $i2$.

The Interest $i2$ is sent to call the same function, which means the process is repeated: retrieving the PM2.5 data from Repository, and making a subsequent call. As the function is calling itself recursively, the Interest $i3$, $i4$, $i5$, ..., $i10$ are then produced by each call where the TTL are reduced successively.

Finally, the Interest $i10$ which has the smallest TTL gets timeout before a new Data can be retrieved from Repository, the function which is called by $i10$ is terminated itself without a reply. When $i10$ is expired, the function that creates $i10$ immediately resumes execution and returns the Data $d9$. The Data $d9$, which contains the average PM2.5 value and number of samples, satisfies Interest $i9$ and returns to the $i9$ caller. The $i9$ caller function receives the Data and uses the information from $d9$ to calculate a new average and return them as $d8$ Then, $d8$ that satisfies $i8$, is sent to the $i8$ caller which then returns $d7$, and so on, until the Data $d1$ is returned to the consumer as the final result.

Regarding the result shown in Figure 4.6, the PM sensor can read the measurement of PM2.5 at 1 sample per second. However, only 9 samples were collected in 10 seconds. This is because the lifetime of each Interest is chosen to be reduced by a constant $t_{return} = 100ms$, to compensate the forwarding delay. This was made to complete the Data return before the caller Interest timeout, or else the Data would have been dropped by the PIT because of the timeout.

4.3.2 FindMax External Function

Next, the *FindMax* function, which is capable of finding the maximum value of any specific measurement, is introduced. Similar to the *FindAv_i* function, this prototype focuses on the PM2.5 sensor readings. However, the *FindMax* is designed for the external function call where the Interest packets are sent out to other ActiveNDN nodes as described in

Section 3.5.2 and Section 4.2. The function can be called by an Interest packet with a name prefix:

`/findmax/(value_name)/(max_value)/(caller-nonce) lifetime=TTL.`

As an external function, the aggregation window (TTL_i) is randomly reduced and calculated by the following equation:

$$TTL_i = TTL_{i-1} * random(min, max) \tag{4.2}$$

where the TTL_{i-1} is the aggregation window of the function call given by the lifetime of incoming Interest while multiplying with the random function. In this experiment, the boundary of random function (min and mix) is set between 0.3 and 0.7. The floor plan with node's locations and the expecting packets are shown in Figure 4.7.

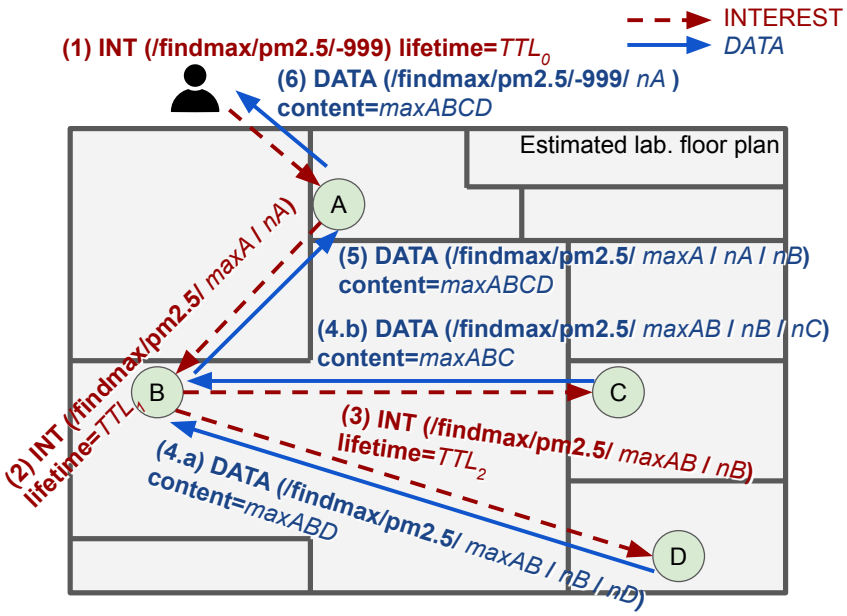


Figure 4.7 – Testbed node placement and visualized packets

An Interest packet with name prefix */findmax/pm2.5/-999 lifetime=10000* is injected to the node A who is the anchor node of this call. Consequently, this Interest packet is forwarded to the Function Library of node A to activate the function *FindMax*. With Interest Traversal and Data Aggregation processes explained in section 4.2, the function call is traversed recursively throughout the network while aggregating the maximum value of PM2.5 from multiple nodes. As the function scope is set as 10,000 ms in the lifetime field, the aggregated result will be returned within the time scope of 10 seconds.

The sequence of Interest and Data packets from the *FindMax* experiment is illustrated in Figure 4.8. The Interest $i1$ which is sent to the node A, calls the *FindMax* function in

```
i1 /findmax/pm2.5/-999 lifetime=10000

A receive i1
A's RepoData: /data/pm2.5/%5E%90%11%F6 content=35
A send i2 /findmax/pm2.5/35.0/%CB%F9%80%EC lifetime=4668

B receive i2
B's RepoData: /data/pm2.5/%5E%90%11%F7 content=37
B send i3 /findmax/pm2.5/37.0/%D8%BC%2A%3B lifetime=1844

C receive i3
C's RepoData: /data/pm2.5/%5E%90%11%F8 content=24
C send i4 /findmax/pm2.5/37.0/F%D1%15%D3 lifetime=325

D receive i3
D's RepoData: /data/pm2.5/%5E%90%11%F8 content=27
D send i5 /findmax/pm2.5/37.0/b%D6%BE%CA lifetime=125

-- i5 expires --
D send d3.1 /findmax/pm2.5/37.0/%D8%BC%2A%3B/b%D6%BE%CA content=37.0
B receive d3.1

-- i4 expires --
C send d3.2 /findmax/pm2.5/37.0/%D8%BC%2A%3B/F%D1%15%D3 content=37.0
B receive d3.2

-- i3 expires --
B send d2 /findmax/pm2.5/35.0/%CB%F9%80%EC/%D8%BC%2A%3B content=37.0
A receive d2

-- i2 expires --
A send d1 /findmax/pm2.5/-999 content=37.0
```

Figure 4.8 – Sequence of Interest and Data message of FindMax

the node. The called function takes the *max_value* from the Interest which is -999, then retrieves the latest PM2.5 values from the Repo with the Psync protocol and compares the new *max_value* which is $35\mu g/m^3$. Then, it generates a new nonce and build a new Interest $i2$ then broadcast to neighbours. During this experiment, the 4 nodes provided different PM2.5 readings which are $35\mu g/m^3$, $37\mu g/m^3$, $24\mu g/m^3$, and $27\mu g/m^3$ from nodes A, B, C and D respectively. Node A which is the anchor node of this call returns the correct maximum value which is $37\mu g/m^3$ when the function scope is expired (see the last line of Figure 4.8)

Through the experiments with *FindAv_i* and *FindMax*, it has proven that the prototype of ActiveNDN can operate properly with both internal and external recursive function call. The calling sequence of the two functions correctly reproduced to what was designed in section 3.5.2 and provided correct answers. Besides, the prototype has been

implemented on the real IoT platform, which will be a base for further development with complex applications.

4.4 Large-Scale Evaluation on Network Simulator

To understand how ActiveNDN operate in WSN at scale, its performance is evaluated using the official NS3-based NDN simulator, called ndnSIM [Mastorakis et al., 2017].

4.4.1 Simulation Setup on ndnSIM

The performance of the proposed ActiveNDN is studied with a wireless network environment using the standard IEEE 802.11 Wi-Fi model on the NS3-based NDN simulator, ndnSIM-2.7 [Mastorakis et al., 2017], with the standard IEEE 802.11 Distributed Coordination Function (DCF) for CSMA/CA [NS3, b] and a constant bit rate of 1 Mbps in ad hoc mode. The *Range propagation loss model* and *Constant speed propagation delay model* are used to simulate the transmission range of 250 meters. Each node was configured with a FL containing the *FindMax* function, as described in the example in section 3.5.3, and a Repo which is publishing a PM2.5 value. The PM2.5 data in the Repo of each node is uniquely defined once before each simulation. The simulated CPU/hardware processing time is not included and assumed as zero in the simulation.

A consumer application sends an initial Interest packet to a random node to invoke *FindMax* with a function scope of 10 seconds (TTL_0) while reduction rate is set to a 50% constant ratio; thus the new aggregation window is set to half of the incoming Interest lifetime each time. The updated of new aggregation window of caller (TTL_i) is calculated as follows:

$$TTL_i = TTL_{i-1} * 0.5 \tag{4.3}$$

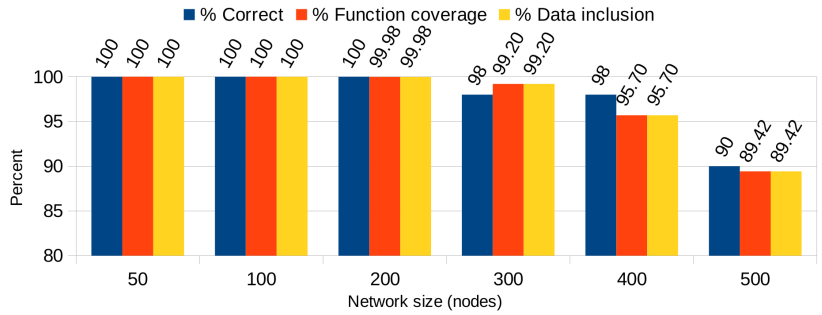
where TTL_{i-1} is the previous caller's aggregation window size (from the lifetime field in the incoming Interest).

4.4.2 Validation of the Simulation

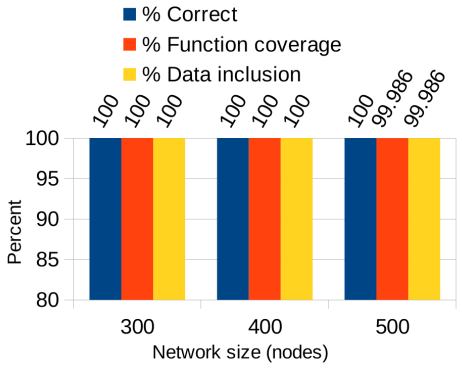
The implementation of simulation is validated on an ideal wireless network without wireless packet collision. To simulate the ideal wireless network without packet collision, the NS3's Wi-Fi model is modified to disable the packet collision.

With dynamic wireless networks and distributed computation on ActiveNDN, our goal is to find correct results within a certain period of time. The *FindMax* function was simulated with the most basic baseline configuration to compare the result correctness between different network sizes of 50, 100, 200, 300, 400 and 500 nodes in which the nodes are randomly distributed within the areas of 0.5, 1, 2, 3, 4 and 5 km^2 respectively. The *FindMax* function works recursively to find the maximum PM2.5 value in the network.

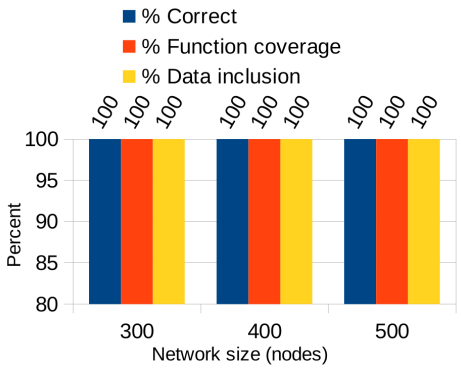
For the simulation, each node in the network is assigned to produce a unique PM2.5 value and the value is to be used throughout the simulation. Each simulation is repeated 100 times with different random node placement. The evaluation targets to analyse three performance metrics as follows; the percentage of *correct results* by counting the number of experiments where the final answer of *FindMax* function is equal to the actual maximum value of all nodes in the network, the *Function Coverage* set which is the number of nodes reachable by the anchor nodes, and the *Data Inclusion* set, or nodes that contribute to the computations that produce the final answers.



