Information Centric Networking based Collaborative Edge Computing Framework for the Internet of Things

Qian Wang

Thesis presented for the degree of

Doctor of Philosophy

to the

Technological University of the Shannon:

Midlands Midwest

Supervisors:

Dr. Yuansong Qiao

Dr. Brian Lee

Dr. Niall Murray

Submitted to the Technological University of the Shannon: Midlands Midwest,

August 2023

Abstract

The Internet of Things (IoT) has connected billions of devices and its proliferation will continue. As IoT grows, so do the volumes of data it produced and exchanged. The challenge lies in efficiently processing the massive amounts of IoT data. Moreover, IoT applications prioritize extracting meaningful knowledge rather than building connections with multiple devices. This results in a mismatch between the host-centric nature of the current Internet and the information-centric demands of IoT applications.

To address these challenges, this thesis presents an Information Centric Networking (ICN) based collaborative edge computing framework for distributed IoT data processing. Firstly, the functional architecture is investigated to enable in-network data processing in IoT edge environments. Within this architecture, three software components, namely Computation Manager, Computation Executor and Function Repository, collaborate to resolve, deploy and execute IoT jobs. This thesis leverages the powerful and prevalent MapReduce paradigm in the architecture design. The ICN-based implementation empowers MapReduce job execution by categorizing Computation Executors as mappers and reducers, developing a distributed computational job tree construction protocol for the Computation Manager, and defining an ICN naming scheme for request expression and data/function acquisition. The Function Repository is distributed and maintained by each Computation Executor, which retrieves and saves functions by parsing users' requests. Experimental simulations have verified the feasibility of the proposed design and demonstrated its effectiveness in reducing network traffic.

Secondly, this thesis improves the proposed ICN-based computing framework by considering the resource constraints of heterogenous edge devices. It classifies edge devices into two types: processing-capable nodes (i.e. mappers and reducers) and forwarding-only nodes (called forwarders). Both types of nodes join in the computational job tree construction procedure. A job maintenance scheme is

developed to disseminate IoT jobs to appropriate devices and coordinate their collaboration in serving multiple jobs simultaneously. Performance evaluation tests have confirmed the effectiveness of the proposed framework, indicating decreased network traffic compared to the centralized data processing approach.

Thirdly, this thesis enhances the proposed framework to ensure exactly once data computation. Interruptions in IoT network connections during edge collaboration can lead to data loss or duplicated data transmission and processing, which is unacceptable for IoT applications with exactly once computation requirement. Although checkpoint-based schemes have been successfully developed in traditional big data processing frameworks to achieve exactly once data delivery/processing, it is challenging to directly apply these solutions in IoT scenarios due to the differences between IoT networks and datacentre environments. This thesis identifies three specific challenges of achieving exactly once computation in IoT collaborative edge scenarios and devises a five-phase protocol to address them. The proposed protocol consists of a job execution procedure for normal job operations and a job recovery procedure to handle network failures. Simulation tests have shown that the proposed design outperforms the checkpoint-based benchmark solution in terms of network traffic and job execution time.

Declaration

I hereby declare that this thesis, which I now submit for the assessment in the program of research leading to the award of Doctor of Philosophy at Technological University of the Shannon: Midlands Midwest, is entirely my original work and has not been plagiarized from the work of others.

I affirm the following:

- This research was conducted wholly during my candidature for the research degree at this University.

- In cases where I have consulted published works by others, proper attribution has always been provided.

- Whenever I have quoted from external sources, the original sources have been duly acknowledged. Aside from these quotations, the entire thesis is the product of my own work.

- I have acknowledged all significant sources of assistance received during this research.

Signed: ______________________________

Qian Wang

Date: ______________________________

Acknowledgement

I would like to express my deep gratitude to my main supervisor, Dr. Yuansong Qiao, for his invaluable guidance and unwavering support throughout the journey of this PhD project. His expertise, insightful feedback, and patience have been instrumental in shaping this research work and pushing me to achieve my best. I am also truly grateful to my co-supervisors, Dr. Brian Lee and Dr. Niall Murray, for their fruitful discussions, shared their expertise, and provided assistance during various stages of this research project.

Furthermore, I am thankful to the fellow researchers and colleagues in the Software Research Institute for their knowledge sharing. Their insights and enthusiasm have played a significant role in making this research experience wonderful.

I extend my heartfelt appreciation to my family for their unconditional love and belief in my abilities throughout this challenging journey. Their encouragement, sacrifices, and understanding have been the driving force behind my pursuit of academic aspirations.

List of Figures

Figure 1 Thesis Outline 12
Figure 2 Literature Review Outline 15
Figure 3 IP and NDN Architecture [29] 16
Figure 4 NDN Packet Processing [30] 17
Figure 5 MapReduce/HDFS Framework 19
Figure 6 Functional Architecture 36
Figure 7 Workflow Illustration 38
Figure 8 Workflow of NDN-based MapReduce Job Execution 44
Figure 9 Network Topology for Tests 48
Figure 10 Network Traffic Comparison 49
Figure 11 Number of Interest at User Node 49
Figure 12 Network Topology Example: Original IoT Vs. Proposed_design 55
Figure 13 Application Layer Functionality for Multiple Jobs Execution 58
Figure 14 Network Topology and Generated Job Trees 63
Figure 15 Network Traffic Comparison 69
Figure 16 Five Phases of the Proposed Protocol 74
Figure 17 Illustration of ID Allocation 78
Figure 18 Illustration of Nodes' ID Table 81
Figure 19 Procedure of Job State Sync Phase 92
Figure 20 Job Tree Built and Updated by the Proposed_design 99
Figure 21 Traffic Generated by Nodes on Job Tree 101
Figure 22 Network Traffic Comparison: Proposed_design Vs. CP_Benchmark... 103
Figure 23 Job Graph on BRITE-Topology 106
Figure 24 Network Traffic Comparison on BRITE-Topology 108
Figure 25 Overhead of Job Computation Records Storage 110
Figure 26 Node ID affected by Job Tree Depth 111
Figure 27 Overhead of ID Update 112

List of Acronyms

ACK	Acknowledgement
AI	Artificial Intelligence
BJT-Table	BuildJobTree Table
CCN	Content-centric Networking
CCTV	Closed-Circuit Television
CPU	Central Processing Unit
CRF	Clear-Record-Frequency
CR Table	Computation Record Table
CS	Content Store
DB	Database
dinrg	Decentralization of the Internet Research Group
FaaS	Function-as-a-Service
FIB	Forwarding Information Base
FL	Federated Learning
HDFS	Hadoop Distributed File System
icnrg	Information Centric Networking Research Group
ICN	Information Centric Networking
ID	Identification
IIoT	Industrial IoT
IP	Internet Protocol
IoT	Internet of Things
JS Table	Job State Table
JT-Table	Job Tree Table
MEC	Mobile Edge Computing
NFN	Named Function Networking
NDN	Named Data Networking
PIT	Pending Interest Table
PJ-Table	Pending Job Table
QoS	Quality of Service
RAM	Random Access Memory
RSC phases	Job Tree **R**ebuild phase Job State **S**ync/**C**ommit phase

RSUs	Road-Side Units
TF-Table	Task Function Table
UF-M	User-defined Map Function
UF-R	User-defined Reduce Function
VM	Virtual Machine
V2X	Vehicle-to-Vehicle/Infrastructure
WSN	Wireless Sensor Network

Table of Contents

1 Introduction 1
1.1 Research Motivation 1
1.2 Research Question 3
1.3 Scope of Work 6
1.4 Contributions and Publications 6
1.5 Thesis Layout 11
2 Literature Review 15
2.1 NDN Technology Background 16
2.2 MapReduce/HDFS 18
2.3 Collaborative Edge Computing in IoT 22
2.3.1 Single-layer Edge Computing 22
2.3.2 Hierarchical Edge Collaboration 24
2.4 ICN based Edge Computing for IoT 25
2.5 Exactly Once Data Processing 30
2.5.1 Datacentre-oriented Solutions 30
2.5.2 IoT-related Solutions 31
2.6 Distributed Consensus Protocol 32
3 Functional Architecture and its ICN-based Implementation for MapReduce Jobs 35
3.1 Functional Architecture Overview 35
3.2 Illustration of Workflow in the Functional Architecture 37
3.3 ICN-based Implementation for MapReduce Jobs 39
3.3.1 Functional Units Instantiation 39
3.3.2 Computational Job Tree Construction 40
3.3.3 MapReduce-Job Execution 43
3.4 Evaluation 45
3.4.1 Test Design 45
3.4.2 Test Results and Analysis 48
3.5 Summary 50
4 Protocol for Multiple MapReduce Jobs Execution with Resource Constraints53
4.1 Concept Overview 53
4.2 Computation Job Tree Construction and Maintenance 56
4.3 Multiple Jobs Execution 60
4.4 Evaluation 62
4.4.1 Feasibility Test 63

4.4.2 Network Traffic Comparison 64
4.5 Summary 70
5 Protocol for Exactly Once Data Computation 71
5.1 Motivation 71
5.2 Five-phase Protocol Design 74
5.2.1 Job Tree Build Phase 75
5.2.2 Job Execute Phase 75
5.2.3 Job State Commit Phase 83
5.2.4 Job Tree Rebuild Phase 84
5.2.5 Job State Sync Phase 88
5.3 Protocol Overhead Analysis 93
5.3.1 Network Traffic Overhead 93
5.3.2 Computation Record Storage Overhead 96
5.4 Evaluation 97
5.4.1 Feasibility Test 98
5.4.2 Network Traffic Overhead Analysis 102
5.4.3 Overhead of Computation Record Storage 109
5.4.4 Overhead of ID Allocation and Update 111
5.5 Summary 114
6 Conclusion and Future Work 117
6.1 Conclusion 117
6.2 Future Work 121
Reference 123

1 Introduction

1.1 Research Motivation

The Internet of Things (IoT) refers to physical objects with sensing and/or processing ability to capture and exchange data with each other over the Internet [1]. In recent years, IoT has emerged as a crucial facilitator for numerous smart systems [2] and its growth and adoption have been steadily increasing. The report from DELL Technologies [3] estimates that the number of IoT devices will be 41.6 billion in 2025, which could generate 79.4 zettabytes of data. As a result, two significant challenges have arisen. The first challenge is efficiently processing the enormous volumes of data from IoT devices. The second challenge involves refining the IoT data processing mechanism to better align with the information-oriented nature of IoT applications, ensuring optimal performance and relevance.

Edge computing [4] moves the data computation and storage to the edge of the network, which has emerged as a widely accepted model for IoT data processing due to the following reasons. Firstly, although cloud servers have rich power and resources, the influx of enormous IoT data poses non-trivial challenges for cloud-only computing [5]. Edge computing is valued as the complementary of cloud computing by bringing cloud capabilities at the proximity of IoT devices. Secondly, it is the fact that many IoT systems are deployed spanning across wide geographical areas, such as Smart City [6] and Industrial IoT (IIoT) [7]. These data samples are collected in geographically dispersed locations and they can be processed practically without adversely affecting each other. Thirdly, a growing number of IoT-connected devices have computing capabilities thanks to the advances in hardware technologies.

The potential of edge devices should be explored to prune and/or aggregate data before transmission, thereby reducing IoT network traffic.

Along with the above benefits, edge computing meanwhile brings challenges in processing data in IoT scenarios. When compared with cloud servers, edge devices are still constrained on processing capability and storage space. Consequently, the execution of computationally intensive tasks may overwhelm an individual edge server. Moreover, the utilization of multiple edge devices becomes essential for completing an IoT task because of its widespread nature of sensing data. To this end, many factors are involved to successfully execute an IoT task using edge computing paradigm, such as generating task-specific execution plans, distribution and deployment of IoT tasks across edge devices and the synergy scheme to coordinate multiple edge nodes participating in one IoT task.

More challenges arise in consideration of the information-oriented nature of IoT applications which prioritizes to obtain meaningful knowledge analysed from lots of raw data. It is not the primary concern of IoT applications to know which edge device processes specific data samples (although security concerns necessitate identity verification, which is out the range of this thesis). In addition, it is impractical for IoT users to be aware of the capabilities of each IoT edge device and then establish connections with the corresponding ones for task allocation. As a result, the existing host-centric Internet communication model inherently falls short in meeting the above requirements. The novel Information Centric Networking (ICN) [8] addresses these limitations by focusing on the data/information itself rather than its physical location. It provides name-based content forwarding, which aligns more effectively with the information-oriented nature of IoT applications than the current Internet

Protocol (IP). Thus, ICN offers the possibility to be a more appropriate network substrate to build an IoT edge computing paradigm.

1.2 Research Question

The aim of this thesis is to answer the research question defined below:

"Is it possible to build an ICN based edge computing framework to efficiently process massive amounts of data for various IoT applications?"

The research question is broken down into the following research objectives:

- Propose an ICN based computing framework to support in-network data processing for IoT edge environments, encompassing both IoT and edge devices.
- Design the protocol(s) for computational tasks deployment and execution with the proposed framework considering IoT edge device capabilities.
- Design the protocol(s) to meet the requirements of different computational tasks, with a focus on exactly once data processing.

The expanding IoT has led to the interconnection of billions of devices. However, the current capacity of cloud remains insufficient to handle the immense volume of data generated by numerous IoT devices on a continuous basis. Moreover, it is inefficient and wasteful to transmit non-valuable data, e.g. noisy data samples, to the cloud. In this context, edge computing emerges as a viable solution by alleviating the burden on the cloud by undertaking less complex IoT tasks or part of the complicated IoT tasks. Examples include data filtering at the source nodes and data aggregation on the fly. This thesis aims to tackle the following challenges to deploy and execute IoT tasks in edge environments.

(1) Data processing scheme in ICN-style. ICN treats each data element individually by utilizing unique names, facilitating name-based in-network data forwarding and caching. This characteristic fits well with the information-oriented nature of IoT applications. However, IoT applications usually require data aggregation or filtering at intermediate nodes to derive meaningful information. The original ICN design lacks the functionality to support the in-network data processing for IoT edge environments. Therefore, an ICN-based data processing scheme should be developed to accommodate IoT edge computing framework.

(2) Collaborative execution of IoT tasks by heterogenous edge devices. Edge devices, positioned between IoT end devices and applications, can contribute their resources to execute computational task on the data flowing through them. In real world scenarios, edge devices involve closed-circuit television (CCTV) cameras, servers at base stations, road-side units (RSUs) and edge routers, with substantial variations in performance capabilities. Some devices possess data processing capabilities, while others do not [9] [10] [11]. Differentiating the capability of edge nodes and assigning computation tasks to appropriate ones are the essential pillars of IoT task deployment. Additionally, task execution plan and maintenance are vital for coordinating multiple edge devices working together in a distributed manner. This encompasses task division among edge nodes, functions/data management for different tasks and synchronization of task execution processes, which can be managed by a centralized node or achieved through consensus algorithms [12] for distributed systems.

(3) IoT applications with exactly once data computation requirement. Three types of semantics on data processing/delivery have been defined, i.e., at-least-once, at-most-once and exactly-once [13]. An IoT application may fall into one of the three

types depending on its scenario. This thesis centres on IoT applications with exactly once data computation requirement because of its significance in various domains. For instance, ensuring the accuracy and integrity of financial transactions, such as smart payment systems and Automated Teller Machine (ATM) monitoring tasks, necessitates the implementation of exactly once processing. In critical domains like risk assessment in automated vehicle driving and patient health monitoring, exactly once data processing is imperative to promptly detect and respond to potential dangers. Certain query jobs demand exactly once computation on sensory data to validate the precision of processed results [14]. An example is the accurate counting of abnormal sensory readings in IIoT to track equipment status and trigger alarms when the count of abnormalities surpasses a predefined threshold. The popularity of Federated Learning (FL) based framework [15] [16] [17] for training IoT data in parallel also emphasizes the importance of exactly once processing on each data point at edge devices, which guarantees the accuracy of trained model. Similarly, the correctness of window-based average computation requires each data sample within this window computed exactly once.

Exactly once data delivery/processing has the most stringent standard compared with the other two types. Traditional big data processing frameworks, e.g. Apache Flink [18] and Kafka [19], have developed mature solutions for exactly once data delivery/processing leveraging checkpoint schemes. However, IoT networks differ from data centre environments in terms of unstable network connections, less-powerful edge devices and limited storage space. It faces difficulties to directly apply existing checkpoint-based solutions in IoT scenarios. Initial attempts in IoT areas have borrowed the checkpoint scheme to enable task migration [20] and processing information transfer between different tasks [21]. The limitation is that these works

concentrate on task execution on a single edge device. Consequently, an IoT-tailored approach should be developed to guarantee exactly once data computation in collaborative edge environment.

1.3 Scope of Work

The target IoT applications are those requiring processing the data from multiple static IoT end devices, e.g. temperature sensors, speed sensors on the road and CCTV cameras. While the IoT data is transmitted from data sources to users, the intermediate nodes (e.g. IoT, edge, and cloud devices) along the path undertake data computation or aggregation or forwarding according to their capabilities. The generated job execution plan aims to deploy the data computation to the edge node that is closest to the required data sources.

Although every computing job has specific requirements related to both data and the capabilities of computing devices, it is not the main concern of this thesis to describe computing resources of edge devices and match the proper one for specific jobs. Moreover, security and privacy issues in data retrieval and processing are beyond the scope of this thesis.

1.4 Contributions and Publications

The contributions arising from this research include:

Contribution I:

Proposed a functional architecture for IoT Collaborative Edge Computing and its ICN-based implementation for executing MapReduce jobs [22] [23].

Publications:

- Qian Wang, Brian Lee, Niall Murray, Yuansong Qiao, "IProIoT: an In-network Processing Framework for IoT using Information Centric Networking", The 9th International Conference on Ubiquitous and Future Networks (ICUFN), Milan, Italy, 4-7 July 2017.

- Qian Wang, Brian Lee, Niall Murray, Yuansong Qiao, "MR-IoT: an information centric MapReduce framework for IoT", the 15th IEEE Consumer Communications & Networking Conference (CCNC), Las Vegas, USA, 12-15 January 2018.

To harness the potential of the edge computing framework for efficient IoT data processing, this thesis proposes a functional architecture with three software components to resolve, deploy and execute IoT tasks on edge nodes. The architecture consists of the following components and their functional units:

- **Function Repository:** It stores the processing functions to be applied on IoT data and can be deployed within IoT edge in a centralized or distributed manner.

- **Computation Executor:** The Computation Executor represents an available edge device capable of providing computational services, leveraging its idle computing power. Computation Executors are responsible for *Task Resolution, Function and Data Acquisition, Task Processing* and returning processed results.

- **Computation Manager:** The Computation Manager has comprehensive knowledge of the entire IoT system and is responsible for administrating task execution and managing other edge devices. It maintains the *Computation Resource Database*, e.g. regularly updating available Computation Executor and functions stored in the Function Repository, to facilitate task distribution among Computation Executors. The Computation Manager bridges the gap

between users' requests and appropriate Computation Executor(s). After receiving a task, the Computation Manager performs *Task Resolution* and *Execution Optimization* based on the currently available computing resources. For instance, it strategically assigns the task to the Computation Executor that is closest to the required data source(s) to reduce network traffic. Finally, the Computation Manager initiates *Task Dissemination* according to the execution plan.

The MapReduce programming model [24] has gained significant popularity for distributed processing of big data. It accepts user-defined functions as an input for data processing. IoT applications could gain from this feature to flexibly express data processing logic. In addition, the outputs of MapReduce tasks are standardized as key-value pairs, which simplifies various IoT applications to share different data types. Executing MapReduce jobs on the proposed architecture is achieved through the following implementation based on ICN.

(1) An ICN naming scheme is defined to express desired data content and (map and reduce) functions for each user's request, referred to as an Interest in ICN. More importantly, the naming scheme supports *Task Dissemination* and *Data/Function Acquisition* by using ICN name-based forwarding.

(2) The Function Repository is deployed and maintained by each Computation Executor. As user-defined map and reduce functions are incorporated in the ICN Interest, Computation Executors can acquire the processing function through *Task Resolution* and then save the function locally for *Task Processing*.

(3) A job tree construction protocol, based on the shortest path algorithm, is developed to create the job execution plan. A unique job tree is built with each user as the root node and all desired data source nodes as leaf nodes. It selects appropriate

Computation Executors between the root and leaf nodes to participate in the data processing. Tasks are assigned along the paths on the job tree. The protocol implements the functional units of the Computation Manager.

(4) Computation Executors are grouped into mappers and reducers. Mappers are defined as the stub nodes of IoT edge network, which connect with multiple sensors to collect raw sensing data according to users' Interests. Reducers are the intermediate edge nodes between the root node and mappers, which receive data from child reducers or directly connected mappers. Mappers/Reducers run user-defined map/reduce function to process data.

Contribution II:

Designed the protocol to execute multiple MapReduce jobs on the proposed framework with the consideration of resource constraints on edge devices [25].

Publication:

- Qian Wang, Brian Lee, Niall Murray, Yuansong Qiao, "MR-Edge: a MapReduce-based Protocol for IoT Edge Computing with Resource Constraints," the 16th IEEE Consumer Communications & Networking Conference (CCNC), Las Vegas, USA, 11-14 January 2019.

Due to the heterogeneity of edge devices, some of them have the computational resources to process data while others do not. To address this, the proposed framework further introduces a new type of Computation Executor, i.e. called forwarders, to represent those edge devices with no processing capability. The functionalities of mappers and reducers remain the same as defined in **Contribution I.** Forwarders neither resolve nor run functions embedded in users' Interests. They forward packets from/to their neighbours and integrate all received data into one and then return. Data aggregation helps to reduce the number of transmitted packets [26].

A Computation Executor may change its role depending on the requirements of different jobs.

The proposed protocol enhances the application layer functionalities to support multiple MapReduce jobs execution on the proposed framework. The key enabler is to distinguish each computational job tree and maintain job specific processing information. The proposed design assigns a unique identifier to each computational tree and each job. All Computation Executors locally save the tree/job identifier with corresponding information (e.g. upstream/downstream neighbours on the specific tree, parsed map/reduce functions for a specific job) as a pair to ensure the job execution correctness, i.e. matching raw/computed data samples to their corresponding jobs.

Contribution III:

Developed the protocol to provide exactly once data computation on the proposed framework [27].

Publication:

- Qian Wang, Brian Lee, Niall Murray, Yuansong Qiao, " ECE: Exactly Once Computation for Collaborative Edge in IoT using Information Centric Networking", IEEE Internet of Things Journal, 2023.

IoT network connections between edge devices may be interrupted during job execution. It may result in data loss or duplicated data transmission and/or processing, which are not acceptable for IoT applications with exactly once data computation requirement. Existing checkpoint-based solutions for datacentres [18] [28] are not suitable for IoT collaborative edge computing scenarios because the underlying network topology is normally not considered in datacentre based solutions. However,

in IoT networks, the logical job graph is tightly coupled with the physical network topology. The gain of data processing versus data transmission should be considered when mapping the logical job graph into the physical devices.

The proposed protocol identifies and solves the following three challenges to achieve exactly once computation in collaborative edge computing for IoT data processing: (1) backup essential data processing information in distributed edge nodes, (2) handle network failures during edge collaboration while guarantee exactly once computation on the same data and (3) limited storage space at edge devices to permanently save data processing related information (as required in challenge (1)).

As a solution, a job tree based data identification (ID) assignment approach is devised as the fundamental support to save and delete job processing related information. The proposed protocol consists of a job execution procedure (to solve challenge (1) and (3)) and a job recovery procedure (to solve challenge (2)). The two procedures can coexist with each other. The job execution procedure enables edge nodes on the job tree to achieve a consensus on the data processing plan and remove out-of-date processing related information with exactly once data computation guarantee. The job recovery procedure handles link failures happened during the job execution. It dynamically updates the job tree to eliminate failed links and checks the data delivery (received or un-received) and computation (processed or un-processed) state to avoid data loss and/or duplicated data computation.

1.5 Thesis Layout

The rest of this thesis is organized as below.

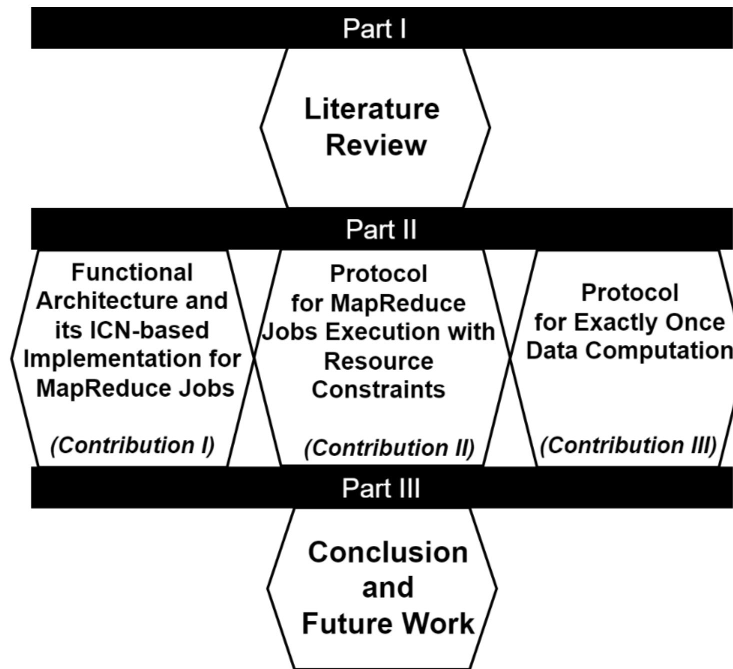


Figure 1 Thesis Outline

Part I:

Chapter 2 briefly introduces the background knowledge of the Named Data Networking (NDN) architecture in Section 2.1, which is one of the most popular and active implementations of ICN and the one chosen for this thesis. In Section 2.2, an overview of the MapReduce programming model is presented, accompanied by an analysis of the challenges associated with applying the original MapReduce framework to IoT scenarios. Section 2.3 delves into a comprehensive review of collaborative edge computing frameworks currently prevalent in the IoT domain. Section 2.4 outlines the advantages of ICN in the context of emerging Internet applications, with a specific focus on IoT edge computing solutions developed on the foundation of ICN. Furthermore, Section 2.5 revisits existing schemes to guarantee exactly once data delivery or processing. The introduction of distributed consensus protocol is described in section 2.6.