(a) 10,000ms function scope



(b) 200,000ms function scope



(c) 500,000ms function scope

Figure 4.9 – Validation results with different function scope

The results of simulation are validated by comparing the actual values from each node and the results of function *FindMax* running on ActiveNDN. The result of the validation is shown in Figure 4.9. In Figure 4.9a, with a function scope configured as 10 seconds, all three sets achieved 100% performance. Small percentages of incorrect answers and discrepancies between the *Function Coverage* and *Data Inclusion* sets began to appear when the network size exceeded 200 nodes and the scope window was increased to 500 seconds, after which the 100% performance could be achieved again. This means that some nodes is unreachable if the function scope is set too small; or some nodes may be completely isolated or out of range, as shown in Figure 4.1. Even if there is only one

node in the network that cannot be reached by the propagation of the Interest, there is a possibility that the computed answer may be incorrect, especially for *FindMax*. In some cases, the *FindMax* function can still provide a correct answer, even if there are a small subset of reachable nodes. This is because the nodes which are unreachable have lower value than the reachable nodes, and thus the uncovered value is not effected to the correctness of computation. However, this is not applicable for the function that requires the high data inclusion, such as the finding average function (*FindAv*) where missing data significantly impacts on the correctness of computation. On the other hand, the large function scope setting could be considered when the network size is large. As shown in Figure 4.9c, the function scope of 500,000 ms can achieve 100% of function coverage and data inclusion, even if the network size is increased to 500 nodes.

The selection of function scope size is crucial as it effects to the accuracy of the result. Even though, the function scope can be set with a large number to ensure that all nodes are reached. However, the computation may suffer from long waiting time for collecting an answer. Therefore, selecting an appropriate function scope to match with the network size is crucial for the configuration of ActiveNDN. We further ensure the correctness of our implementation by checking if the data for the computation are retrieved from the network correctly by comparing the nodes in the *Function Coverage* and *Data Inclusion* sets which return the raw sensor data, as already shown in Figure 4.9.

By disable the collision model in Wi-Fi simulation, the results confirms the implementation of ActiveNDN simulation is valid and shows that the TTL impacts the accuracy of the computation in different network scale, which will be discussed later in section 4.5.

4.4.3 Performance Evaluation

In this section, the performance of ActiveNDN with baseline setting is evaluated and compared with the three proposed mechanisms; random aggregation window, Interest Retransmission and Interest exclude selector.

We measure the effect of packet collision to ActiveNDN in normal wireless condition while comparing performance improvement from the proposed mechanisms.

The same baseline configuration is experimented with the same function scope size as in the validation runs, but with a normal Wi-Fi model, where packet collisions can be occurred. The packet loss rate is measured in each experiment by comparing the ideal and actual total number of received packets. Since the *Range propagation loss model* for limiting the transmission range is applied, the nodes located within the transmission range is known from the distance between the transmitting node and the other nodes. The ideal total number of received packets $n(R_{ideal})$ is known by accumulating the number of transmission multiply by the number of the nodes located within the transmission radius of each corresponding transmission. The actual total number of received packets $n(R_{actual})$ is accumulated from the actual number of node that received each transmission. Finally, the measured packet loss rate is calculated from $(n(R_{ideal}) - n(R_{actual}))/n(R_{ideal}) *$

Table 4.1 – Overall measured packet loss rate (%)

Network size (nodes)	**50**	**100**	**200**	**300**	**400**	**500**
Baseline	76.88	74.45	70.73	68.51	67.70	66.31
Random AW	38.81	38.51	36.65	36.41	36.33	35.86
Random AW + RTX	14.68	14.75	16.51	18.41	19.68	20.22
Random AW + RTX + Exclude	15.02	16.50	18.62	21.10	22.72	23.30

100. Table 4.1 shows the packet loss rate with different settings of ActiveNDN while varying the network size from 50 to 500 nodes. The baseline approach obtains the packet loss up to 76.88%, which is not sufficient for ActiveNDN to compute an accurate result. The approaches with three proposed wireless mechanisms have lower packet loss because the packet collision is significantly reduced. The random aggregation window (Random AW) significantly reduces the packet loss, down to 35.86%. An approach with Interest retransmission (RTX) significantly increases the overall number of packet transmission which also more successful as the transmissions are spread out in time leading to less transmission. When including exclude selector (Exclude) in the Interest packet, the size of the Interest packet increases. This makes the transmission to take longer time, which slightly increase the chance of packet collision.

From the validation result discussed in Section 4.4.2, the Baseline configuration without packet collision can achieve 0% incorrect answer in 50, 100 and 200 nodes network size. In Figure 4.10, the dark blue histogram shows that once the collision was introduced in the same simulation setting, due to packet loss, more than 95% of incorrect answers is counted even for networks of size less 200 nodes. This is a very significant decline as compared to the validation results. Nevertheless, after applying the three mechanisms for wireless communication, the incorrect answers can be reduced down to 0%.

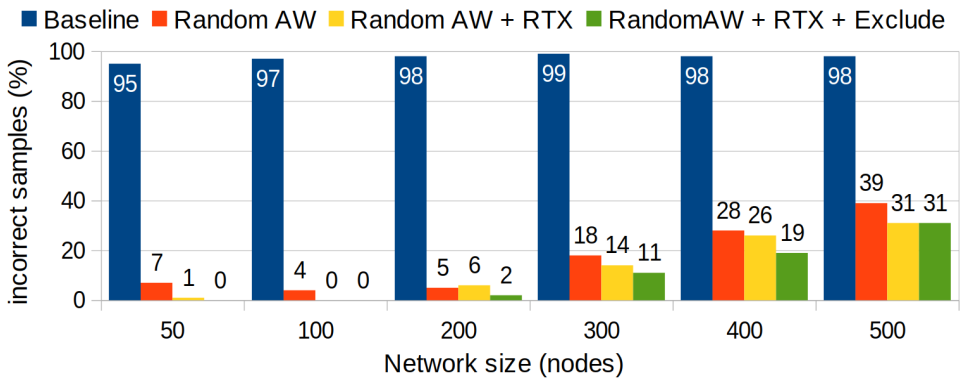


Figure 4.10 – Percentage of samples that give incorrect answers

These incorrect results are caused by: (1) the function scope is too small, so there

Table 4.2 – Function coverage and Data inclusion

Network size (nodes)		**50**	**100**	**200**	**300**	**400**	**500**
Baseline	Func. coverage (%)	98.56	99.08	99.54	98.55	95.02	88.55
	Data inclusion (%)	5.70	3.76	2.54	1.38	1.17	0.79
Random AW	Func. coverage (%)	98.58	99.10	99.59	98.75	96.12	90.88
	Data inclusion (%)	97.56	95.23	91.94	83.38	74.49	66.89
Random AW. + RTX	Func. coverage (%)	100.	100.	99.94	99.00	96.37	91.20
	Data inclusion (%)	99.78	99.46	96.57	86.84	78.21	69.83
Random AW. + RTX + Exclude	Func. coverage (%)	100.	100.	99.93	99.02	96.35	91.14
	Data inclusion (%)	100.	99.94	97.67	89.82	81.25	72.93

are nodes that are unreachable within the specified time interval, as mentioned earlier, and/or (2) packet loss due to packet collisions in the wireless networks. It is evident from the differences between the set of nodes covered by the search, or *Function Coverage* set, and the set of nodes contributing to the computation, or *Data Inclusion* set, as shown in Table 4.2. These two sets of nodes were exactly the same when there was no packet loss during the validation runs. To improve the correctness of computation results, three mechanisms for wireless sensor network are applied in this simulation for further analysis as following.

Optimizing the Random Aggregation Window

Adding a random reduction to adjust the aggregation window size (Random AW) reduces the number of packet losses due to packet collisions, and thus reduces the probability of having incorrect responses. To specify an aggregation window size TTL_i for a function call, a random reduction rate between 30 and 70 percent is applied to the incoming aggregation window size TTL_{i-1} as follows:

$$TTL_i = TTL_{i-1} * random(0.3, 0.7) \tag{4.4}$$

Whenever TTL_i is less than 10 ms, the node will not transfer the Interest. For recursive calls, at this point the recursion is terminated, and the values are returned to the parent callers. Figure 4.10 shows that applying this mechanism leads to an improvement in result accuracy of more than 90% compared to the baseline. This also indicates that Wi-Fi's CSMA/CA alone is not sufficient for collision avoidance when many ActiveNDN nodes are transmitting simultaneously.

Impacts of Interest Retransmission

Interest retransmission (RTX) is introduced, aiming at alleviating the effect of packet loss. The re-transmission timer is set to start re-transmitting the Interest after 70% of

the outgoing aggregation window TTL_i has lapsed to ensure that all callees have enough time to respond. The re-transmitted Interest's lifetime, or aggregation window, TTL_{i+1} is nested inside the outgoing TTL_i. Re-transmitting Interests with a smaller aggregation window can be successively repeated until the aggregation window is less than 10 ms.

Enabling the RTX (yellow histogram) can increase the number of correct results until the network size reaches 200 nodes, as shown in Figure 4.10. From this investigation, the RTX traffic causes packet collision, especially at the edge of the coverage when the aggregation window is too small.

From our experiments, network coverage can be separated into two clusters: one is closer to the anchor node and the second is a cluster located at the edge of function coverage. Interest re-transmission gives more benefit for the cluster closed to the anchor, where the aggregation window is large, the window size becomes more diverse between node and more time for packet to be transmitted. However, at the edge cluster, the aggregation window is small due to the multi-hop reduction, the re-transmission duration is also small. By having multiple nodes' retransmission within a small period, the channel will become more congested and cause more packet collision. For the larger network, the choosing a suitable size of function scope TTL_0 is crucial as nodes located at the edge cluster can be reached and have enough time to return the Data. From this study, RTX traffic causes packet collisions, especially at the edge of the coverage or termination of the recursion, when the aggregation window is getting too small.

Applying Interest Exclude Selector

Inclusion of an *Exclude selector* in the Interest can suppress unnecessary Data packet transmission due to the Interest re-transmission, thus, reducing network collision, which then improves correctness of network computation. Figure 4.11 shows the number of network Data packet transmissions. The re-transmission with an Exclude selector have a much lower number of Data packet transmissions and thus a lower chance of packet collision. However, when the coverage becomes smaller compared to the network size, e.g. 500 nodes, this mechanism has shown limited benefit as it is overwhelmed by an out-of-coverage problem due to inappropriate choice of aggregation window size.

These three mechanisms can significantly help to reduce the differences between the Function Coverage and Data Inclusion sets, as shown in Table 4.2. With function scope TTL_0 of 10 seconds, it can be said that *FindMax* with random aggregation window + RTX + exclusion provides correct answers for a network with up to 200 nodes.

The simulation results show that the packet loss in wireless network effects the ActiveNDN computation accuracy, and the proposed mechanisms: randomized aggregation window, Interest re-transmission and Interest Exclude Selector, can deal with the packet loss and reduce the inaccuracy from the packet lost. The accuracy related to the coverage of the function call, which is determined by the size of the scope TTL_0, is discussed in the next section.

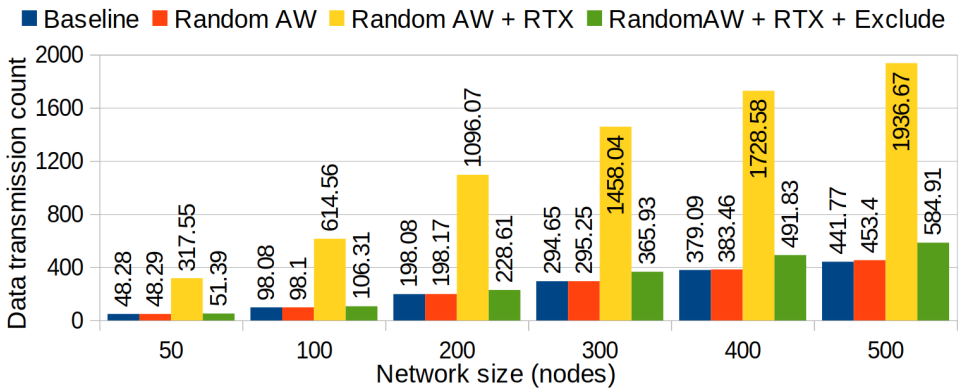


Figure 4.11 – Number of Data packet transmission

4.5 Function Scope Analysis

With distributed computation on wireless networks, it is expected that the computation finds different results at different times due to changes in the sensor readings, while different sizes of the initial function scope (TTL_0) can play an important role in the data inclusiveness of IoT nodes in the computation. When the data inclusiveness is not completely covered to all node, it means that some data has gone missing, and the computation result can become inaccurate. This can be caused by having too small TTL_0, or function scope that won't allow the Interest to be forwarded to all nodes in the network or by some physical conditions of the network that causes some nodes to be isolated, such as wrong configurations or hardware malfunctions. The network is disconnected where parts of the network cannot be reached for a given transmission range (T) causing the correctness of results to downgrade where it is left to the IoT application to decide on what would be an acceptable error rate for them. Some application may require data from all nodes in the network be included in the computation, and some other may set satisfaction rates much lower; the higher the inclusion of nodes (C), the higher the accuracy of the computation.