Part II:

Chapter 3 (Contribution I) proposes the functional architecture of IoT collaborative edge computing to empower in-network data processing. An ICN-based implementation of the architecture is developed to execute MapReduce type jobs.

Chapter 4 (Contribution II) improves the proposed framework by considering different computing capability of edge devices. A job execution scheme is designed to coordinate multiple edge nodes working together to complete data processing jobs, where some of them are processing-capable while others are not. Moreover, a job maintenance scheme is defined to support multiple MapReduce jobs running on the proposed framework simultaneously.

Chapter 5 (Contribution III) illustrates the protocol design to achieve exactly once data computation on the proposed framework. It contains a job execution procedure to deliver IoT jobs with exactly once data computation guarantee and a recovery procedure to dynamically update the IoT job execution graph while experiencing network failures. A data identification approach based on the job graph is devised to support the proposed functionality.

Part III:

Chapter 6 concludes the whole thesis and discusses potential future works.

2 Literature Review

This chapter provides relevant background of this research project and presents a comprehensive of related literature in the field. Figure 2 illustrates the relationship between each section and this thesis. The technology background of NDN is elucidated in Section 2.1. Section 2.2 provides an overview of the fundamental concepts underpinning the MapReduce architecture and specific challenges of applying MapReduce model into IoT. Research inquiries into collaborative edge computing in IoT scenarios are explored in Section 2.3. In Section 2.4, the focus shifts to existing initiatives aimed at integrating ICN principles into IoT edge computing. The review of methodologies to achieve exactly once data delivery or processing is presented in section 2.5. The convergence of Section 2.3, 2.4 and 2.5 delineates the scope of work undertaken in this thesis. Furthermore, Section 2.6 describes the commonly used consensus protocol in distributed systems, i.e. Two-Phase Commit protocol, which inspires the proposed solution in this thesis to address the defined research objectives.

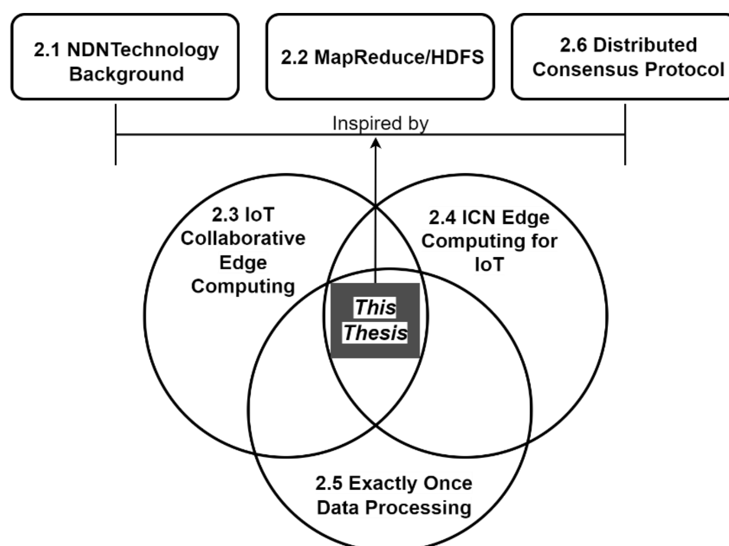


Figure 2 Literature Review Outline

2.1 NDN Technology Background

Similar with the current hourglass architecture in IP, NDN revolutionizes the thin waist of the network by using data names, as shown in Figure 3. Thus, it regards data content as the first-class citizen rather than its container, like the endpoints in IP. The motivation behind NDN stems from the observation that the dominant usage of the Internet has shifted to data distribution and retrieval, driven by the sustained growth of emerging applications, e.g. IoT.

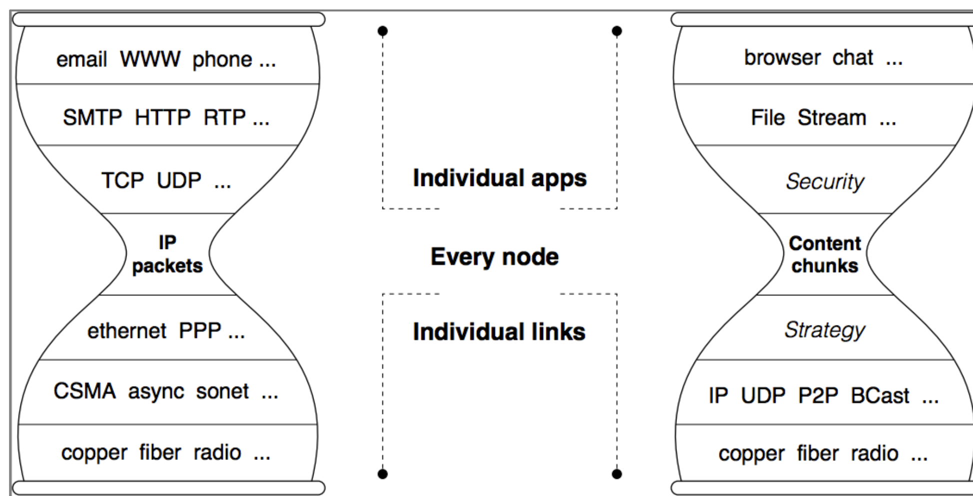


Figure 3 IP and NDN Architecture [29]

In NDN, everything is assigned a unique name, e.g. a video segment, a light switch, or a piece of deployable code. Each name is hierarchically structured. For instance, the name /tus/sri/room1/humidity/reading1 represents the first reading value of the humidity sensor located in room1 of SRI office in the TUS campus, where the slash symbol '/' separates two components in a name and is not considered part of the actual name itself. The naming scheme holds paramount importance in NDN as it is used for data retrieval and forwarding.

NDN defines two types of packets: Interest and Data. Both types of packets carry a name to identify a specific piece of data or content. The communication in NDN is

initiated by data consumers, who encapsulate the name of desired data/content into an Interest packet and send it to the network. NDN routers analyse the name in the Interest packet to forward it towards data producers. Eventually, the Interest reaches a node that has the matched data/content. The node embeds the requested data/content in a Data packet and returns it along the reverse path of the Interest. The specific naming of Interest is defined to support the functionalities of the proposed protocols in each contribution of this thesis.

NDN routers maintain three tables for the processing of Interest and the return of Data: Forwarding Information Base (FIB, to forward Interest to next hop), Pending Interest Table (PIT, to add waiting Interest and to return Data in the reverse route of Interest) and Content Store (CS, to save received data/content for a certain time). Figure 4 illustrates the procedure of packet processing in NDN and the following explanation presents the steps involved.

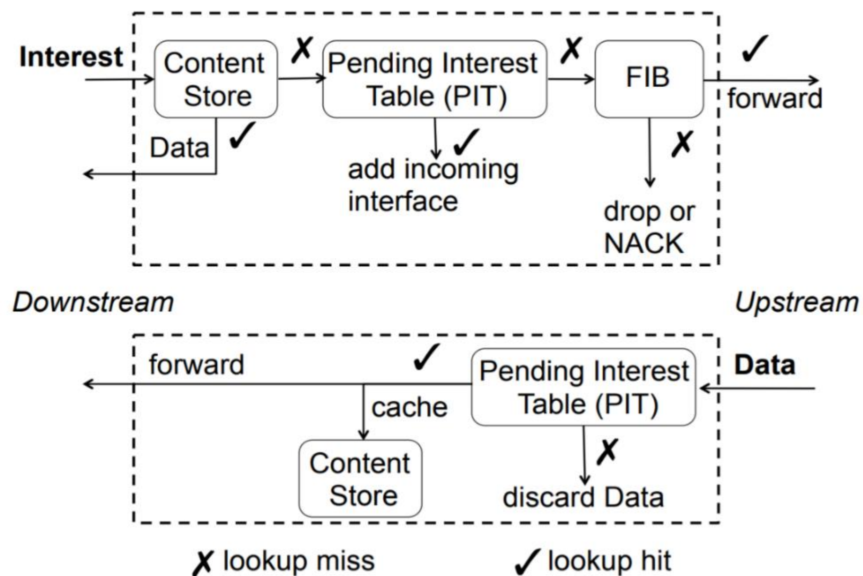


Figure 4 NDN Packet Processing [30]

Step 1: When a NDN router/node receives an Interest packet from downstream, it firstly checks its local CS to determine whether the required content is already

cached. If the Data is found in the CS, the node immediately returns the corresponding Data packet. Otherwise, the node searches this Interest in its local PIT.

Step 2: If the current Interest is not found in the local PIT, the node inserts a new record in its PIT, including the Interest name, incoming interface and other relevant information. In the case the Interest already exists in the PIT, the node maintains a list of interfaces that issued the same Interest and adds the new incoming interface to the list associated with the corresponding Interest.

Step 3: After creating a new PIT record, the node checks its FIB for further processing. If the search result for this Interest is positive, the node uses the routing information to forward the Interest accordingly. However, if no entry if found in the FIB, the Interest is dropped as the node has no knowledge of the possible next hops to retrieve the requested Data.

Step 4: Upon receiving the corresponding Data packet from upstream, the node finds the PIT records that match the name of the Data. If no record matches, the node discards the received Data. Otherwise, the node forwards the Data packet to downstream according to the matched PIT information, removes the corresponding PIT records and finally saves the Data in its CS to fulfil future requests.

2.2 MapReduce/HDFS

MapReduce is a programming model to process large data sets in a distributed fashion pioneered by Google [24]. Apache Hadoop [31] is an open-source implementation of the Google's MapReduce approach, which consists of a data storage module named Hadoop Distributed File System (HDFS) and a data processing module called Hadoop MapReduce.

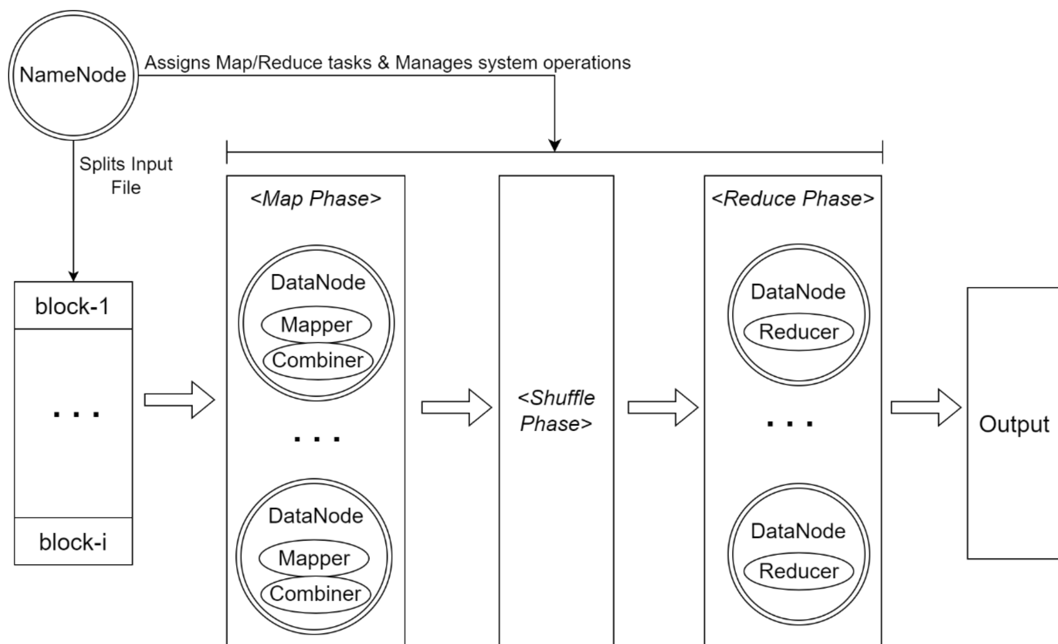


Figure 5 MapReduce/HDFS Framework

As illustrated in Figure 5, the HDFS architecture has a central NameNode serving as the controller and multiple DataNodes for executing MapReduce tasks. The controller role is a common concept in big data processing frameworks, for instance the JobManager in Flink and scheduler in Spark. The NameNode is responsible for:

- Splitting input file into several blocks and replicating each block on a subset of DataNodes for fault tolerance
- Assigning Map and Reduce tasks to idle DataNodes, continuously monitoring the status of each DataNode and scheduling replacements for failed nodes
- Handling all file system operations and maintaining essential system metadata

A single MapReduce job typically involves three phases: Map, Shuffle and Reduce. Each DataNode in the Map Phase runs user-defined mapper function on the assigned data block to generate key/value pairs, i.e. labelled as Mapper in Figure 5. A Combiner or combine-function is optional applied for partial combinations of the generated values by the same mapper with the same key, which lowers the transferred data size across the system. The Shuffle Phase sorts the output of mappers by keys,

and then sends the key/value pairs to reducers. This process ensures that the intermediate results with the same key are sent to the same reducer. Subsequently, DataNodes in the Reduce Phase execute user-defined reduce functions on the sorted data to produce the final output.

The canonical MapReduce use case is the word count job [32]. For instance, the input data is a text file with multiple lines of words. The map function is defined to generate a ("wordX", 1) pair for each word in each line. The shuffle stage in this example sorts the map output pairs by the same key ("wordX"). To count the number of appearances for each word in this text file, the reduce function is defined to sum up the value ("1") with the same key ("wordX"). The reduce function can be run by one or more nodes, depending on the job configuration.

Many types of tasks can benefit from the MapReduce processing model, such as data mining, distributed sort and so on. MR-CCN [33] aims to improve the performance of big data analytics in datacentre environments by implementing an ICN-based MapReduce framework. This approach enables the integration between computing and networking through routing with aggregation in ICN nodes. It also integrates caching and computing by utilizing ICN caching for intermediate and popular datasets to reduce network loads. MR-CCN proposes a novel naming scheme containing a routing section and a content section. The former section propagates Interest packets within the network, while the latter one describes the input data, processing functions and final result format. However, it is worth to mention that this work is specifically designed for the CamCube [34] data centre structure, which may limits its applicability to other network topologies, e.g. IoT.

Recent researches [35] argue that the MapReduce-style processing method (e.g. distributed machine learning [36]) is more suitable for modern applications. Many

large-scale industry systems collect and store data separately, it is more desirable to process the data in a distributed fashion to avoid the bottleneck of data transmission to a single server. Moreover, the consumed data size has increased sharply over the past decades, making it hard to deploy a single powerful server to process the massive and continuously growing data.

IoT is one of the systems with wide-spanning areas and vast amounts of data, making it an ideal application scenario for MapReduce. Beyond merely reducing network traffic, MapReduce can aid in the sharing of IoT infrastructure by accepting user-defined functions as input. This enables the same dataset executed by different processing functions serving specific information needs for various IoT applications. This thesis aims to apply the MapReduce processing concept to IoT scenarios to reap the benefits and while tailoring the MapReduce framework to accommodate the distinct constraints and prerequisites of IoT data processing.

Firstly, a significant consideration in IoT networks is the evaluation of data and node locality, which is often overlooked in traditional MapReduce deployments. Unlike conventional setups where the NameNode does not consider data locality when deploying Map/Reduce tasks on DataNodes and reallocating tasks to handle failures, it's critical in IoT networks to assess the proximity between a data source and a processing device to effectively reduce network traffic. Secondly, given IoT applications shifting from Cloud computing to edge computing, this thesis endeavours to implement the NameNode functionality in a distributed manner, effectively aligning with this paradigm shift. Thirdly, the original MapReduce design confines the combiner to operate solely on local datasets. However, given edge devices positioning between IoT data sources and sink nodes, their potential can be

harnessed to aggregate intermediate results from various processing nodes along the data transmission path to further minimize network traffic.

2.3 Collaborative Edge Computing in IoT

Edge Computing is proposed to complement Cloud Computing to deal with the high volume/velocity/variety of data produced by enormous number of IoT devices. It aims to reduce the response time for delay-sensitive IoT applications by placing computation and storage closer to data sources. Extensive research studies have explored the edge computing paradigm to boost IoT data processing, which can be categorized into two types: single-layer edge computing and hierarchical edge collaboration.

2.3.1 Single-layer Edge Computing

The single-layer edge computing refers that the data computation undertaken by a single edge server or by the cooperation of multiple edge nodes in a single layer. The focus is on optimising computation offloading in IoT scenarios, particularly in the context of the IoT-Edge-Cloud model. Some works consider a single edge device as the computation execution node, e.g., a scheme to execute CPU-intensive tasks on either the edge or the cloud to reduce network traffic and improve service quality [37], an algorithm to enable efficient handover between two fog servers for high mobility users to avoid service disruptions [38], a deep reinforcement learning-based approach to maximize the benefits of IoT devices when allocating edge or cloud resources for data processing and Blockchain mining tasks [39], and a recommendation mechanism for IoT devices in smart city to select a trustworthy edge service provider to decrease service latency [40].

Other research works explore the horizontal cooperation of multiple edge nodes to provide the computation service for IoT applications. To name a few, a case study in IIoT demonstrates reduced production order delivery time by utilizing self-organized task mechanism among multi-robots [41]. In the event of a node failure, robots negotiate and transfer the task of the failed node to an adjacent one, without central server scheduling. CoopEdge [42] designs a blockchain-based decentralized platform for edge servers to leverage each other's computing capabilities via peer offloading, ensuring timely task computation without overloading individual edge node. A multi-edge assisted query processing system is developed in [14] to minimize response latency and alleviate the workload of the cloud. A user's query is submitted to the cloud which generates a query execution plan and distributes to corresponding edge devices to process sensory data in a distributed manner.

The work in [15] integrates the FL technique with collaborative edge computing to allow multiple edge servers to perform partial model aggregation, which achieves faster training time with less energy consumption compared to with cloud-based FL approaches. Collaborative cross-edge analytics [43] focuses on data pre-processing phase before entering the model training and model inference phases of the artificial intelligence (AI). The raw data generated and stored at each edge site is labelled and transferred into trainable data samples before extracting by the AI model.

The contributions arising from this thesis fill different research gaps compared with works mentioned above. Firstly, this thesis designs a multiple-layer edge computing framework that considers the heterogeneous computation capabilities of edge devices. More powerful edge nodes handle complex processing tasks while less powerful ones simply assist in packet aggregation and forwarding. Secondly, this thesis develops a scheme to guarantee exactly once data processing in IoT edge

environments, which has not been extensively addressed in the related works discussed in this section.

2.3.2 Hierarchical Edge Collaboration

The hierarchical edge collaboration utilizes a multi-layer computation execution graph to coordinate several edge nodes working together to complete IoT tasks. Given the performance differences among heterogenous edge devices, e.g. base stations, electrical cars, laptops and mobile phones. A hierarchical structure is formed to organise edge nodes based on their capabilities in performing (sub) tasks.

A four-layer fog computing architecture is proposed for big data processing in smart cities in [11]. Due to its large-scale and geo-distribution characteristics, the layer-4 nodes in each region capture sensory data of various public infrastructure and forward to the layer-3 nodes, which aggregate data from multiple layer-4 nodes. Each layer-2 node connects with a cluster of layer-3 nodes and analyses spatial and temporal data for further decision-making. The top layer consists of a data centre that receives results from layer-2 and provides complex computing and long-term storage capabilities.

A three-grade edge computing paradigm for intelligent warehouse system that offers rapid detection and response of emergency cases is proposed in [9]. The grade-1 edge nodes are responsible for data collection and monitoring. The grade-2 nodes perform preliminary data processing and execute control commands received from higher grade edge nodes or the cloud. The edge nodes in grade-3 contributes to more complex data analysis for prediction and control.

LayerChain [10] designs a hierarchical storage architecture for large-scale IIoT data, employing a blockchain-based approach that involves cloud and multiple edge

nodes. To accommodate the varying computing power and storage space of edge devices, it defines three types (light, basic and full) of edge nodes according to the requirements of blockchain technology. Specifically, basic edge nodes have minimal storage space so that they only assist in forwarding messages, while light edge nodes have larger storage space and can verify generated blocks. Full edge nodes are the most powerful ones compared with the other two types. They can perform all the function of light edge nodes and act as miner nodes.

The mF2C project [44] introduces an IoT-Fog-Cloud continuum architecture that integrates a centralized cloud with various levels of fog computing. The system incorporates a mF2C agent on each node, providing management and control functionalities such as task orchestration, resource discovery in the edge, and service performance monitoring. The fog layer closest to the cloud can be deputed as the fog leader to aggregate processed results from lower-level fog nodes.

Although there are similarities between this thesis and the related works presented above, where sensing data is processed by intermediate edge nodes in multiple layers on the way transmitted to the cloud or users, the differences are as follows. In existing works, edge devices are pre-grouped into a specific level that remain constant for all computational services. In the proposed architecture of this thesis, the role of an edge node, i.e. processing-capable or not, changes with specific user requests. Moreover, this thesis delivers exactly once data processing guarantee in hierarchical edge computing, which is not the main research concern of the above works.

2.4 ICN based Edge Computing for IoT

Instead of patching the limitations of the IP architecture, research communities propose ICN [45] as a new clean-state architecture for emerging Internet applications.

Two research groups within the IETF cover topics related to this thesis. i.e. the Information Centric Networking Research group (icnrg) and the Decentralization of the Internet Research Group (dinrg). While the dinrg is currently focusing on discussing potential issues and threats of current Internet centralization, its documents are not discussed as related works in this thesis. On the other hand, documents published by the icnrg have explored the potential of ICN in various areas, including ICN-enable 5G next-generation core architecture [46], adaptive video streaming over ICN [47], ICN-based distributed architecture for microservices communication [48], ICN for efficient IoT [49] and more.

With its ability to name data and services, ICN offers many advantages for IoT compared to the current host-centric Internet. For example, improved data delivery performance in poor-quality links in vehicular networks thanks to ICN in-network caching feature [50], reliable data retrieval from multiple source nodes by a single request [51] and publish-subscribe scheme for IoT networks with intermittent connectivity [52] [53] utilizing ICN communication pattern, reduced IoT network load and task completion time through the implementation of ICN Quality of Service (QoS) management protocol [54] and an ICN framework designed for the delay-tolerant communication between long range wireless networks and the Internet [55]. However, challenges emerge when adopting ICN to support IoT applications given the heterogeneity of devices, data processing and content distribution models [49]. The original design of ICN supports in-network data forwarding and caching but lacks in-network processing functionality. This thesis aims to extend ICN to enable data/content processing, with a special focus on IoT scenarios.

Named Function Networking (NFN) [56] is the pioneer project to enhance the network's functionality to process data and cache previous computed results in the

CS of network devices. It proposes to name the data processing logic so that the network can work as a computational machine when both data and function pieces are available. The Interest name is expressed as Lambda Calculus to drive the whole network using find-or-execute rules:

- Find: the network returns the required result if it exists within the network.
- Execute: if the required content is stored within the network, the network searches both data and functions for the task and then executes the task to obtain processed result.

The NFN team states that the locality-of-execution is discovered and decided by the network according to the processing policy or resource availability, but no detailed solutions are given. Moreover, NFN scheme is not specifically designed for the usage in IoT scenarios.

RICE [57] augments the capabilities of NFN to enable long-running computations within the network, which decouples the invocation of a computation/function from the retrieval of computed results. The "Thunk" name is proposed to identify the specific node for content fetching assuming that the computation is handled by a single sever. Compute First Networking [58] leverages RICE and the conflict-free replicated data structure to implement a distributed computing framework. It defines the "program" as a set of computations requested by a user. Each computation in the program can be deployed on a worker node located nearest to the input data with the biggest size. This approach maintains the same assumption as RICE that all inputting data needs to be transferred to a single node for computation.

NDN-Q [59] is a distributed query mechanism for data collection in V2X (Vehicle-to-Vehicle and Vehicle-to-Infrastructure) scenarios based on NDN. The proposed naming scheme embeds data selection, filtering and aggregation rules for

query processing. It allows the nodes on the routing path to process data in a database mode, with some of them returning the result only if it meets the user's requirements. The name of aggregation/reduce logic is encapsulated in the Interest name and the processing nodes need to request the logic if it's not available locally. At the end, the processed result is sent back to the query source. As mentioned in their paper, each node in NDN-Q needs to maintain a key-value Database (DB) to facilitate future processing. This thesis argues that DB style processing cannot offer the flexibility to take and run user-defined functions directly. Moreover, current NDN-Q only implements the data processing at one level of intermediate nodes. The cooperation among processing nodes is not considered in their paper.

Keyword-based ICN-IoT [60] combines the scalability of NDN hierarchical naming scheme with the flexibility of keywords as a hybrid routing scheme to ease data sharing in same IoT domain. It includes function tag as part of the naming scheme, which describes the function that should process the desired IoT dataset. The assumption in this paper is that all edge nodes are capable of data processing and then the final execution placement depends on the trade-off between the data transmission and computing resource cost.

NFaaS [61] assumes functions saved as Virtual Machines (VMs) in the form of unikernels. It dynamically migrates the function execution among edge nodes and the cloud depending on nodes' storage capacity, computation capability and specific requirements of services, i.e. the delay-sensitive or bandwidth-hungry type defined in the paper.

In IoT-NCN [62], the edge node that acts as a data processing executor is decided and manipulated by the implemented "ExecAvailability" boolean field to the NDN PIT entry. The default executor is the branch node of multiple requested data sources,

which sets the “ExecAvailability” to true to avoid duplicated processing at other nodes. If the branch node fails to execute, it generates a Data packet with error code “no-computation” and returns. On the reverse path of the Interest, the first neighbour node that receives the Data packet becomes the executor.