The simulation result in the previous section, Figure 4.9a, shows that the correction of the computation results are reduced in large network sizes because there are nodes that cannot be reached; hence they are unable to contribute to the computation. As demonstrated in section 4.4.2, if the network is fully connected regarding a given T, then C could be as high as 100% in linear time, $O(N)$ as our traversal algorithm is based on breadth-first-search. To cover the entire network, TTL_0 should be chosen to relate to the size of the network (N) as well as the time taken for function calls, which is where the topology of the network comes into play. The time taken for function calls is related to the maximum number of hops (H), which can be approximated by the diameter of the network.

However, if the network is sparse with small T, it is likely that parts of the network cannot be reached hence C can downgrade while higher number of hops may occur; with large T, we can achieve higher C regardless of the choice of TTL_0. If we increase TTL_0 for a sparse network with small T, the performance may improve as more time can be spent visiting more nodes. But for high T, whether the network, is sparse or dense, the scenario would be become closer to a single hop network in which case the choice of TTL_0 would depend largely on the number of nodes more than the number of hops. For dense network, more time is needed to visit all nodes within a given range T as the network is dense and more so for low T. Here, the choice of TTL_0 is important. If the TTL_0 was too small, many reachable nodes will not reach and the data can be missed out, thus reducing the inclusion rate C.

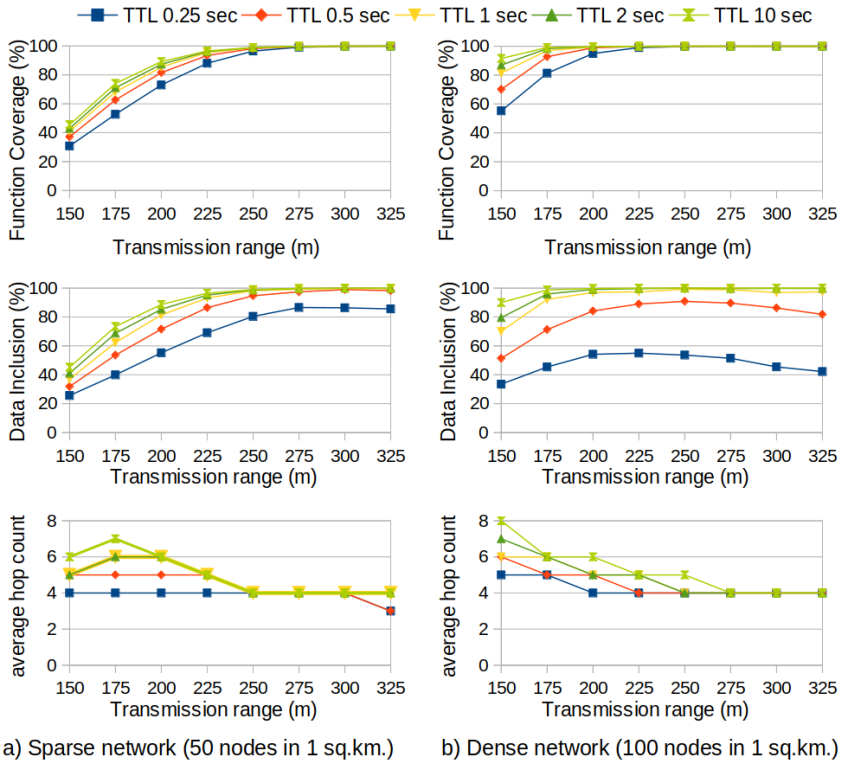


Figure 4.12 – Comparing TTL in sparse and dense network

We simulated the ActiveNDN to compare different TTL_0 with varying transmission range T in sparse and dense network. Our simulation results shown in Figure 4.12 has confirmed with above analysis. For sparse network, with large T, small H is expected while for small T, to achieve large C, higher H and higher TTL_0 is expected. For dense network, if TTL_0 is set too small, C drastically downgrade regardless of choices of the transmission range T. On both cases, longer T is preferred for both cases while TTL_0

should be set to cover all nodes N and H, or, $TTL_0 \geq a * N + b * H$ where a is the unit cost in visiting each node and b is the unit cost for a function call.

For our target applications, the users can decide to accept the level of confidence C of their applications and configure their T and TTL_0 accordingly.

4.6 Conclusion

In this chapter, the ActiveNDN architecture is demonstrated through the proof-of-concept implementation and large-scale simulation. It was demonstrated that with an IoT application on a wireless network, the occurrence of packet collisions is the most important issue affecting system performance because the Interests of the Data can be lost, giving incorrect answers as the result of the in-network computation. To recover from Interest packet loss, Interests are retransmitted so missing Interests can be serviced properly. To avoid packet collision, a randomized aggregation window is deployed and to reduce an unnecessary transmission, exclusion fields are adopted in the Interest packets. These three mechanisms have drastically reduced the percentages of incorrect answers. However, to ensure that the system can correctly answer at all times, it is important to choose appropriate aggregation windows, or Interest lifetime TTL, for different network sizes and densities.

We have shown that ActiveNDN is capable of performing robust distributed computations on sensor data near the sensors themselves by offloading the computations to a function library attached to the NDN forwarders through the proof-of-concept prototype, which will be used in further complex scenario. Using simulation experiments, we have shown that the efficiency of the computations is greatly improved by distributing them across the local wireless sensor network, and that timely responses are possible for IoT applications.

5 ActiveNDN for IoT Data Processing

In-network computation enables distributed computation across the network, whereas traditional IoT systems rely on a centralized cloud to perform various tasks such as storing, processing, and computing all data instances. If the IoT network is isolated and the connection to the Internet is sporadic, cloud-based computation may not be able to provide prompt responses. From this perspective, in this thesis, we propose to perform data analysis within the network instead of relying on the Internet and the cloud. Based on this, an in-network computation architecture using ActiveNDN, described in Chapter 3, has been proposed for wireless sensor IoT networks.

In this chapter, we first demonstrate an implementation of ActiveNDN with a real testbed (Section 5.1) for real-time PM2.5 prediction in an IoT network. Our approach focuses on data processing within the network by using IoT devices or sensor nodes that already exist in the networked system and are already used to forward traffic to perform computations and communicate with other nodes to aggregate the computation results.

In-network computation does not require a dedicated server for data processing, as computations can be performed on any sensor node. The node not only forwards the data, but also processes and aggregates it before sending it. This can not only distribute the computational load among the nodes, but also reduce the network traffic. To further evaluate our proposed ActiveNDN, this chapter conducts several experiments comparing the performance of in-network computation in ActiveNDN with the centralized approach where the computations can be performed by one of the nodes acting as an edge server within the IoT network (Section 5.2). Our comparisons were performed through simulation and real testbed experiments using in-network ActiveNDN against two centralized edge computing approaches using NDN and an IP-based CoAP IoT protocol. For the simulation experiments, we extended our ActiveNDN simulation from Section 4.5 to compare the system with different network sizes. Then, we compared the three schemes with a real PM2.5 prediction scenario in a real IoT network in a remote community with an area of 0.3 square kilometres.

5.1 Real-world Application

5.1.1 ActiveNDN Testbed

To provide proof of concept, a real-world IoT network was set up to monitor air quality using a sensor kit developed as part of the SEA-HAZEMON [SEA-HAZEMON,] project. Figure 5.1 shows an example of an IoT node named Canarin connected to multiple air quality sensors such as PM1/2.5/10, temperature, humidity, CO, CO2, and GPS, using UDOO NEO [UDOO,], a single board computer, as the main controller and computational unit. In this testbed, we use the ActiveNDN codebase of the prototype developed in Chapter 4 and placed the four Canarin nodes in Thai Samakkee village (see Figure 5.2) in Mae Sot district, Tak province, Thailand, using the connectivity of the existing TakNet network [Kanchanasut et al., 2018], a flat OLSR [Jacquet et al., 2001] network with over 400 nodes.

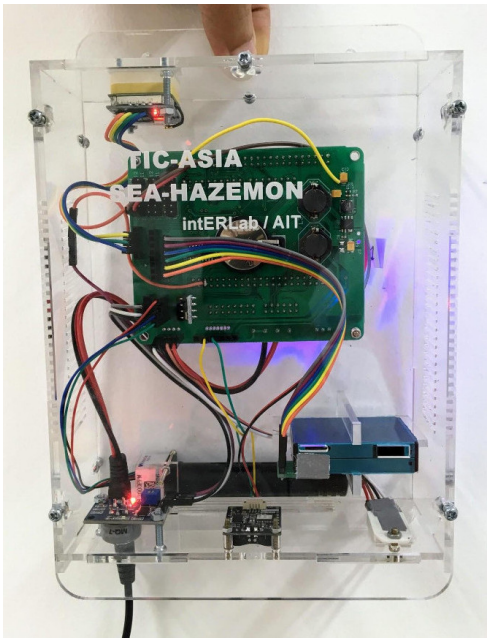


Figure 5.1 – A Canarin sensor node

To prepare the testbed for ActiveNDN and NDN, the Named Data Networking Forwarding Daemon (NFD) [NDN, b] responsible for managing name-based routing over the network is installed on all Canarin nodes creating an NDN overlay over the existing OLSR routing protocol [Jacquet et al., 2001] in the TakNet network. Then, we use a static routing to set up the forwarding table (FIB) of each node to assign the name prefixes and NDN faces.

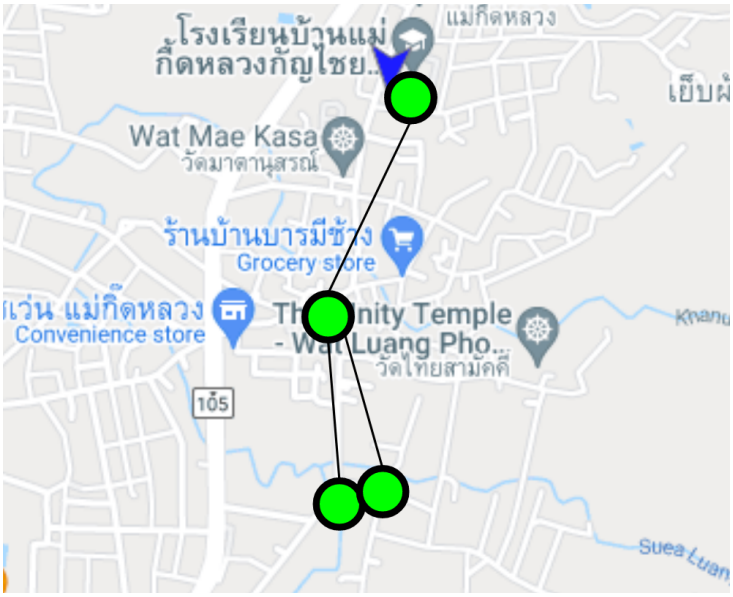


Figure 5.2 – ActiveNDN testbed in Community Wireless Mesh Network

5.1.2 *Prediction of PM2.5*

ActiveNDN will be used to monitor air quality in communities isolated from the Internet to provide real-time alerts to local members. On the ActiveNDN testbed, we applied plume movement modelling proposed in [Kanabkaew et al., 2019] to predict PM2.5 concentration in the local village. Here, we demonstrate an air quality IoT application where PM2.5 prediction is calculated from multiple function calls, including *FindAv*, *Sigma*, which sums the data, and *Predict*, to predict the PM2.5 concentration for the next hour. *FindAv* is used on each node in the network to calculate the local PM2.5 averages or aggregate the averages of the entire network. Consequently, the *Sigma* function is deployed on a node that continuously aggregates the last 24 hours of average values obtained by *FindAv*. Note that all three functions are run as background tasks to collect data for a linear regression model. Finally, the *Predict* function is triggered by a user to compute the prediction result. Whenever it is called by the Interest request, the *Predict* function takes the sums from the *Sigma* function to build a linear regression model, and then returns the PM2.5 prediction result embedded in the Data packet.