To optimize IoT service deployment and execution by utilising ICN features, a content-centric networking (CCN) based protocol [63] is designed to help service discovery among mobile edge computing (MEC) nodes. It chooses the best MEC to guarantee the QoS and avoids deploying the same service duplicated on neighbouring MECs. ICedge [64] proposes a self-learning scheme to dynamically discover multiple network paths to all computing nodes that offer the same service. It reuses previous-computed results among users to minimize execution/completion time by parsing the NDN name of each request and dispatching requests with similar names to the same computing node. Docker-based services are dynamically deployed on edge servers based on service popularity using ICN-featured forwarding strategy [65]. Furthermore, the VM-based service optimal deployment is evaluated through the service gain defined as the processed data amount per CPU cycle [66].

The main differences between this thesis and the above works are: (1) employing the multi-layer collaborative edge computing paradigm to execute IoT data processing as computation-intensive tasks, e.g. image processing and speech recognition, are proven to benefit from the synergy of multiple edge devices than offloading to a single edge server [67], and (2) guaranteeing the correctness of the computed results in a distributed manner for IoT applications with the exactly once data computation requirement.

2.5 Exactly Once Data Processing

In the realm of data processing or delivery, three distinct semantics have been defined: at-least-once, at-most-once, and exactly once. Amongst these semantics, achieving exactly once semantics has the strictest requirement compared with the other two. Notably, Google's research [68] emphasizes that exactly-once processing is a requirement for many of their revenue-processing customers. Moreover, duplicated record deliveries could cause spurious spikes for Google's hot trends service. Other applications include service agreements for stock traders to ensure the visibility of every trade event without duplication [69], fault-tolerant Write-Ahead-Log entries for providing transaction atomicity and durability [70] and non-duplicated aggregation of sub-models deployed on edge servers within a hierarchical FL system to maintain the correctness of trained model [15].

2.5.1 Datacentre-oriented Solutions

Traditional big data processing systems, e.g. Apache Flink [18] and Apache Spark [28], have developed mature solutions to support exactly once operations. Both frameworks employ a checkpoint scheme as the foundation to achieve the exactly once semantics. A checkpoint consists of a snapshot of the state of each operator, which is typically saved to a durable storage system, e.g. HDFS [71]. The system can recover from failures by reverting each operator to the previous state saved in a snapshot and reprocessing the input data from the respective checkpoint barrier. This approach assumes that the logical job graph remains the same when restarting from a successfully saved checkpoint. With the powerful servers close to each other in traditional data centre scenarios, the logical job graph is independent of the physical job graph, allowing the restoration of the logical graph onto different devices if

needed. For example, the job manager in Flink may allocate the task on a different work node if the previous one fails.

However, this thesis argues that the checkpoint-based solution is challenging to be applied in IoT scenarios. Firstly, the logical job graph is tightly coupled to the physical topology in IoT edge environment. Data processing task(s) should not be arbitrarily placed on an edge device to replace the previous failed one as it needs to evaluate the benefits of transmitting data versus processing/aggregating the data. Secondly, high-performance durable storage systems for saving periodic snapshots are not widely available in edge computing environments, such as the HDFS.

2.5.2 IoT-related Solutions

Despite fruitful research studies on IoT edge computing, few findings have specialised in guaranteeing exactly once data delivery and processing.

The solution proposed in [72] improves the message queue systems, e.g. Kafka [19] and RabbitMQ [73], to ensure exactly once processing through a consumer side protocol. All messages are stored in a shared DB and a state transition graph is introduced on each message to control access and operations. IoTEF [16] is a federated edge-cloud architecture based on Docker containers, which deploys one Kafka cluster in the edge and one in the cloud. It uses Kafka to buffer data streams in case of network failures and ensure exactly once data semantics within a cluster.

Initial attempts have explored the checkpoint-based approach to save the state of an IoT task into Docker images. Researchers in [20] focus on deploying the Function-as-a-Service (FaaS) model on IoT devices. They checkpoint the states of long-running functions and save them as containers, which enables function execution migration from one device to another, considering the resource constraints of a single

IoT device to finish a computation-intensive task. Similarly, the state of data processing is checkpointed as an image in [21] to facilitate information transfer between different tasks running on IoT devices with limited Random Access Memory (RAM). However, these works only concern the task execution on a single edge device.

2.6 Distributed Consensus Protocol

To achieve exactly once computation in IoT collaborative edge environment, it is essential to obtain a consensus on the data computation plan amongst the edge devices. In distributed computing systems, ensuring consistency and reliability in data transactions across multiple nodes is of utmost importance. The two-phase commit protocol [75] [76] is a well-known algorithm in distributed systems to coordinate all parties to agree or abort an action.

The two phases include the commit-request phase and the commit phase. It designates a coordinator node, and the rest of nodes are participants. The main procedure of the protocol is summarised as follows. In the commit-request phase, the coordinator sends a message to all the participants to notify them preparing for the commit operation. Each participant votes yes or no according to its state. The commit phase starts when the coordinator receives all participants' replies. If all participants vote yes, the coordinator sends a commit message to all participants. If any participant replies no, the coordinator sends a rollback message to all participants to abort the operation.

In Apache Flink [77], two-phase commit protocol is utilized to coordinate distributed checkpoints so as to provide exactly-once semantics. The starting of a checkpoint represents the commit-request phase. Every operator in the data

processing pipeline, spanning from the data source to the sink, takes a snapshot of its state and sends a response message of either "commit" or "abort" to the Flink JobManager. Once receiving messages from all operators, the Flink JobManager checks the responses. If at least one operator fails in the commit-request phases, all other operators are aborted and the system reverts to the previous completed checkpoint. Conversely, if all operators reply to commit, the checkpoint is considered successfully completed. The Flink JobManager issues a checkpoint-completed callback for each operator, representing the commit phase.

To achieve exactly once data computation, this thesis is inspired by the two-phase commit protocol, which defines a Job Execute Phase (chapter 5.2.2) for disseminating and executing jobs (i.e. the commit-request phase) and a Job State Commit Phase (chapter 5.2.3) to commit the job completion state only if all nodes returning computed job results correctly (i.e. the commit phase).

3 Functional Architecture and its ICN-based Implementation for MapReduce Jobs

This chapter describes the first contribution arising from this thesis. It proposes the functional architecture of IoT edge computing, which emphasises the paramount functional units to deploy and execute IoT data processing in the edge environment. Followed by that, the ICN-based implementation of the proposed architecture to run MapReduce jobs is presented with two purposes: (1) to meet the information-oriented nature of IoT applications by utilizing the novel ICN rather than the traditional IP which requires to establish end-to-end connections before data retrieval, and (2) to allow users to flexibly express their processing logic, by leveraging the MapReduce model with user-defined functions as input. Experimental studies have proven the feasibility of the ICN-based IoT edge computing framework and its effectiveness in decreasing network traffic.

3.1 Functional Architecture Overview

The heterogeneous nature of IoT network devices results in significant variations in performance capabilities among different devices. Therefore, it is necessary to assign data processing tasks to appropriate devices. This thesis proposes a functional architecture to outlines the indispensable components of an IoT edge computing framework. The architecture consists of three software components and their duties are illustrated in Figure 6.

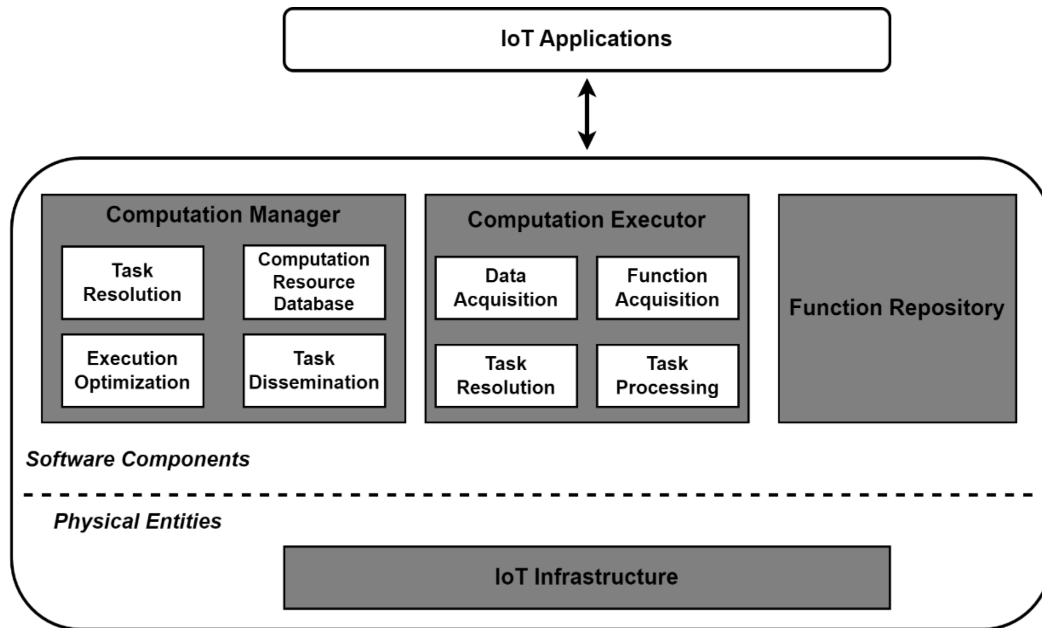


Figure 6 Functional Architecture

(1) Function Repository: It stores all data processing functions provided by edge devices, which can be managed by the Computation Manager or other edge nodes for easy access and availability of functions for processing data.

(2) Computation Executor: Computation Executors are responsible for executing tasks in the proposed architecture. The workflow of Computation Executors includes *Task Resolution* to parse the required data and function of the received task. Subsequently, Computation Executors initiate *Data Acquisition* by sending requests to corresponding data sources and *Function Acquisition* by retrieving the required function from the Function Repository. Finally, *Task Processing* is carried out once the data and function are obtained.

(3) Computation Manager: The Computation Manager bridges users' requests and proper Computation Executor(s), considering that users outside of the IoT network may not possess detailed information about the capabilities of numerous devices. The Computation Manager encompasses several functional units that optimize the task

execution and assignment, which can be centrally controlled by the Computation Manager or distributed within the network. These functional units include:

- *Task Resolution*: This unit not only determines whether the request can be processed but also to select the suitable Computation Executor(s) for this request.
- *Computation Resource Database*: The database stores and regularly updates information about the computing resources within the edge network, e.g. available Computation Executors and network topology, which serves as a foundation for the Computation Manager to make decision of assigning tasks.
- *Execution Optimization*: Possible execution plans are made after checking available computation resources in the database. In this thesis, the job execution plan prioritizes selecting the Computation Executor that is closest to the data source(s), considering factors such as proximity and network latency.
- *Task Dissemination:* This unit allocates the user's request to the designated Computation Executor(s) according to the generated task execution plan, ensuring efficient task assignment and execution.

3.2 Illustration of Workflow in the Functional Architecture

Figure 7 illustrates the working procedure of the proposed components in the functional architecture. The following steps provide the explanation in detail.

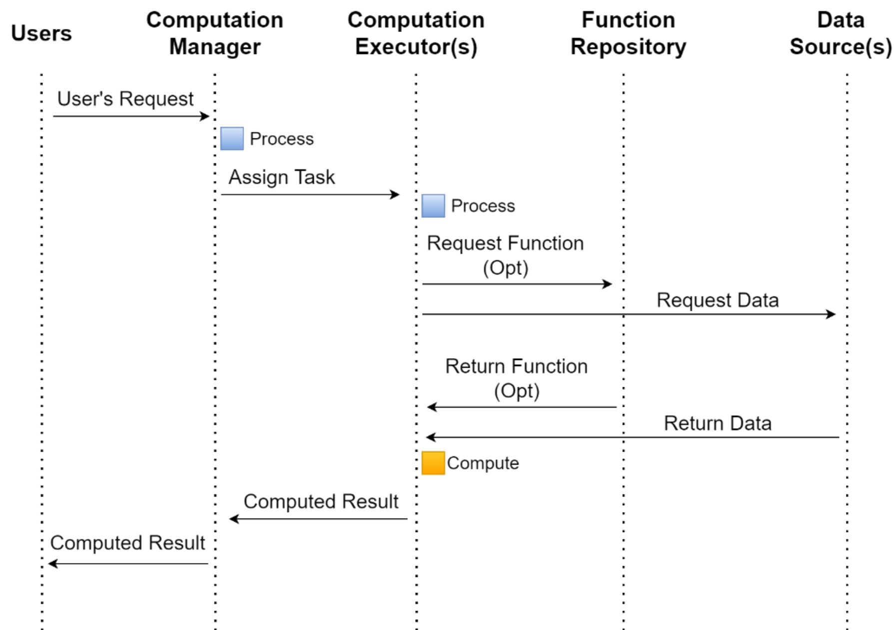


Figure 7 Workflow Illustration

Step-1: a user sends a request to the IoT network.

Step-2: the Computation Manager receives the user's request and begins processing. It checks the availability of edge nodes in the computation resource database and selects the optimised Computation Executor to which the task will be assigned.

Step-3: the selected Computation Executor receives the request and resolves it to obtain the required data and function. It is important to note that requesting the function is optional because the function could be available from local cache if it was used in previous task processing. Similarly, if the data does not update in real time and the same data is cached locally, the Computation Executor does not need to request it. Otherwise, the Computation Executor needs to request the current sensing data for each received task.

Step-4: the Function Repository returns the corresponding executable code for the received request, while the matched data sources provide the desired data.

Step-5: the Computation Executor performs the computation when both function and data are obtained. It returns computed result to the Computation Manager once the processing is done.

Step-6: Finally, the Computation Manager returns the processed result to the user.

3.3 ICN-based Implementation for MapReduce Jobs

It is a fact that many IoT systems are deployed widely and densely in geographical areas. These distributed datasets can be processed partially without impacting others, utilizing large-scale data processing models like the MapReduce programming framework. To align with the information-oriented nature of IoT applications, this thesis implements the proposed functional architecture upon NDN to execute MapReduce-type jobs.

3.3.1 Functional Units Instantiation

- Computation Executor: Mapper and Reducer

This thesis assumes that the selection of processing-capable edge nodes has already been done. There are two types of Computation Executors defined: mapper nodes directly connect with sensors, which processes sensing data and returns processed result to upstream nodes. Reducers, on the other hand, receive and process data from downstream neighbours, including directly connected mappers and/or child reducers.

- Task Dissemination

As explained before, the task deployment procedure relies on the computation resources that are centrally stored and manipulated by the Computation Manager (i.e. utilising the Computation Resource Database and Execution Optimization modules

of the Computation Manager in the proposed functional architecture). This procedure is improved by the development of a protocol for building a computation job tree to autonomously resolve and disseminate jobs among edge nodes. The job tree is unique for each user and optimized to select the Computation Executors on the shortest path between the user and required data source.

- Task Processing

A NDN naming scheme is designed to express users' requests, incorporating user-defined map and reduce functions, as well as required datasets in a single Interest. The Function Acquisition step, one of the responsibilities of the Computation Executors, can be achieved by parsing the received Interest. Computation Executors proceed with Task Resolution to obtain the NDN name(s) of required dataset and sends Data Acquisition Interest(s) to the corresponding data sources. A computational job execution protocol is devised to guide Computation Executors in Task Processing and returning computed results.

3.3.2 Computational Job Tree Construction

Since tree-based data aggregation is a widely used approach in Wireless Sensor Networks (WSN) [78] [79], this thesis borrows the idea to implementing the execution of MapReduce jobs in IoT edge networks. Thus, it is essential to avoid network connection with loops. Real world IoT networks exhibit various types of topologies, such as ring, star and tree structures. In the proposed architecture for executing the MapReduce job, a tree topology is adopted with the current user as the root and mappers as the leaves so that the intermediate results can be aggregated effectively.

The construction of the computational job tree is launched if there is no information of computational neighbours existing on the reducer nodes. The tree is built using the NDN shortest path routing algorithm. This choice simplifies the need to develop a specialised routing protocol to support the proposed edge computing framework. This thesis acknowledges that additional parameters should be considered to optimise the construction of the computational job tree, which will be part of future work.

To construct the computational job tree, the reducers and mappers are designed to query each other about the routing path to the root of the tree (i.e., the current user who has issued the task). In details, every node asks its neighbour "Am I your upstream node to the current user". With the support of NDN routing, each node knows the routing information from itself to any specific node. However, nodes are not aware the routing information saved in the other nodes, and consequently all nodes need to communicate with their neighbours to obtain the information.

For clarity, the user is regarded as the root reducer. This thesis defines the BuildJobTree Interest for this procedure, and it is written as command (a):

/NeighborName/BuildJobTree/JobTreeID (a)

Where: (1) /NeighborName is the name of a neighbor of the current node. (2) /BuildJobTree is the identifier to trigger the procedure of computational job tree construction. (3) /JobTreeID is unique for each tree and combines the name of the root node and a random number.

Initially, the root reducer sends the BuildJobTree Interest (a) to its neighbours. Other reducers need to handle two events in the computational job tree construction procedure: receiving the BuildJobTree Interest and sending out the BuildJobTree Interest.

Upon receiving a BuildJobTree Interest from a neighbour (either the root reducer or other reducers), a reducer decomposes it to obtain the name of the root reducer. Then, this node checks its own NDN routing table and get the neighbour (called SelectedUpstream for clarity) on the shortest path to the root reducer. If the received BuildJobTree Interest is not sent by its SelectedUpstream, the node replies "nope" immediately, indicating that it will not use that path to return data for the current root reducer. Otherwise, the node postpones the reply to its SelectedUpstream and continues discovering its neighbours. The subsequent scenarios depend on the outcome of this discovery process:

- All its neighbours reply "nope" to this node, it needs to reply "nope" to its SelectedUpstream. Because this node has no child neighbours joining the job tree of the current root node.
- At least one of its children replies "yes", this node replies "yes" to its SelectedUpstream to join the current computational job tree.

If a reducer needs to continue exploring its computation neighbours, it modifies the original BuildJobTree Interest by inserting its own name. The re-organized Interest is represented as command (b). This step ensures downstream nodes know which node sent this Interest if multiple neighbours send the same BuildJobTree (a).

/NeighborName/BuildJobTree/JobTreeID/UpstreamNodeName (b)

Mappers do not forward the Interest. They receive and reply to the BuildJobTree Interests during the computational job tree construction. A mapper may receive multiple BuildJobTree Interests from reducers. It chooses only one reducer as the SelectedUpstream by replying "yes" and meanwhile responses "nope" to others. The computational job tree construction completes when all neighbours of the root reducer provide feedback.

3.3.3 MapReduce-Job Execution

3.3.3.1 Naming Scheme

To enable users to specify the desired data and define how it should be processed, this thesis proposes a NDN naming scheme to include data and MapReduce functions within an Interest. More importantly, the naming scheme supports task dissemination and data/function acquisition.

The defined functional Interest consists of four parts, organized as command (c):

/NeighborName-/map/f1(x->(k,v))-/reduce/f2((z1,z2)->(z3))-/contentFilter (c)

Where, (1) /NeighborName: matches the name published by neighbours to ensure that the specific node receives the Interest. (2) /map/f1(x->(k,v)): “f1” is the user-defined map function that should be applied to each input data. “x” is the input data and it can be in any format according to specific use cases. For example, in a MapReduce wordCount task, the input data is a text file and every word in this file is regarded as the key. In a MapReduce statistics task, e.g. to count the periodical updates, the sensor readings are captured in (timestamp, sensory-value) format. The input data in this case is already in a (key, value) format. The output (or processed data) of a mapper is always a (k, v) pair. (3) /reduce/f2((z1,z2)->(z3)): “f2” is the user-defined reduce function to process received data from mappers and/or child reducers. The reduce function operates on a list of values with the same key. (4) /contentFilter: is the desired content name or a defined filter on content.

The functional Interest is constructed hierarchically, with each part starting with a slash and different parts separated by a short dash. For instance, in smart city scenarios, if an officer wants to know the number of noisy sensors with the same monitoring level (L), the request can be expressed as below:

/NeighborName-/map/(L)->(L,1)-/reduce/(L1, L2)->(L1+L2)- /allNoiseSensor (d)

3.3.3.2 Job Execution Procedure

The proposed architecture can disseminate and execute MapReduce-type jobs when the computational job tree is ready. The workflow is shown in Figure 8, which mainly consists of three steps, i.e. job decomposition (D1), job deployment (D2) and job distribution (D3).

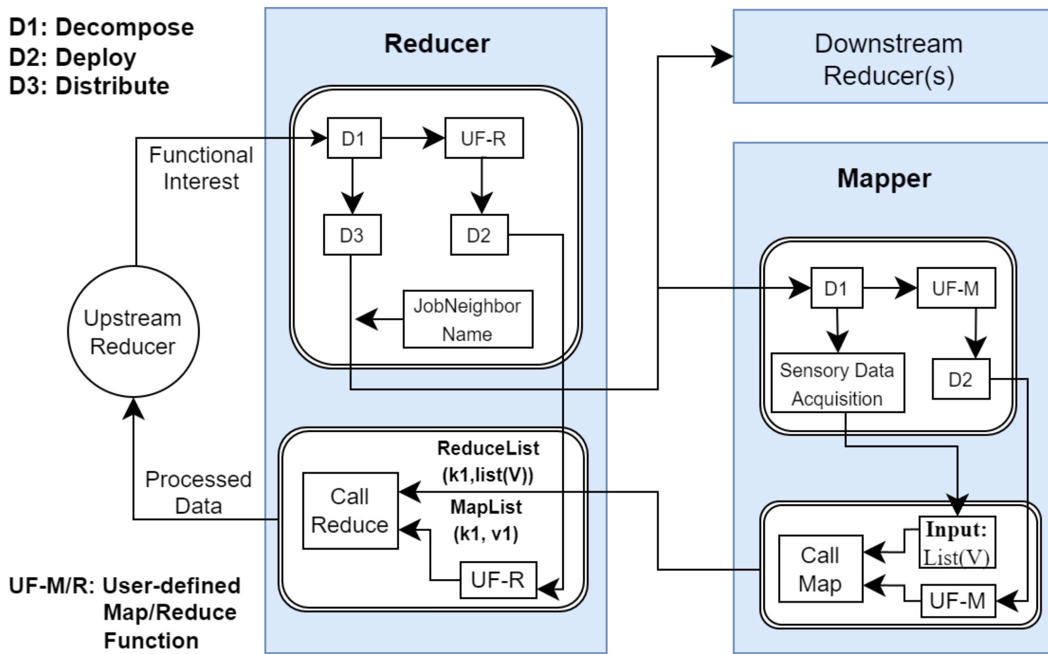


Figure 8 Workflow of NDN-based MapReduce Job Execution

For reducers, when they receive a functional Interest, they decompose (D1) it to extract the user-defined reduce function (UF-R) and deploy (D2) it, waiting for the data returned by its downstream neighbours. At the same time, each reducer continues to distribute (D3) the functional Interest to its neighbours (downstream reducers and/or mappers) on the computational job tree. The process is similar for the mappers, except that mappers decompose the user-defined map function (UF-M). In addition, as the mappers are directly connected with sensors, they request all sensory data instead of distributing functions to sensors.

For task execution, the mappers initially run the UF-M on multiple sensory data. Each mapper produces a list of (key, value) pairs and then returns the pairs to its SelectedUpstream reducer. When a reducer receives the processed data from all its downstream neighbours, it executes the UF-R to all the datasets. The final processed result is the combination of all sub-results produced by the reducers on the computational job tree. Its format is defined as command (e) and encapsulated into the Data packet for return.

k1,v1-/k2,v2-/…-/k*,v* (e)

The computational jobs defined in this thesis can be used to aggregate data into a smaller size or filter noisy data as soon as possible. It is up to the specific functions provided by the users. As a result, users only need to describe what they want without worrying about how and where the processing is performed.

3.4 Evaluation

The proposed design is implemented on ndnSIM [80], which is a simulator specially designed for NDN. To verify the design, a network generator BRITE [81] is employed to generate network topologies.

3.4.1 Test Design

Three types of topologies are depicted in Figure 9. Each topology comprises one user node and nine reducers. The distinguishing factor among the topologies is the number of mappers: 50, 100, and 150 for the topologies denoted as 60-Nodes, 110-Nodes, and 160-Nodes, respectively. For each fixed number of nodes, this thesis uses BRITE to generate ten topologies, captures network traffic on each topology and calculate the average value.

There are three types of data transmission speed, each characterised by a combination of bandwidth and delay. The first type has a speed of 100 Mbits per second with a 25-millisecond delay between the user network and the IoT network utilizing Ethernet standards. The second type operates at a speed of 250 kbits per second with a 10-millisecond delay between mappers and reducers based on the Zigbee protocol [82]. The third type has a speed of 54 Mbits per second with a 1-millisecond delay between two reducers using IEEE 802.11 parameters [83].

To conduct a comparison study, referred to as "Request-Directly-benchmark" approach, the user node directly sends NDN Interests to request sensory data for each network topology size. Both the proposed design and the Request-Directly-benchmark solution run for 100 seconds, with the user sending one Interest per second. Every mapper generates a data sample for each received Interest.