ActiveNDN provides a suitable solution for performing regression analysis and generating short-term predictions of PM2.5 levels for on-site users. As shown in Figure 5.3, an ActiveNDN node can be set to periodically provide PM2.5 predictions to the network. Every 15 minutes, an average PM2.5 value is calculated from the sensor nodes in the network. The average is input into a linear regression program to obtain a predictive model and predict future PM2.5 values for the time period of interest. The entire process involves time-based distributed computing functions running on possibly different nodes

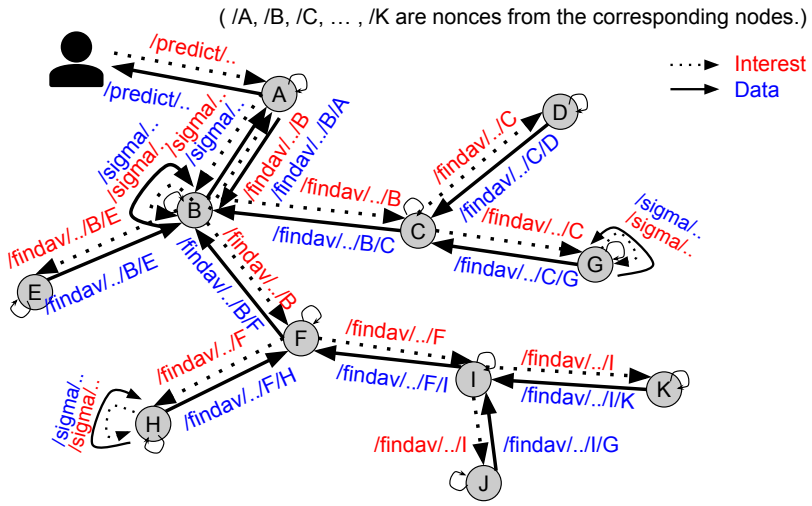


Figure 5.3 – Real-time PM2.5 Data Analytics

in the network. Some nodes may take time-based averages from sensors in the network, while other nodes may perform linear regression on the time series of average data. ActiveNDN. The plume prediction model includes three main steps that can be implemented as function calls in ActiveNDN, as described below.

Finding average PM2.5: This involves two functions,

1. Function *FindAv_ i*: This function is responsible for determining the average PM2.5 values over a specified time period in node i. The time period can be adjusted via a programmable configuration. According to the model proposed in [Kanabkaew et al., 2019], a period of 15 minutes is preferred. The *FindAv_ i* is placed in each node to continuously calculate the local PM2.5 readings. The average value of PM2.5 readings is generated as a Data packet every 15 minutes and stored in the content store (CS) of each node.

2. Function *FindAv*: This function is used for aggregating the averages from all published averages in the network that come from the *FindAv_ i*. As mentioned in Section 4.3.2, ActiveNDN allows recursive function call to be propagated throughout the network. The *FindAv* function follows this principle to recursively aggregate values from other nodes in the network. An example of an Interest message can be expressed as follows: */findav/pm2.5/(caller-nonce) lifetime=*TTL_i

The name prefix consists of the function name (*findav*) and the requested value (*pm2.5*), while the caller nonce is used to identify the caller, as explained in Section 3.5.1. This Interest is expressed after the internal calculation of the average value is completed (*FindAv_ i*). The Interest will be broadcasted to the neighbour nodes. The function starts an aggregation window timer for TTL_{i+1} and waits for the returning Data

until the timer ends. Then it calculates the average and the count of all collected data and returns a result as Data: */findav/pm2.5/(caller-nonce)/(nonce) content=(average PM2.5),(data count)* back to the caller.

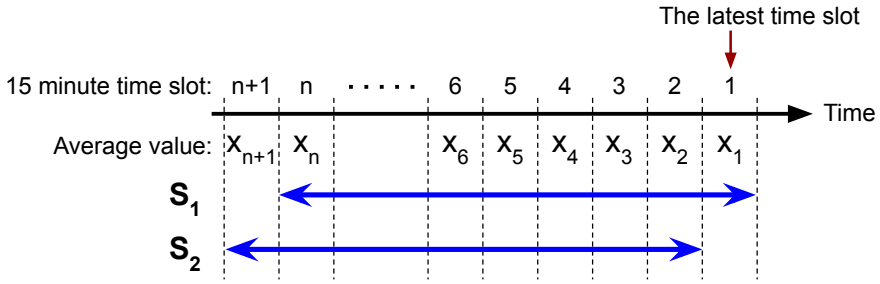


Figure 5.4 – Duration of summations

Summing of average values function (*Sigma*): This function continuously computes sums of x_t, x_{t-1} , $x_t \times x_{t-1}$, x_t^2, and x_{t-1}^2, where x_t is the current 15-minute average PM2.5 measured by all sensors in the network and x_{t-1} is the previous data point measured 15 minutes earlier. A summation is performed over 96 data points representing the last 24 hours of data at 15-minute intervals, as shown in Equations 5.1, 5.2, 5.3, 5.4 and 5.5, where x_1 is the most recent 15-minute average PM2.5 value, x_n is the 15-minute average PM2.5 value from 24 hours ago, and n is the number of training data points collected in 24-hour (96). The duration of summation is shown in Figure 5.4, where S_1 summation includes data from x_1 to x_n and S_2 includes data from x_2 to x_{n+1}.

$$S_1 = \sum_{t=1}^{n} x_t \tag{5.1}$$

$$S_2 = \sum_{t=2}^{n+1} x_t \tag{5.2}$$

$$S_3 = \sum_{t=1}^{n} (x_t x_{t+1}) \tag{5.3}$$

$$S_4 = \sum_{t=1}^{n} x_t^2 \tag{5.4}$$

$$S_5 = \sum_{t=2}^{n+1} x_t^2 \tag{5.5}$$

Notice that the *Sigma* function can be assigned to any ActiveNDN node in the network, but in our case, *Sigma* is contained in the same node as the *Predict* function.

At setup time, a function call is made to initiate the *Sigma* functions that continuously build the sums of the average PM2.5 values needed for the prediction calculations. *Sigma* can be kept active because the corresponding PIT entry has an infinite lifetime, set in the Interest of the first function call. Every 15 minutes, *Sigma* calls *FindAv* with an Interest: */findav/pm2.5/(caller-nonce) lifetime=15m* to recursively collect the average PM2.5 from the network and send the results Data: */findav/pm2.5/(caller-nonce)/(callee-nonce) content=(average PM2.5) freshness=15m* to be cached at CS. When an Interest: */predict/pm2.5/villageA/2021.01.02-01:00/60 lifetime=10s* is received from the first ActiveNDN node found with the *Predict* function, which means a 60-minute PM2.5 prediction is requested. The *Predict* function in its FL is called to initiate a chain of executions and return the final result Data: */predict/pm2.5/villageA/2021.01.02-01:00/60 content=(predicted pm2.5 value after 60 minutes)* to the user, as shown in Figure 5.3. The *Predict* function sends an Interest: */sigma/pm2.5/villageA/t* to the network with a time of day t synchronized with the 15-minute interval of the clock. The *Sigma* function calculates the summations and produces a Data: */sigma/pm2.5/villageA/t content=(n, S_1, S_1, S_3, S_4, S_5)* for each 15-minute clock interval and stores these data packets in the CS of its ActiveNDN node. These data packets can be retrieved from an Interest with a matching name prefix and sent to the calling function to be applied to the linear least squares regression in *Predict*.

Predicting function (*Predict*): The function applies the linear regression method in predicting PM2.5 levels based on past training data [Seltman, 2018]. A closed form for the prediction of PM2.5 value (y) is expressed by the following equation:

$$y = \alpha + \beta x_{t_c} \tag{5.6}$$

where α and β are coefficients calculated from past training data, which can be computed as follows.

$$\beta = \frac{nS_3 - S_1S_2}{nS_5 - (S_2)^2} \tag{5.7}$$

$$\alpha = \frac{S_1 - \beta S_2}{n} \tag{5.8}$$

Note that, all input variables in the β- and α- equations such as S_1 and S_2 are the sums of the average PM2.5 values in the past observation period (24 hours), which can be retrieved from the *Sigma* function. Using this model equation, the 15-minute PM2.5 prediction y can be calculated by inserting the last average PM2.5 measurement x into the model. A longer prediction period can be calculated by applying the previous prediction to the same model. For example, a 30-minute forecast can be calculated by applying the 15-minute forecast to the model.

To deploy ActiveNDN on the real test bed in Thai Samakkee Village, Thailand, we first simulated our deployment plan in our laboratory. We created 10 ActiveNDN nodes

with PM2.5 sensors and placed them in an area the size of a village. In four nodes, we deployed the functions *Predict* and *Sigma*, while the functions *FindAv_i* and *FindAv* were placed in all node. A user's request is sent to the first ActiveNDN node found using the *Predict* function. This then initiates a chain of executions and returns the final results to the user. The *Sigma* can be assigned to any node in the ActiveNDN network, but for our simulation it is contained in the same node as the *Predict* function. We simulated PM2.5 prediction with ndnSIM using raw data from 2018 [Kanabkaew et al., 2019] with satisfactory accuracy. The *Predict* function can produce a one-hour prediction as 195.287 vs. 195.245 ($\mu g/m^3$) based on historical data for the last 24 hours with an RMSE (4.533). In the real deployment of ActiveNDN in Thai Samakkee Village, Thailand (Section 5.1.1), with the same settings as in the simulation, the one-hour predictions could achieve satisfactory prediction accuracy with an average RMSE (6.27).

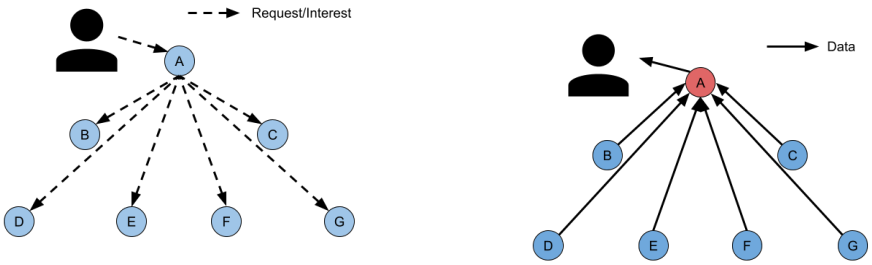
5.2 Edge vs In-Network Computation

To provide prompt responses to user requests, an IoT application must retrieve data from many devices, process the data, and perform the desired computations. With edge computing, all of these activities are centralized and performed by an edge server, which in the case of an isolated IoT implementation might be located within the same local network. With in-network computing, the IoT application can rely on computations performed by members of the network.

In this section, we perform simulation experiments to compare the performance of ActiveNDN with other centralized computing models in large wireless sensor networks. ActiveNDN is inherently designed for an in-network computational model, where computations can be performed anywhere in the network in a distributed manner. To evaluate the performance of ActiveNDN, we selected two communication protocols, including the Constrained Application Protocol (CoAP) [Shelby et al., 2014] and a native Named Data Networking (NDN). The CoAP is an IP-based protocol that is widely used in many IoT systems, while the computational model is best suited for the centralized approach. As for the NDN, in this evaluation, we only use the native approach of the NDN, where a single user sends a request to retrieve data from all members and perform all computations at the anchor or edge node, so it is classified as a centralized computation model. For this evaluation, the scenario of a wireless sensor network is used, considering single-hop and multi-hop communication topologies.

In the single-hop topology, all nodes in the network are within radio range, so they can communicate directly with each other without the need for forwarding. As shown in Figure 5.5, the central or anchor node, acting as an edge server, can send the Interest or request directly to all sensor nodes in a broadcast or multicast transmission (see 5.5a), and the sensor nodes can send the data directly back to the central node (5.5b).

In a multi-hop topology, each pair of nodes may be out of radio range of the other and



(a) Interest traversal (b) Data aggregation

Figure 5.5 – Single-hop network

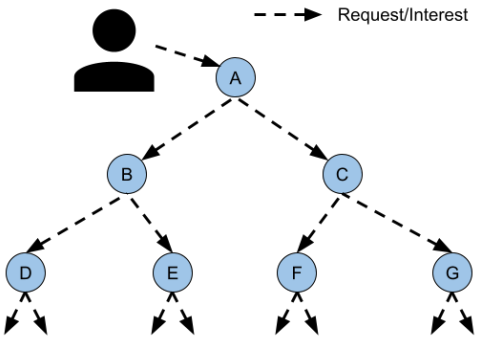
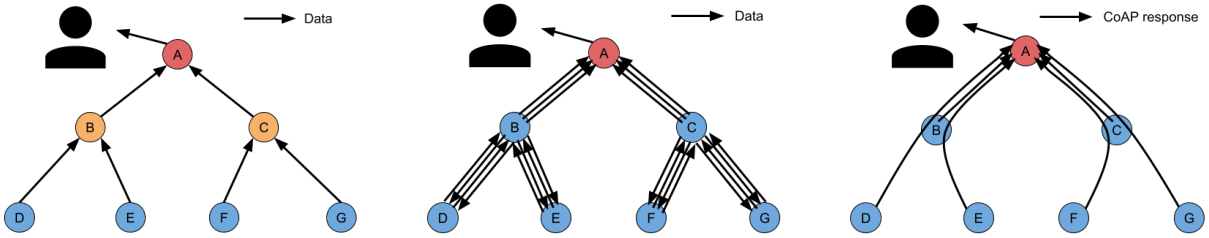


Figure 5.6 – Interest traversal in multi-hop networks

inter-node communication must be relayed by intermediate nodes. The request/Interest traversal in this network is shown in Figure 5.6. The edge server or the anchor node A and nodes D, E, F, and G cannot communicate with each other without nodes B and C relaying the messages. In this case, node A's request is forwarded by nodes B and C. For data aggregation phase in the multi-hop topology, the three schemes have different network traffic characteristics. In ActiveNDN, the Data aggregation is performed by in-network computations, as shown in Figure 5.7a. Relaying nodes B and C receive and aggregate the Data from nodes D and E, and F and G, respectively. Thus, nodes B and C can each send only one aggregated data result back to the anchor node. In CoAP, on the other hand, nodes B and C do not process the responses from nodes D, E, F, and G and only forward them separately to the central node, as shown in Figure 5.7c. In the NDN case, shown in Figure 5.7b, the network traffic is significantly higher. This is due to the forwarding problems of native NDN in wireless networks, as mentioned in Section 2.3.5, due to overhearing and lack of network information.

In both NDN and ActiveNDN, a packet is broadcasted while the neighbouring nodes' information is not available. In order for the packet to reach the entire network, the nodes in the network must forward the packet. Unlike ActiveNDN, native NDN forwarding



(a) ActiveNDN (b) NDN (c) CoAP

Figure 5.7 – Data aggregation in multi-hop networks

does not have a nonce checking mechanism to exclude irrelevant packets from forwarding. Therefore, in the NDN scheme, any node can listen for a packet from its neighbours and forward the packet as well, resulting in high network traffic. This can be seen in Figure 5.7b. For example, node D that listens to any data sent by node B, such as data originating from node B or node E, will unnecessarily forward this data to its neighbours.

The performance evaluation of ActiveNDN, NDN and CoAP is performed in both single-hop and multi-hop topologies, varying the network size with 50, 60, 70, 80, 90 and 100 nodes, respectively. The nodes are placed uniformly with a density of 100 nodes/km^2. Each node is connected to a Wi-Fi interface configured in ad hoc mode with a fixed data rate of 1 Mbps. To simulate the single-hop topology, we set the transmission range of each node to 2 km to enable a one-hop connection between all nodes.

For the multi-hop topology, the transmission range is set to 250 m. The protocol configuration of the three schemes remains the same in single-hop and multi-hop networks. In each scheme, we measure and compare the average data inclusion from repeated 100 simulation runs. To perform the simulation experiments, we use ndnSIM [Mastorakis et al., 2017], an official simulation platform for NDN.

The simulation results are shown in Figure 5.8. In the single-hop topology, the network is connected by one hop, while in the multi-hop topology, the average number of network hops ranges from 4.1 hops for the 50-node network to 5.6 hops for the 100-node network. In both the single-hop and multi-hop cases, ActiveNDN outperforms the other systems with 99-100% data inclusion within the same time span, while NDN and CoAP only reach up to 47.9% and 51.1% for single-hop and 47.8% and 22.0% for multi-hop, respectively. This is because ActiveNDN applies data aggregation to reduce the number of packets to be transmitted and employs various collision avoidance and packet loss recovery strategies.

In the single-hop case, the CoAP scheme performs better than the NDN scheme, and as the size of the network increases, the difference in data reception between the two increases. This is due to wireless forwarding in NDN, where Data packets are unnecessarily transmitted by each node to forward the packets, causing the amount of traffic to increase polynomially according to the number of nodes, and thus packet collisions increase tremendously. CoAP, on the other hand, uses unicast forwarding, which does not cause as

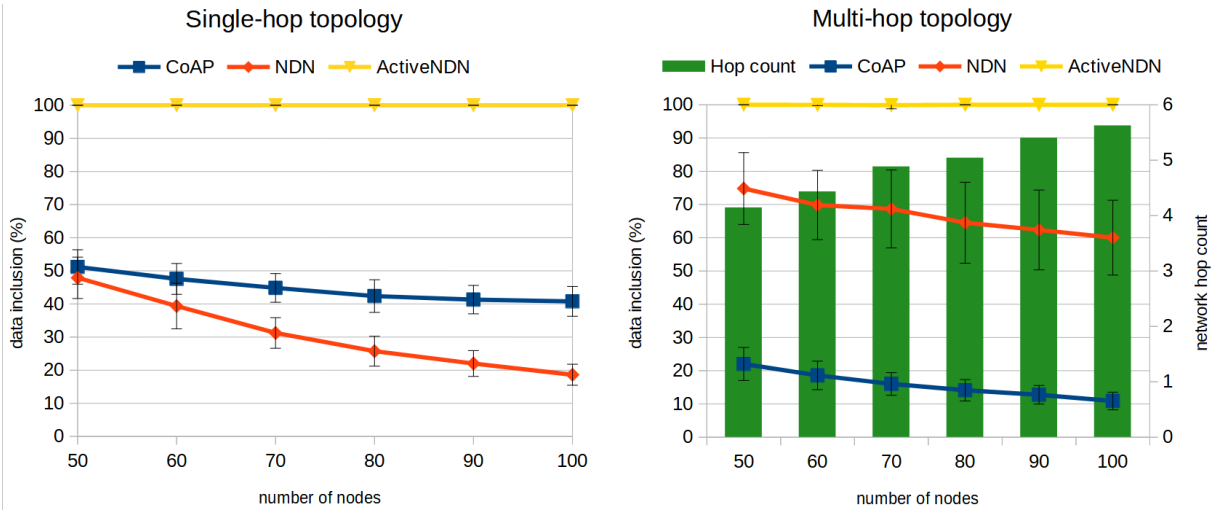


Figure 5.8 – ActiveNDN vs. Centralized computing

much additional traffic as NDN, so it can provide better scalability than NDN. However, both systems are heavily affected by the collision problem.

On the other hand, NDN achieves higher data inclusion than the CoAP scheme in a multi-hop topology. Due to the redundancy of relay transmissions in NDN, each node can provide a spare copy of the lost packets, and the number of nodes in a collision domain, i.e., in one hop, is smaller and does not scale with the network size as in the single-hop scenario. CoAP is not able to handle packet loss on the intermediate relay nodes, and end-to-end retransmission is not sufficient to resolve packet collisions in a timely manner.

The simulation confirms that ActiveNDN achieves better performance and scalability than the centralized computations schemes for data aggregation in wireless sensor networks. More data can be successfully collected by reducing traffic and using collision avoidance strategies in the wireless network. While the CoAP scheme performs better than NDN in single-hop networks, the NDN system turns out to be more reliable in multi-hop experiments. In the next section, we will further test the three methods in a real test environment to compare their performance in the real world.

5.2.1 Comparisons Using Testbed Experiments

We used the testbed described in 5.1.1 to compare our in-network ActiveNDN with centralized edge computing in NDN and CoAP.

For the NDN scheme, a dedicated edge node is implemented with the PM2.5 prediction model as a Python program with NFD forwarder via the PyNDN library [NDN, b]. The edge node periodically broadcasts an Interest message to collect PM2.5 data from all sensor nodes deployed in the network. The collected data is stored in the program memory (RAM) of the edge node. Once the PM2.5 prediction is requested, the edge node pulls

all the collected data from the last 24 hours from the program memory and executes the chain of functions that includes *FindAv_i*, *FindAv*, *Sigma*, and *Predict*. The computed result, i.e. the prediction of the average PM2.5 concentration in the next hour, will be returned to the requested user through a Data message.

In the CoAP scheme, data collection is done on a push basis. Each node sends its raw PM2.5 data to the CoAP server in the edge node. The Repo program in each node is implemented as a CoAP client, which sends the PM2.5 measurements directly to the CoAP server in the edge node over the normal IP network. As with the NDN setup, the *compute program*, which contains both storage and FL, is implemented as another CoAP client that uses CoAP's Observe method to subscribe to the local CoAP server for the PM2.5 measurement. The server then forwards the PM2.5 measurement to the compute program, which keeps the data in program memory ready for processing by the prediction function. The prediction program in the CoAP scheme works in the same way as in the NDN scheme, where all processes - averaging, summation, linear regression, and prediction are contained in one program and are started as soon as it is requested.

We compare the accuracy of prediction results, response time, computational resource usage, and network traffic between the three schemes: ActiveNDN, NDN, and CoAP. For each experiment, we let the system collect data for 6 hours before requesting the PM2.5 prediction.

Prediction Accuracy

We compare the accuracy of the prediction error of each scheme with different prediction durations: 15, 30, 45, and 60 minutes. The predictions are requested every 15 minutes for 6 hours, giving a total of 24 predictions per case. We take the actual measurements taken from the same sensors collected on the Hazemon server, average them at the 15-minute interval, and use them as a reference to calculate the error of the predictions.

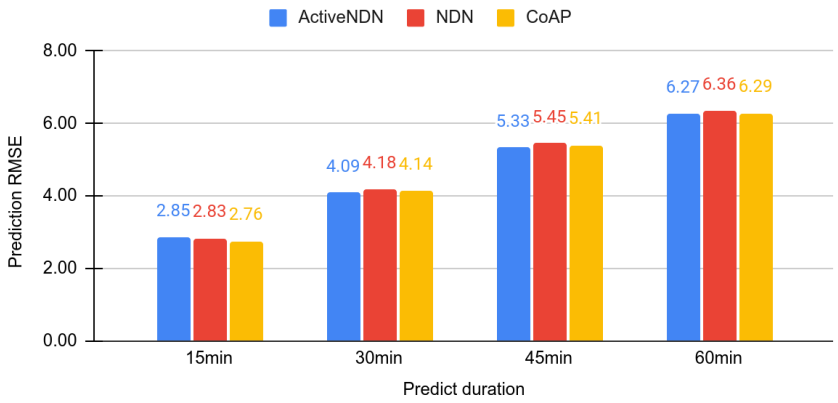


Figure 5.9 – Error of the prediction results

The RMSE of the prediction results is shown in Figure 5.9. Overall, the longer the prediction period, the larger the error. Comparing the three schemes, ActiveNDN has the lowest prediction error, while CoAP is a close behind and NDN has the worst result, but the errors are not significantly different.