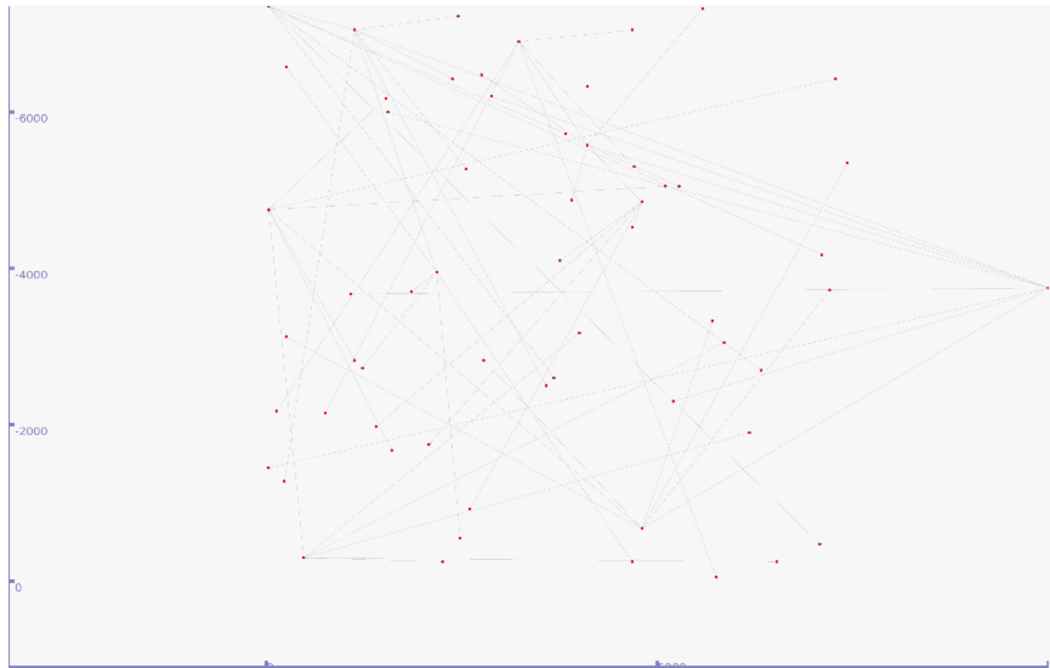
To test the protocol of the computational job tree construction, it should ensure that every reducer participant in the MapReduce job execution because they are all configured to connect with downstream mappers. It is also necessary to verify whether each reducer processes the correct number of mapper data (1 reducer for 5/10/15 mappers in the three topologies respectively). Furthermore, as the network topology generated by BRITE contains loops, the proposed protocol should decompose the loops into a tree during the construction of the computational job tree.

To validate the job execution procedure, a MapReduce job is assigned. The functional Interest during tests is expressed as command (f). Both map and reduce functions are expressed as mathematical expression to examine whether mappers and reducers can execute user-defined functions correctly. The normal Interest used in the comparison study is expressed as command (g). The user node sends out all Interests and waits for the corresponding Data. Since the comparison study involves

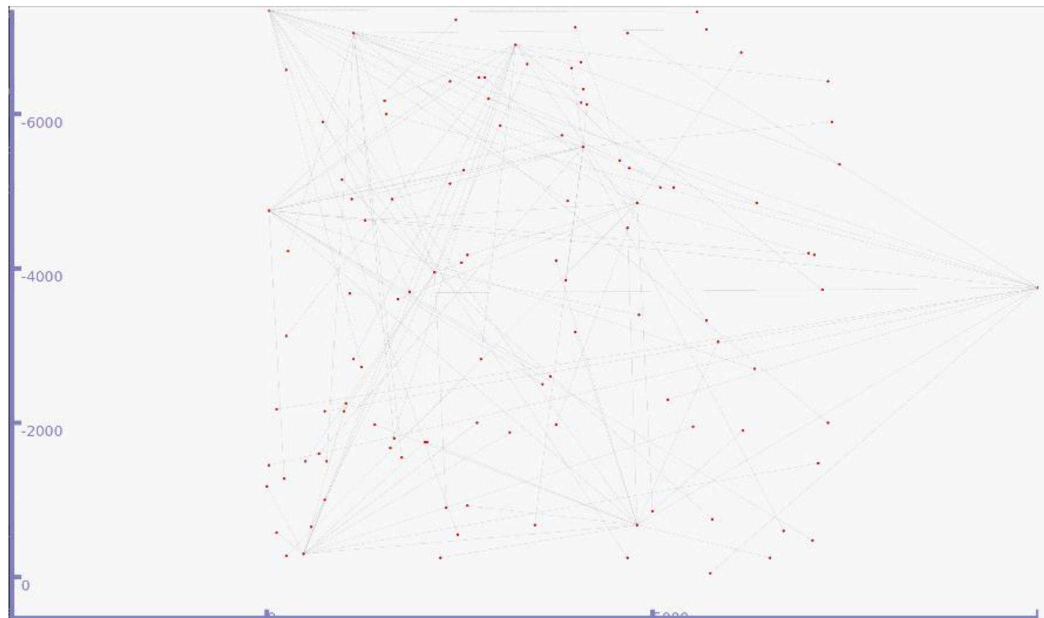
no computation, the task is considered complete when the user receives all returned Data.

/NeighbourName-/map/(k,v)->(k+1,v*2)-/reduce(n1,n2)->(n1+n2)-/allMapper (f)

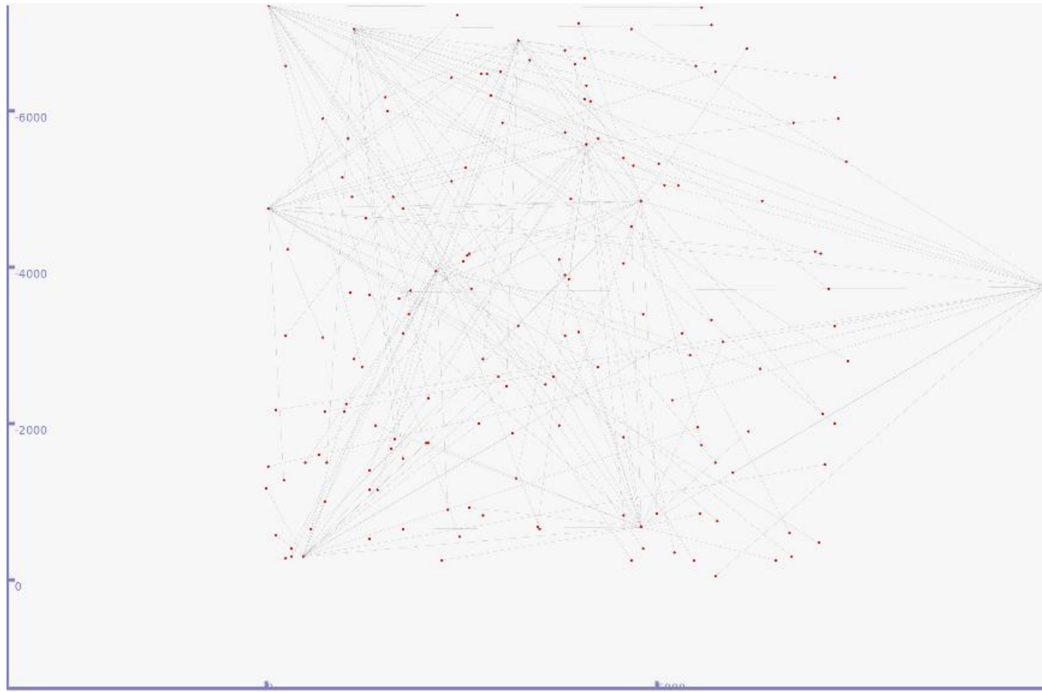
/mapperName (g)



(a) Topology of 60-Nodes



(b) Topology of 110-Nodes



(c) Topology of 160-Nodes

Figure 9 Network Topology for Tests

3.4.2 Test Results and Analysis

This section presents the test results and analysis. It focuses on examining the computational job tree topology, calculating network traffic and recording the number of Interests sent by the user.

Figure 9 illustrates the original network connections with loops. Although reducers and mappers connect with each other on the job tree, only the responsible reducer nodes forward user's Interest to the corresponding mappers. As a result, nodes that are not on the computational job tree do not experience any traffic related to the current job on their edges.

Figure 10 presents a comparison of the generated network traffic between using the proposed design and the Request-Directly-benchmark approach to complete the same job. The comparison reveals both solutions experience an increase in network traffic along with the raising number of mappers. The proposed design could keep

decreasing approximately 40%~46% network load compared with the benchmark approach in all three test cases. For instance, in the topology with 160 nodes, the proposed design generates roughly 3385 kilobytes of traffic, while the Request-Directly-benchmark approach produces 6261 kilobytes of traffic. Based on the test results, it can be inferred that the comparison study generates heavier IoT network traffic than the proposed design to complete the same job.

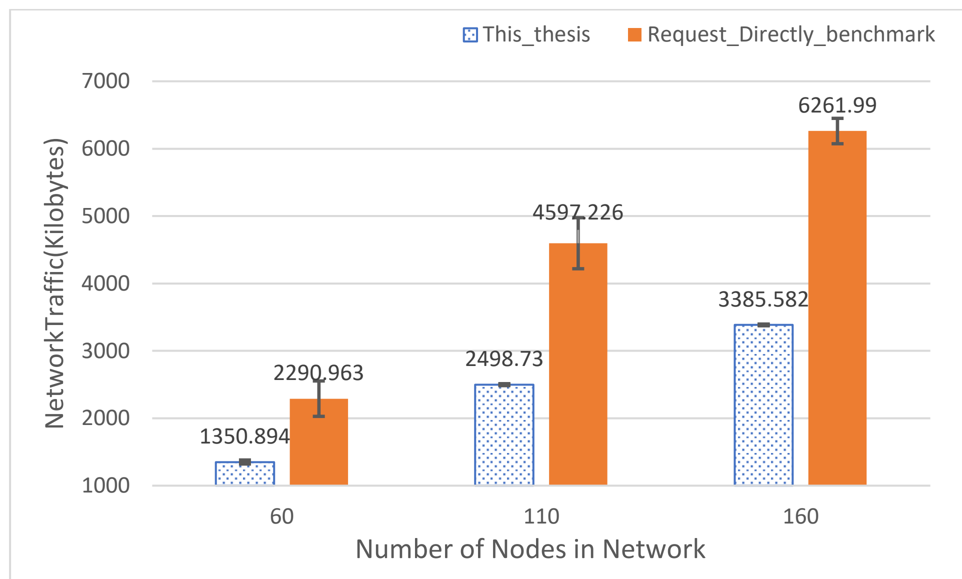


Figure 10 Network Traffic Comparison

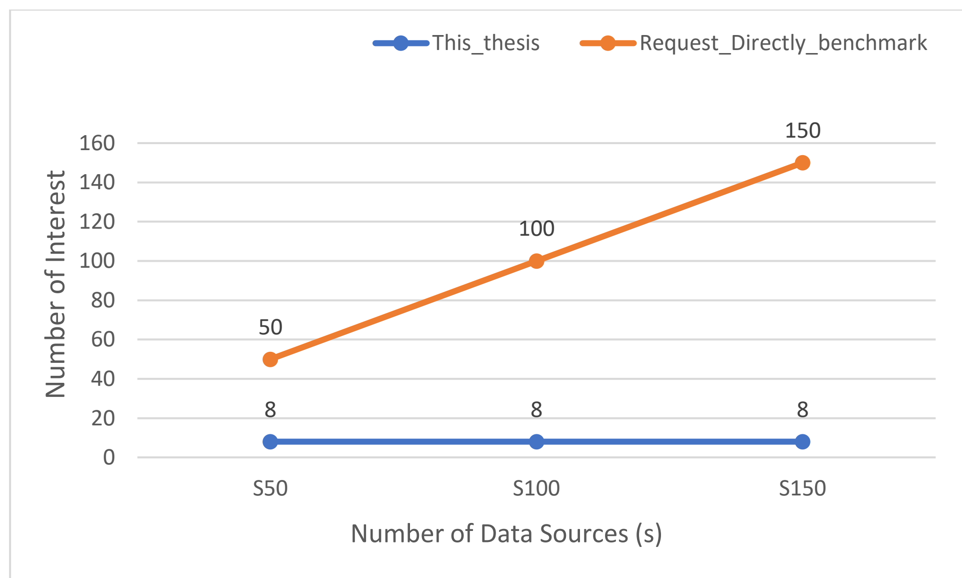


Figure 11 Number of Interest at User Node

Figure 11 illustrates the number of Interests sent by the user in three test topologies. The user node has to send a separate Interest to each data source in order to obtain all sensory data, reaching a maximum of 150 requests sent in the largest test topology. Moreover, one Interest packet is matched by one Data in NDN. Therefore, the user receives the same amount of Data packets as the number of Interest sent. The difference in traffic between the proposed design and the comparison study becomes more pronounced as the number of mappers involved increases. In the proposed design, the user always sends Interests to its job neighbours on the computational job tree regardless of the number of mappers involved in different jobs. To be noted, the average number of job neighbours for the user node is set to 8, varying between 6 and 9 across the ten generated topologies. It ensures efficient data retrieval without overwhelming the user node with excessive traffic. On the other hand, in the comparison study, the number of Interests sent by the user is equal to the number of mappers or data sources in the network. This implies that to obtain data from a wide area involving numerous sensors, the user would need to send a large volume of messages. This approach is not practical as it results in the user node being overloaded with traffic while running a single task.

3.5 Summary

The combination of edge computing and IoT is promising to deal with the massive amount of raw data produced and exchanged by IoT devices. However, building an effective IoT edge computing framework presents its own set of challenges. On the first research stage, this thesis explores the potential of heterogenous edge nodes that are divided into different roles to undertake appropriate processing tasks, i.e. Computation Manager, Computation Executor and Function Repository. A functional

architecture is proposed to define the responsibilities of each role to empower in-network data processing at IoT edge.