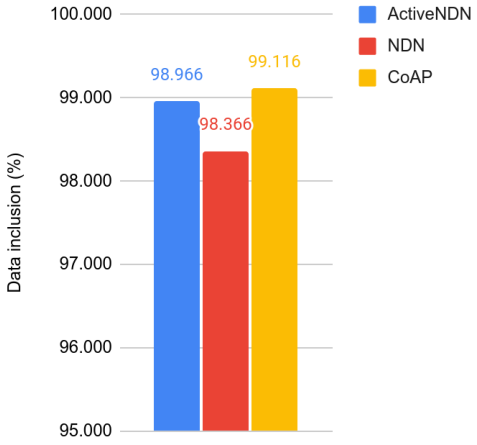


Figure 5.10 – Data inclusion of the prediction

We further investigated the inclusion of the data and found that, as expected, the results are not very different at less than 1%. As can be seen in Figure 5.10, ActiveNDN and CoAP have comparable data inclusion, while NDN has the lowest data inclusion, which could be the reason for the lowest accuracy.

Response Time

Response time is measured by how much time a function takes to work and provide a response after being started by a call. We measured the response time of a process to produce 60 minute PM2.5 predictions. We repeated the process 10 times and averaged the response time.

The result in Figure 5.11 shows that ActiveNDN was able to provide a response within 0.232 seconds, which was faster than the other schemes, CoAP was second with 1.749 seconds and NDN took 1.604 seconds.

We further examine the time taken in each part of the prediction program, as shown in Table 5.1. We found that averaging in CoAP and NDN takes the most time. It takes more than 1.5 seconds to process 6 hours of raw data from 4 nodes, which is expected given the processing power of an SBC processor. In ActiveNDN, the average and *Sigma* are continuously calculated, and the result is output, which is cached in the background at CS. It takes a total of 0.166 seconds to retrieve the Average and *Sigma* results from NDN's CS. Of this time, 0.074 seconds is spent waiting for the response from CS and 0.092 seconds is spent creating the Interest packets and parsing the results from Data

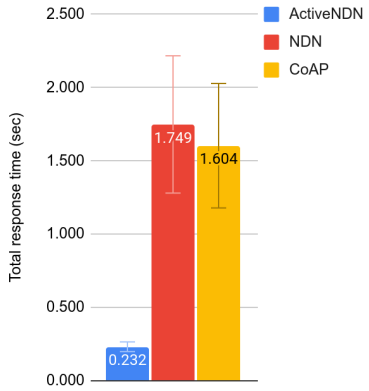


Figure 5.11 – Average response time comparison

Table 5.1 – Average time taken by parts of the predict program

Part of prediction program	ActiveNDN	NDN	CoAP
Predict	0.012	0.018	0.018
Sigma	-	0.010	0.009
Average	-	1.636	1.573
NDN CS response time	0.074	-	-
Packet creation/parsing for retrieving data from CS	0.092	-	-
Packet creation/parsing for incoming call and returning result	0.054	0.085	0.005
Total response time	**0.232**	**1.749**	**1.604**
SD of total response time	0.033	0.468	0.425

packets. The calculation time taken in the *Sigma* part (for NDN and CoAP schemes) and in the *Predict* part (for all schemes) is comparable between the schemes, as the difference is only 6 ms. Also, when comparing the NDN and CoAP protocols, the ActiveNDN and NDN schemes take more time to parse and create packets than CoAP. However, this is due to how the different programming libraries are implemented for the protocol.

Resource Consumption

We measure CPU and memory usage of ActiveNDN, NDN, CoAP for 2 hours while requesting new prediction results every 15 minutes. We focus only on the processes related to the prediction program, categorized by the system components: Repo, FL /Compute program, NDN Forwarder and CoAP server.

The CPU usage of the process in the context of prediction is shown in Table 5.2 CoAP consumes the least CPU consumption because it operates directly on the IP layer, which

Table 5.2 – CPU usage comparison

CPU usage (%)	**ActiveNDN**		**NDN**		**CoAP**	
	non-anchor nodes	**anchor node**	**sensor nodes**	**edge server**	**sensor node**	**edge server**
FL/Compute program	8.739	8.800		7.250		1.480
Repo	5.733	5.660	5.271	5.788	0.028	0.034
NDN Forwarder	1.206	1.301	0.963	1.109		
CoAP-server						0.411
Total	**15.678**	**15.761**	**6.234**	**14.147**	**0.028**	**1.925**

is natively supported by the OS and highly optimized, while the ActiveNDN and NDN schemes use an additional NDN protocol layer and therefore have more CPU processing overhead. ActiveNDN also consumes the most CPU, compared to the other schemes because multiple computational processes are constantly running in the background. In NDN, the central node requires a high CPU consumption as in ActiveNDN, but the NDN sensor nodes consume less CPU time since they are not responsible for running the computational processes.

Table 5.3 – Memory usage comparison

Memory usage (%)	**ActiveNDN**		**NDN**		**CoAP**	
	non-anchor nodes	**anchor node**	**sensor nodes**	**edge server**	**sensor node**	**edge server**
FL/Compute program	0.068	0.066		0.033		0.150
Repo	0.013	0.013	0.012	0.013	0.146	0.145
NDN Forwarder	0.024	0.024	0.023	0.024		
CoAP-server						0.056
Total	**0.104**	**0.103**	**0.035**	**0.069**	**0.146**	**0.351**

The memory consumption for each scheme is shown in Table 5.3. The CoAP scheme significantly consumes the most memory, especially for the Repo and the Computing process that constitute the CoAP client. This could be due to the fact that the client part of the CoAP library was not implemented optimally. When comparing ActiveNDN to NDN schemes, NDN requires less memory due to its lower function overhead. This is due to the fact that in NDN there is only one function that performs all operations, while in ActiveNDN there are several functions that are executed simultaneously, especially the *FindAv_i* function that calls itself recursively. ActiveNDN uses the same amount of memory on each node because of the running background functions, and the process is distributed on each node.

Network Traffic

We measured the total network traffic on the data collection of each scheme for 2 hours.

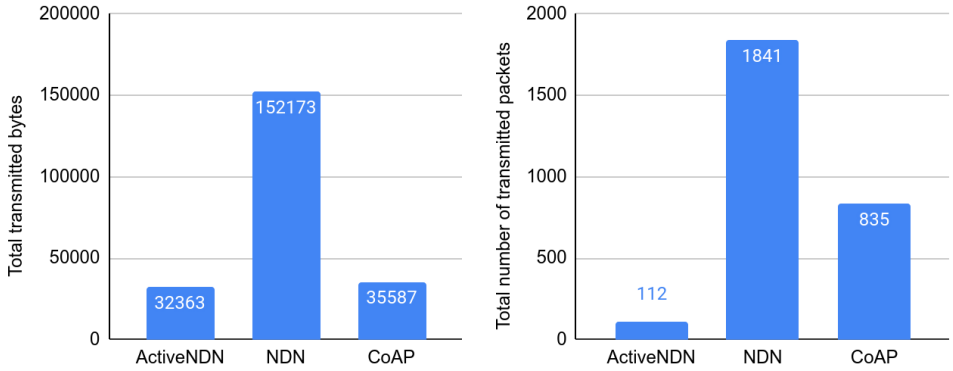


Figure 5.12 – Network traffic comparison: (left) total traffic shown in number of bytes, (right) total transmission in number of packets.

As can be seen in Figure 5.12, NDN has the highest traffic, followed by CoAP and ActiveNDN. This is because all transmission in NDN is flooded to every node in the network, while transmission in ActiveNDN is a one hop broadcast and in CoAP it is an end-to-end unicast with IP routing. Also, the ActiveNDN has the least number of packets because the data is aggregated at each hop, so only the processed data is transmitted.

5.3 Conclusion

In this chapter, we have demonstrated the use of ActiveNDN in a wireless OLSR network with real-world applications that provide sufficiently accurate hourly PM2.5 predictions. Further improvements can be made with in-network calculations to handle possible abrupt changes in the environment in real time, such as changes in wind speed and direction that cannot be effectively handled with post-data analysis. In addition, complex real-time simulations of physical phenomena can be achieved by programming the network and performing computations such as image processing and machine learning on the network.

We also evaluated the performance of in-network computations on ActiveNDN compared to centralized computations through experiments. It was found that ActiveNDN performs better than the centralized approaches in this local IoT network environment for both NDN and CoAP schemes. It provides more accurate results due to higher data inclusion, consumes the least network traffic with the lowest response time, while consuming more computational resources than the centralized schemes. The ability to distribute the computational load across many nodes in the network so that not all data needs to be sent to a central node makes ActiveNDN suitable for large-scale IoT deployments.

6 Conclusions and Future Works

With the explosive growth of the "Internet of Things" (IoT) devices, various services such as smart cities, smart monitoring, and smart transportation have been proposed in the last decade. Although numerous efforts have been made to enable IoT services in remote areas, most solutions are still based on a cloud or centralized architecture. These are prone to be disrupted by intermittent internet connectivity while the central server cannot communicate with remote devices.

In-network computation has been proposed to enable data processing at intermediate nodes in the network, with the network processing the data rather than just forwarding it. However, due to the complexity of the mapping process between data and host or the mismatch between application and network layers, such a design cannot be efficiently implemented in the traditional Internet protocol. The Named Data Networking (NDN) architecture, on the other hand, identifies content by name and applies name-based routing, making the network location-independent. This allows a user to make a name request, called an Interest, to the network, which can be forwarded to the nearest node that has a data object matching the name. This makes NDN an ideal candidate for in-network computation.

6.1 Thesis Summary

In this thesis, the existing IoT developments and in-network computation techniques were investigated to identify the challenges in implementing a novel in-network computation framework for IoT networks. To this end, ActiveNDN, a general distributed computing framework using in-network computing techniques is designed, implemented, and evaluated. ActiveNDN was extended from the Named Data Networking architecture and adds a Function Library (FL) to the NDN forwarder to store the executable codes and execute the function. A function call can be expressed as a prefix of the Interest name that is forwarded across the network to find the node that has a matching name in FL. In turn, the computed result is appended to a Data packet and forwarded back to the requested user. A function call can also be a call to itself or a recursive call; recursions can occur within the same node or be applied in a distributed manner across the network.

In wireless sensor networks, which are often used for IoT applications, ActiveNDN de-

ployments may experience packet loss as many nodes transmit simultaneously, which may cause packet collisions, and overhearing induced by the shared wireless medium, causing redundant responses or transmissions. Three mechanisms, including a randomized aggregation window, retransmission of Interests, and Interest exclude selector are used to avoid packet collisions, compensate for lost transmissions of Interests, and reduce unnecessary transmissions, respectively. A prototype of ActiveNDN has been implemented as a proof of concept, which has been validated in both laboratory and real-world deployments. In addition, simulation experiments were conducted using ndnSIM to benchmark the performance of ActiveNDN on a large scale. The results of the experiments confirm that our proposed wireless network mechanisms can drastically reduce packet loss and results inaccuracy.

The performance of in-network computation on ActiveNDN is evaluated through extensive experiments and compared with other centralized computation approaches (native NDN and CoAP). The results show that ActiveNDN achieves more accurate results due to its higher data inclusion, while consuming less traffic and having faster response time. However, ActiveNDN seems to consume more resources (CPU and memory consumption). Another interesting result is the efficiency of ActiveNDN in distributing computational tasks among many nodes in the network. This proves that ActiveNDN is suitable for large-scale deployments in the IoT network.

For a more complex scenario, ActiveNDN was deployed in a wireless sensor network with real-world applications that provide sufficiently accurate PM2.5 predictions. The PM2.5 prediction application uses linear regression analysis with a set of computational functions. This demonstration confirms that ActiveNDN is capable of performing complex real-world applications through network programming and distributed computation execution.

6.2 Future Works

The emergence of data-intensive applications has significant implications for the current Internet architecture. New digital applications such as Augmented Reality (AR), Mixed Reality (MR), and Virtual Reality (VR) require intensive real-time computations to build a virtual environment. These tasks include collecting and processing multiple sensor data, high-resolution (HD) image rendering, and 3D processing that are too heavy to be performed in terminals (e.g., VR headset, smartphone, digital set-top box) or a dedicated edge node. On the other hand, pushing computation workload to the hi-end data center could handle all the complexities. Uploading large amounts of data and sending back the computed results can lead to high latency, which can affect the user experience.

In-network computation is considered a promising solution that can distribute the computational load across available devices. The computational functions can be performed within the network, bringing the computations close to the user's device. As a

result, it can meet the time-critical requirement while significantly reducing bandwidth consumption.

ActiveNDN provides in-network computational capabilities by allowing computations to be performed in nodes with function call forwarding. However, in-network programmability has not been considered. The programs or function codes executed are preconfigured by the network administrator. With a large sensor deployment, it is very time-consuming to manually add or update the functions in the network. In this context, the function compiler plays an important role by accepting new programs or functions from software developers. The compiled function codes ready for installation can be orchestrated to all available devices in the network. This will be our future research direction by extending the ActiveNDN into a programmable distributed IoT system where high-dimensional data can be processed with a variety of computational programs.

Another research direction is related to the dynamic adaptation of ActiveNDN parameters. For example, the feature set needs to be manually configured at the beginning of the running function. Moreover, there is still a lack of adaptive network topology setting. This leads to inappropriate configuration, as too large function scope can incur long delays in computation, while too small a setting can have an impact on data inclusion. The optimization model to select an appropriate feature set based on network topology and network size is essential for our future study. The exploratory or probing mechanism is required for future development to obtain network information for the model that can be applied with some machine learning algorithms.

Bibliography

[Amadeo et al., 2014a] Amadeo, M., Campolo, C., Iera, A., & Molinaro, A. (2014a). Named data networking for IoT: An architectural perspective. *2014 European Conference on Networks and Communications (EuCNC)*, 1–5. https://doi.org/10.1109/EuCNC.2014.6882665

[Amadeo et al., 2014b] Amadeo, M., Campolo, C., & Molinaro, A. (2014b). Multi-source Data Retrieval in IoT via Named Data Networking. *Proceedings of the 1st ACM Conference on Information-Centric Networking*, ACM-ICN '14, 67–76. https://doi.org/10.1145/2660129.2660148. event-place: Paris, France

[Amadeo et al., 2015] Amadeo, M., Campolo, C., & Molinaro, A. (2015). Forwarding strategies in named data wireless ad hoc networks: Design and evaluation. *Journal of Network and Computer Applications*, 50, 148–158. https://doi.org/10.1016/j.jnca.2014.06.007

[Amadeo et al., 2013a] Amadeo, M., Campolo, C., Molinaro, A., & Mitton, N. (2013a). Named Data Networking: A natural design for data collection in Wireless Sensor Networks. *2013 IFIP Wireless Days (WD)*, 1–6. https://doi.org/10.1109/WD.2013.6686486

[Amadeo et al., 2018] Amadeo, M., Campolo, C., Molinaro, A., & Ruggeri, G. (2018). IoT Data Processing at the Edge with Named Data Networking. *European Wireless 2018; 24th European Wireless Conference*, 1–6.

[Amadeo et al., 2013b] Amadeo, M., Molinaro, A., & Ruggeri, G. (2013b). E-CHANET: Routing, forwarding and transport in Information-Centric multihop wireless networks. *Computer Communications*, 36(7), 792–803. https://doi.org/10.1016/j.comcom.2013.01.006

[Amadeo et al., 2019a] Amadeo, M., Ruggeri, G., Campolo, C., & Molinaro, A. (2019a). IoT Services Allocation at the Edge via Named Data Networking: From Optimal Bounds to Practical Design. *IEEE Transactions on Network and Service Management*, 16(2), 661–674. https://doi.org/10.1109/TNSM.2019.2900274. Conference Name: IEEE Transactions on Network and Service Management

[Amadeo et al., 2019b] Amadeo, M., Ruggeri, G., Campolo, C., Molinaro, A., Loscri, V., & Calafate, C. (2019b). Fog Computing in IoT Smart Environments via Named Data Networking: A Study on Service Orchestration Mechanisms. *Future Internet*, 11, 222. https://doi.org/10.3390/fi11110222

[Angius et al., 2012] Angius, F., Gerla, M., & Pau, G. (2012). BLOOGO: BLOOm filter based GOssip algorithm for wireless NDN. *Proceedings of the 1st ACM workshop on Emerging Name-Oriented Mobile Networking Design - Architecture, Algorithms, and Applications*, NoM '12, 25–30. https://doi.org/10.1145/2248361.2248369

[Arbib et al., 2019] Arbib, C., Arcelli, D., Dugdale, J., Moghaddam, M. T., & Muccini, H. (2019). Real-time Emergency Response through Performant IoT Architectures. *International Conference on Information Systems for Crisis Response and Management (ISCRAM)*. https://hal.archives-ouvertes.fr/hal-02091586

[Arumaithurai et al., 2014] Arumaithurai, M., Chen, J., Monticelli, E., Fu, X., & Ramakrishnan, K. K. (2014). Exploiting ICN for flexible management of software-defined networks. *Proceedings of the 1st international conference on Information-centric networking - INC '14*, 107–116. https://doi.org/10.1145/2660129.2660147

[Baccelli et al., 2014] Baccelli, E., Mehlis, C., Hahm, O., Schmidt, T. C., & Wählisch, M. (2014). Information centric networking in the IoT: experiments with NDN in the wild. *Proceedings of the 1st ACM Conference on Information-Centric Networking*, ACM-ICN '14, 77–86. https://doi.org/10.1145/2660129.2660144

[Bonnet et al., 2000] Bonnet, P., Gehrke, J., & Seshadri, P. (2000). Querying the physical world. *IEEE Personal Communications*, 7(5), 10–15. https://doi.org/10.1109/98.878531

[Bonomi et al., 2012] Bonomi, F., Milito, R., Zhu, J., & Addepalli, S. (2012). Fog computing and its role in the internet of things. *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, MCC '12, 13–16. https://doi.org/10.1145/2342509.2342513

[Braun et al., 2011] Braun, T., Hilt, V., Hofmann, M., Rimac, I., Steiner, M., & Varvello, M. (2011). Service-Centric Networking. *2011 IEEE International Conference on Communications Workshops (ICC)*, 1–6. https://doi.org/10.1109/iccw.2011.5963587. ISSN: 2164-7038

[CCN-lite,] CCN-lite. *CCN-lite*. https://github.com/cn-uofbasel/ccn-lite

[Collela, 2017] Collela, P. (2017). *Ushering In A Better Connected Future*. https://www.ericsson.com/en/about-us/company-facts/ericsson-worldwide/

india/authored-articles/ushering-in-a-better-connected-future. Last Modified: 2021-02-23T06:31:07+00:00

[Dannewitz et al., 2013] Dannewitz, C., Kutscher, D., Ohlman, B., Farrell, S., Ahlgren, B., & Karl, H. (2013). Network of Information (NetInf) – An information-centric networking architecture. *Computer Communications*, 36(7), 721–735. https://doi.org/10.1016/j.comcom.2013.01.009

[Dauphin et al., 2018] Dauphin, L., Baccelli, E., & Adjih, C. (2018). RIOT-ROS2: Low-Cost Robots in IoT Controlled via Information-Centric Networking. *2018 IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*, 1–6. https://doi.org/10.23919/PEMWN.2018.8548798

[De Silva et al., 2016] De Silva, U., Lertsinsrubtavee, A., Sathiaseelan, A., Molina-Jimenez, C., & Kanchanasut, K. (2016). Implementation and evaluation of an information centric-based smart lighting controller. *Proceedings of the 12th Asian Internet Engineering Conference*, AINTEC '16, 1–8. https://doi.org/10.1145/3012695.3012696

[Diao et al., 2007] Diao, Y., Ganesan, D., Mathur, G., & Shenoy, P. J. (2007). Rethinking Data Management for Storage-centric Sensor Networks. *CIDR*.

[Djama et al., 2021] Djama, A., Djamaa, B., Senouci, M. R., & Khemache, N. (2021). LAFS: a learning-based adaptive forwarding strategy for NDN-based IoT networks. *Annals of Telecommunications*. https://doi.org/10.1007/s12243-021-00850-2

[Dubey et al., 2019] Dubey, V., Kumar, P., & Chauhan, N. (2019). Forest Fire Detection System Using IoT and Artificial Neural Network. *International Conference on Innovative Computing and Communications*, Lecture Notes in Networks and Systems, 323–337. https://doi.org/10.1007/978-981-13-2324-9_33

[Fotiou et al., 2012] Fotiou, N., Nikander, P., Trossen, D., & Polyzos, G. C. (2012). Developing Information Networking Further: From PSIRP to PURSUIT. *Broadband Communications, Networks, and Systems*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 1–13. https://doi.org/10.1007/978-3-642-30376-0_1

[Gao et al., 2016] Gao, S., Zhang, H., & Zhang, B. (2016). *Energy Efficient Interest Forwarding in NDN-Based Wireless Sensor Networks*. https://doi.org/https://doi.org/10.1155/2016/3127029

[Giridhar & Kumar, 2006] Giridhar, A. & Kumar, P. R. (2006). Toward a theory of in-network computation in wireless sensor networks. *IEEE Communications Magazine*, 44(4), 98–107. https://doi.org/10.1109/MCOM.2006.1632656

[Govindan et al., 2002] Govindan, R., Hellerstein, J. M., Hong, W., Madden, S., Franklin, M., & Shenker, S. (2002). *The Sensor Network as a Database*.

[Grassi et al., 2014] Grassi, G., Pesavento, D., Pau, G., Vuyyuru, R., Wakikawa, R., & Zhang, L. (2014). VANET via Named Data Networking. *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 410–415. https://doi.org/10.1109/INFCOMW.2014.6849267

[Grassi et al., 2015] Grassi, G., Pesavento, D., Pau, G., Zhang, L., & Fdida, S. (2015). Navigo: Interest forwarding by geolocations in vehicular Named Data Networking. *2015 IEEE 16th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 1–10. https://doi.org/10.1109/WoWMoM.2015.7158165

[Guo et al., 2021] Guo, X., Yang, S., Cao, L., Wang, J., & Jiang, Y. (2021). A new solution based on optimal link-state routing for named data MANET. *China Communications*, 18(4), 213–229. https://doi.org/10.23919/JCC.2021.04.016. Conference Name: China Communications

[Gündoğan et al., 2018] Gündoğan, C., Kietzmann, P., Schmidt, T. C., & Wählisch, M. (2018). HoPP: Robust and Resilient Publish-Subscribe for an Information-Centric Internet of Things. *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, 331–334. https://doi.org/10.1109/LCN.2018.8638030. ISSN: 0742-1303

[Gündoğran et al., 2018] Gündoğran, C., Kietzmann, P., Lenders, M., Petersen, H., Schmidt, T. C., & Wählisch, M. (2018). NDN, CoAP, and MQTT: a comparative measurement study in the IoT. *Proceedings of the 5th ACM Conference on Information-Centric Networking*, ICN '18, 159–171. https://doi.org/10.1145/3267955.3267967

[Heidemann et al., 2001] Heidemann, J., Silva, F., Intanagonwiwat, C., Govindan, R., Estrin, D., & Ganesan, D. (2001). Building Efficient Wireless Sensor Networks with Low-level Naming. *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, 146–159. https://doi.org/10.1145/502034.502049. event-place: Banff, Alberta, Canada

[Hoque et al., 2013] Hoque, A. K. M. M., Amin, S. O., Alyyan, A., Zhang, B., Zhang, L., & Wang, L. (2013). NLSR: Named-data Link State Routing Protocol. *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-centric Networking*, ICN '13, 15–20. https://doi.org/10.1145/2491224.2491231. event-place: Hong Kong, China

[Jacobson et al., 2009] Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., & Braynard, R. L. (2009). Networking Named Content. *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, 1–12. https://doi.org/10.1145/1658939.1658941. event-place: Rome, Italy

[Jacquet et al., 2001] Jacquet, P., Muhlethaler, P., Clausen, T., Laouiti, A., Qayyum, A., & Viennot, L. (2001). Optimized link state routing protocol for ad hoc networks. *Proceedings. IEEE International Multi Topic Conference, 2001. IEEE INMIC 2001. Technology for the 21st Century.*, 62–68. https://doi.org/10.1109/INMIC.2001.995315

[Kanabkaew et al., 2019] Kanabkaew, T., Mekbungwan, P., Raksakietisak, S., & Kanchanasut, K. (2019). Detection of pm2.5 plume movement from iot ground level monitoring data. *Environmental Pollution*, 252, 543–552. https://doi.org/https://doi.org/10.1016/j.envpol.2019.05.082

[Kanchanasut et al., 2018] Kanchanasut, K., Lertsinsubtavee, A., Tunpan, A., Tansakul, N., Mekbungwan, P., Weshsuwannarugs, N., & Tripatana, P. (2018). Building last-metre community networks in thailand. *Global Information Society Watch*, 232–23.

[Koponen et al., 2007] Koponen, T., Chawla, M., Chun, B.-G., Ermolinskiy, A., Kim, K. H., Shenker, S., & Stoica, I. (2007). A data-oriented (and beyond) network architecture. *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, 181–192. https://doi.org/10.1145/1282380.1282402

[Król et al., 2019] Król, M., Mastorakis, S., Oran, D., & Kutscher, D. (2019). Compute first networking: Distributed computing meets icn. *Proceedings of the 6th ACM Conference on Information-Centric Networking*, ICN '19, 67–77. https://doi.org/10.1145/3357150.3357395

[Król et al., 2018] Król, M., Habak, K., Oran, D., Kutscher, D., & Psaras, I. (2018). RICE: remote method invocation in ICN. *Proceedings of the 5th ACM Conference on Information-Centric Networking*, ICN '18, 1–11. https://doi.org/10.1145/3267955.3267956

[Kunze et al., 2021] Kunze, I., Glebke, R., Scheiper, J., Bodenbenner, M., Schmitt, R. H., & Wehrle, K. (2021). Investigating the Applicability of In-Network Computing to Industrial Scenarios. *2021 4th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS)*, 334–340. https://doi.org/10.1109/ICPS49255.2021.9468247

[Kutscher et al., 2021] Kutscher, D., Al Wardani, L., & Rayhan Gias, T. M. (2021). Vision: information-centric dataflow: re-imagining reactive distributed computing. *Proceedings of the 8th ACM Conference on Information-Centric Networking*, ICN '21, 52–58. https://doi.org/10.1145/3460417.3482975

[Liang et al., 2020] Liang, T., Pan, J., Rahman, M. A., Shi, J., Pesavento, D., Afanasyev, A., & Zhang, B. (2020). Enabling Named Data Networking Forwarder to Work Out-of-the-Box at Edge Networks. *2020 IEEE International Conference on Communications*

Workshops (ICC Workshops), 1–6. https://doi.org/10.1109/ICCWorkshops49005.2020.9145304

[Liao et al., 2019] Liao, Z., Teng, Z., Zhang, J., Liu, Y., Xiao, H., & Yi, A. (2019). A Semantic Concast service for data discovery, aggregation and processing on NDN. *Journal of Network and Computer Applications*, 125, 168–178. https://doi.org/10.1016/j.jnca.2018.10.017

[Liu et al., 2016] Liu, L., Xie, L. T., Bahrami, M., Peng, Y., Ito, A., Mnatsakanyan, S., Qu, G., Ye, Z., & Guo, H. (2016). Demonstration of a Functional Chaining System Enabled by Named-Data Networking. *Proceedings of the 3rd ACM Conference on Information-Centric Networking*, ACM-ICN '16, 227–228. https://doi.org/10.1145/2984356.2985239. event-place: Kyoto, Japan

[Lu et al., 2013] Lu, Y., Zhou, B., Tung, L.-C., Gerla, M., Ramesh, A., & Nagaraja, L. (2013). Energy-efficient content retrieval in mobile cloud. *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, MCC '13, 21–26. https://doi.org/10.1145/2491266.2491271

[Luan et al., 2016] Luan, T. H., Gao, L., Li, Z., Xiang, Y., Wei, G., & Sun, L. (2016). Fog Computing: Focusing on Mobile Users at the Edge. *arXiv:1502.01815 [cs]*. http://arxiv.org/abs/1502.01815. arXiv: 1502.01815

[Madden et al., 2005] Madden, S. R., Franklin, M. J., Hellerstein, J. M., & Hong, W. (2005). TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1), 122–173. https://doi.org/10.1145/1061318.1061322

[Mastorakis et al., 2017] Mastorakis, S., Afanasyev, A., & Zhang, L. (2017). On the evolution of ndnSIM: an open-source simulator for NDN experimentation. *ACM Computer Communication Review*.

[Mastorakis et al., 2020] Mastorakis, S., Mtibaa, A., Lee, J., & Misra, S. (2020). ICedge: When Edge Computing Meets Information-Centric Networking. *IEEE Internet of Things Journal*, 7(5), 4203–4217. https://doi.org/10.1109/JIOT.2020.2966924. Conference Name: IEEE Internet of Things Journal

[Meisel et al., 2010] Meisel, M., Pappas, V., & Zhang, L. (2010). Ad Hoc Networking via Named Data. *Proceedings of the Fifth ACM International Workshop on Mobility in the Evolving Internet Architecture*, MobiArch '10, 3–8. https://doi.org/10.1145/1859983.1859986. event-place: Chicago, Illinois, USA

[Montpetit, 2019] Montpetit, M.-J. (2019). In Network Computing Enablers for Extended Reality. Internet Draft draft-montpetit-coin-xr-03, Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-montpetit-coin-xr. Num Pages: 10

[Mosko et al., 2019] Mosko, M., Solis, I., & Wood, C. A. (2019). Content-Centric Networking (CCNx) Semantics. Request for Comments RFC 8569, Internet Engineering Task Force. https://doi.org/10.17487/RFC8569

[NDN, a] NDN. *NDN Packet Format Specification 0.2.1 documentation.* http://named-data.net/doc/NDN-packet-spec/0.2.1/

[NDN, b] NDN. *NFD - Named Data Networking Forwarding Daemon.* http://named-data.net/doc/NFD

[NDN, c] NDN. *NFD Developer's Guide.* https://named-data.net/publications/techreports/ndn-0021-7-nfd-developer-guide

[NDN,] NDN. *PyNDN: A Named Data Networking client library with TLV wire format support in native Python.* https://github.com/named-data/PyNDN2

[NDN,] NDN. *"NDN protocol design principles".* http://named-data.net/project/ndn-design-principles

[NS3, a] NS3. *ns-3 : a discrete-event network simulator for internet systems.* https://www.nsnam.org/

[NS3, b] NS3. *ns-3 Wi-Fi Module's Design Documentation.* https://www.nsnam.org/docs/models/html/wifi-design.html

[Poslad et al., 2015] Poslad, S., Middleton, S. E., Chaves, F., Tao, R., Necmioglu, O., & Bügel, U. (2015). A Semantic IoT Early Warning System for Natural Environment Crisis Management. *IEEE Transactions on Emerging Topics in Computing*, 3(2), 246–257. https://doi.org/10.1109/TETC.2015.2432742. Conference Name: IEEE Transactions on Emerging Topics in Computing

[Preden et al., 2015] Preden, J. S., Tammemäe, K., Jantsch, A., Leier, M., Riid, A., & Calis, E. (2015). The Benefits of Self-Awareness and Attention in Fog and Mist Computing. *Computer*, 48(7), 37–45. https://doi.org/10.1109/MC.2015.207. Conference Name: Computer

[Qiao et al., 2018] Qiao, Y., Nolani, R., Gill, S., Fang, G., & Lee, B. (2018). ThingNet: A micro-service based IoT macro-programming platform over edges and cloud. *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 1–4. https://doi.org/10.1109/ICIN.2018.8401626

[Satyanarayanan, 2017] Satyanarayanan, M. (2017). The Emergence of Edge Computing. *Computer*, 50(1), 30–39. https://doi.org/10.1109/MC.2017.9. Conference Name: Computer

[Saxena & Raychoudhury, 2019] Saxena, D. & Raychoudhury, V. (2019). Design and Verification of an NDN-Based Safety-Critical Application: A Case Study With Smart Healthcare. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(5), 991–1005. https://doi.org/10.1109/TSMC.2017.2723843. Conference Name: IEEE Transactions on Systems, Man, and Cybernetics: Systems

[Scherb et al., 2018] Scherb, C., Grewe, D., Wagner, M., & Tschudin, C. (2018). Resolution strategies for networking the IoT at the edge via named functions. *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*, 1–6. https://doi.org/10.1109/CCNC.2018.8319235

[SEA-HAZEMON,] SEA-HAZEMON. *STIC-ASIA: SEA-HAZEMON low-cost real-time monitoring of haze air quality disasters in rural communities in thailand and southeast asia*. https://interlab.ait.ac.th/HAZEMON/

[Seltman, 2018] Seltman, H. J. (2018). *Experimental Design and Analysis*. Department of Statistics at Carnegie Mellon (Online Only). https://www.stat.cmu.edu/~hseltman/309/Book/Book.pdf

[Shang et al., 2016] Shang, W., Bannis, A., Liang, T., Wang, Z., Yu, Y., Afanasyev, A., Thompson, J., Burke, J., Zhang, B., & Zhang, L. (2016). Named Data Networking of Things (Invited Paper). *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 117–128. https://doi.org/10.1109/IoTDI.2015.44

[Shang et al., 2017] Shang, W., Wang, Z., Afanasyev, A., Burke, J., & Zhang, L. (2017). Breaking Out of the Cloud: Local Trust Management and Rendezvous in Named Data Networking of Things. *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 3–14.

[Shelby et al., 2014] Shelby, Z., Hartke, K., & Bormann, C. (2014). *The Constrained Application Protocol (CoAP)*. RFC 7252. https://doi.org/10.17487/RFC7252

[Sifalakis et al., 2014] Sifalakis, M., Kohler, B., Scherb, C., & Tschudin, C. (2014). An Information Centric Network for Computing the Distribution of Computations. *Proceedings of the 1st ACM Conference on Information-Centric Networking*, ACM-ICN '14, 137–146. https://doi.org/10.1145/2660129.2660150. event-place: Paris, France

[Tennenhouse et al., 1997] Tennenhouse, D. L., Smith, J. M., Sincoskie, W. D., Wetherall, D. J., & Minden, G. J. (1997). A survey of active network research. *IEEE Communications Magazine*, 35(1), 80–86. https://doi.org/10.1109/35.568214

[Torrent-Moreno et al., 2006] Torrent-Moreno, M., Corroy, S., Schmidt-Eisenlohr, F., & Hartenstein, H. (2006). IEEE 802.11-based one-hop broadcast communications: understanding transmission success and failure under different radio propagation environments. *Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems*, MSWiM '06, 68–77. https://doi.org/10.1145/1164717.1164731

[Tschudin & Sifalakis, 2014] Tschudin, C. & Sifalakis, M. (2014). Named functions and cached computations. *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, 851–857. https://doi.org/10.1109/CCNC.2014.6940518

[UDOO,] UDOO. *Discover the UDOO NEO*. https://www.udoo.org/udoo-neo/

[Wang et al., 2012] Wang, L., Afanasyev, A., Kuntz, R., Vuyyuru, R., Wakikawa, R., & Zhang, L. (2012). Rapid Traffic Information Dissemination Using Named Data. *Proceedings of the 1st ACM Workshop on Emerging Name-Oriented Mobile Networking Design - Architecture, Algorithms, and Applications*, NoM '12, 7–12. https://doi.org/10.1145/2248361.2248365. event-place: Hilton Head, South Carolina, USA

[Ye et al., 2016] Ye, Y., Qiao, Y., Lee, B., & Murray, N. (2016). PIoT: Programmable IoT using Information Centric Networking. *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, 825–829. https://doi.org/10.1109/NOMS.2016.7502908

[Yu et al., 2013] Yu, Y.-T., Dilmaghani, R. B., Calo, S., Sanadidi, M. Y., & Gerla, M. (2013). Interest propagation in named data manets. *2013 International Conference on Computing, Networking and Communications (ICNC)*, 1118–1122. https://doi.org/10.1109/ICCNC.2013.6504249

[Zhang et al., 2014] Zhang, L., Afanasyev, A., Burke, J., Jacobson, V., claffy, k., Crowley, P., Papadopoulos, C., Wang, L., & Zhang, B. (2014). Named data networking. *SIGCOMM Comput. Commun. Rev.*, 44(3), 66–73. https://doi.org/10.1145/2656877.2656887

[Zhang et al., 2017] Zhang, M., Lehman, V., & Wang, L. (2017). Scalable name-based data synchronization for named data networking. *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 1–9. https://doi.org/10.1109/INFOCOM.2017.8057193