Given the powerful MapReduce programming model for distributed big data processing, this thesis implements the proposed functional architecture to execute MapReduce-type jobs. Recognizing that IoT applications are typically data/information-oriented, the novel data-centric ICN architecture, specifically NDN, is chosen as the underlying network architecture, which offers superior network support compared to the current end-to-end IP communication model.

The subsequent research efforts devote to the implementation of the ICN-based edge computing framework for the execution of MapReduce-type jobs. Firstly, a naming scheme is investigated to express the required data and processing logic using ICN communication. Secondly, a scheme is devised to establish the network topology for computation, ensuring the efficient dissemination and accuracy of jobs. Thirdly, a job execution protocol is developed to facilitate job dissemination, function execution, and the return of processed data for job execution. Finally, an experimental environment is set up to validate the research design, and the test results demonstrate a reduction in network traffic compared to the Request-Directly-benchmark approach.

4 Protocol for Multiple MapReduce Jobs Execution with Resource Constraints

This chapter presents the second contribution arising from this thesis. It targets at the job execution graph/tree generated by the computational job tree construction protocol in **Contribution I**, which assumes that all edge nodes have the computing resources and power. As IoT connects various kinds of devices, their processing capabilities cannot be the same, some are capable of data computation while others are not. For this reason, **Contribution II** proposes an enhancement to the framework by considering the resource constraints of edge nodes, which may be caused by the heterogeneity of devices and/or the dynamic allocation of resources to jobs. Moreover, a job maintenance scheme is devised to enable multiple IoT jobs simultaneously serviced by the proposed framework. A testbed is developed on ndnSIM to verify the feasibility of the proposed solution.

4.1 Concept Overview

This thesis concentrates on the job type that requires cooperation of multiple edge nodes to process data. Every computing job has specific requirements not only on data and processing logic but also on the capability of computing devices. For example, an edge device may have been overloaded by other tasks or the hardware capacity is limited to execute the task. Describing computing resources and selecting appropriate devices to meet different job requirements is another research topic that

is not the main concern of this thesis. For simplicity, the proposed solution divides IoT edge nodes into two types: processing-capable (act as a mapper or reducer) and forwarding-only (called forwarder). Processing-capable nodes have sufficient computing resources to meet a job's requirement, while forwarding-only nodes lack the necessary resources to process data for the current job. Both types of nodes participate in the job tree construction process, but only processing-capable nodes parse and execute the assigned map or reduce functions on the required data.

Each user's request is defined as a job in the proposed design, which contains map and reduce tasks and desired datasets. The proposed framework supports the coexistence of multiple jobs, with the assumption that a unique computational job tree is built for each job in sequence. Currently, the NDN routing protocol is utilized to construct a shortest path tree for each job. However, it is aware that more algorithms (e.g. minimum spanning tree or Steiner tree [84]) may be necessary to build trees to meet specific job requirements or optimize IoT edge resources, which will be a part of future work.

Figure 12 provides an example to illustrate the proposed solution assigns different tasks to IoT nodes and organises them working together to accomplish different jobs. It is assumed in this thesis that the procedure of matching computational resource needs with available computing devices has already been performed. The result is that every IoT edge node is categorized as either processing-capable or forwarding-only for executing the job according to their computational resource status.

- Processing-capable nodes

The processing-capable nodes are capable of undertaking partial tasks for the IoT job. This includes both mappers and reducers, which retain the same definition and

responsibilities as proposed in **Contribution I**. Mappers and reducers are correspondingly abbreviated as M and R in Figure 12.

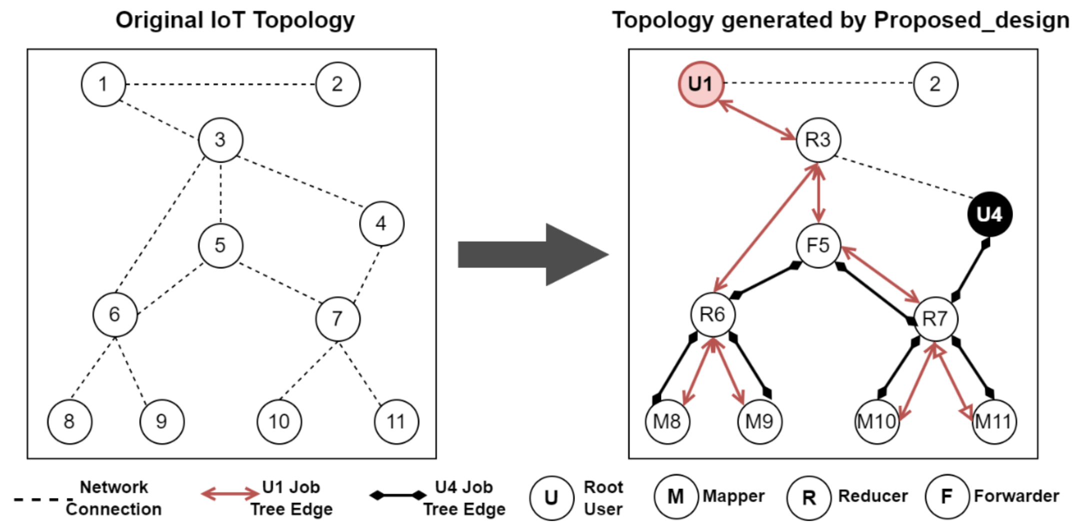


Figure 12 Network Topology Example: Original IoT Vs. Proposed_design

- Forwarding-only Nodes

Some IoT edge nodes cannot execute tasks because of resource constraints. These nodes are named as forwarders, abbreviated as F in Figure 12. In this thesis, forwarders are designed to at least forward packets between their neighbouring nodes if they are incapable of processing data themselves. In detail, a forwarder receives Interests from their upstream neighbours and continues forwarding them to its downstream neighbours along the computational job tree. Forwarders do not need to parse or execute functions within the Interests. When a forwarder receives multiple Data packets for the same job, it integrates all received data samples into a single Data packet before forwarding the data. Data aggregation helps to minimize the number of packet transmissions [26].

- Root User Nodes

Any node within IoT network can issue jobs, it is defined as a root user node, abbreviated as U in Figure 12. A unique job tree is constructed for each root user

before executing their jobs. The root user node serves as the root of its respective tree, with other reducers/forwarders as branches and mappers as leaves. It is important to note that a root user may also function as a branch node in other job trees.

- Job Tree Edges

Regarding the edges on the job tree, the dotted lines in Figure 12 represent original IoT network connections. Each of these connections is possible to become an edge on the computational job trees, depicted as solid lines with arrows and colours. Some of them may be shared by different job trees. However, the loops need to be decomposed to guarantee the correctness of the job computation. Only the edges on the job tree are used to exchange Interest and Data packets.

4.2 Computation Job Tree Construction and Maintenance

The proposed solution requires all IoT nodes to communicate with each other to establish a tree-based logical connection for computation jobs. The computational job tree is built based on the NDN routing table. Every node has its own table so that it knows how to reach a specific node from itself. However, a node has no idea of the routing information inside other nodes. Therefore, it is necessary for all nodes to exchange their routing information to form the computational job tree.

The construction of the computational job tree is initiated when a root user intends to issue jobs. The naming scheme of the BuildJobTree Interest remains the same as commands (a) and (b) proposed in **Contribution I**. To facilitate comprehension, these commands are provided below:

/NeighbourName/BuildJobTree/JobTreeID (a)

/NeighbourName/BuildJobTree/JobTreeID/UpstreamNodeName (b)

The enhancement introduced in this stage involves the introduction of two tables in the application layer. These tables are responsible for storing job tree-related information, enabling all nodes to participate in multiple job trees simultaneously. This allows for greater flexibility and scalability in the execution of different job trees. The first table is Job Tree Table (JT-Table) which stores information in "JobID - JobNeighbours" pairs. This table is instrumental in executing multiple jobs within the framework. It allows nodes to maintain knowledge of their neighbouring nodes within a specific job tree. The second table is BuildJobTree Table (BJT-Table) which serves as a temporary storage for the replies received from neighbour nodes regarding their willingness to join the current job tree. Each row in this table follows the format of "NeighbourName - PendingReply". The records in this table are cleared once the construction of the current computational job tree is completed.

During the process of building computational job trees, all nodes actively participate. Reducers and forwarders handle two key events: receiving BuildJobTree Interests and sending out BuildJobTree Interests. Mappers, on the other hand, solely receive these Interests and provide the necessary replies.

When a reducer or forwarder receives a BuildJobTree Interest (a), it goes through the following six steps to build the job tree as listed in Figure 13:

1. The reducer/forwarder retrieves the JobTreeID within the Interest and checks if the JobTreeID exists in the JT-Table. If it exists and the corresponding JobNeighbours are not null, the discovery of current job tree has been done and the root user can issue tasks, jumping to the job execution procedure.

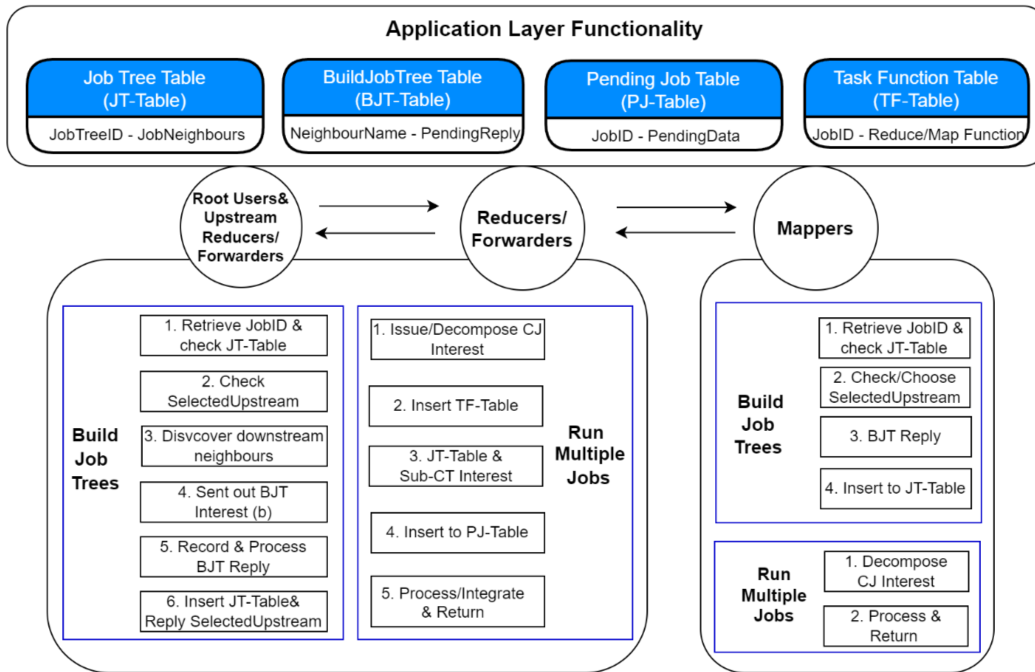


Figure 13 Application Layer Functionality for Multiple Jobs Execution

2. If the JobTreeID is not found in the JT-Table, the reducer/forwarder parses the JobTreeID to extract the root user's name. It then checks if the received Interest is sent by the neighbour on its NDN routing table to reach the root user, named as SelectedUpstream for clarity. If the received BuildJobTree Interest is not from the SelectedUpstream, the reducer/forwarder replies "nope" immediately, which indicates that the reducer/forwarder will not use this path to return data for current job tree. Otherwise, it proceeds to the next step.

3. The reducer/forwarder continues exploring downstream neighbours for this job tree before replying to its SelectedUpstream. However, if a node has no child node to provide data, it also replies "no" immediately to its SelectedUpstream, ending the job tree construction. For instance, node-2 in Figure 12 receives the BuildJobTree Interest from U1, it cannot join U1's job tree because it does not have child nodes.

4. The reducer/forwarder creates a BuildJobTree Interest (b) for each of its child neighbour and sends out the Interest. The child reducers/forwarders repeat the step to explore their downstream neighbours if they also have child nodes.

5. For each received reply for the BuildJobTree Interest (b), the reducer/forwarder extracts the reply content, i.e. yes or no, and saves it in the BJT-Table.

6. Once all child nodes have returned their replies, the reducer/forwarder checks all records in the BJT-Table and saves the child with positive answer into the JT-Table for further job dissemination. If at least child node has returned “yes” as an answer, the reducer/forwarder replies “yes” to its SelectedUpstream node. Otherwise, it returns “no” to indicate not participating in the current job tree.

The R3 in Figure 12 is chosen as an example to explain the actions taken by reducers and forwarders. Suppose the job tree with U1 as the root is built firstly and followed by the job tree for U4. R3 re-writes the BuildJobTree Interest as below for the job tree U1 (e.g. JobTreeID: U1).

/NeighbourName/BuildJobTree/U1/3

As R3 has three neighbours in the original IoT network, it sends a BuildJobTree Interest to node 4, 5, and 6 respectively by replacing the */NeighbourName* with their name. Meanwhile, R3 inserts three records in the BJT-Table and waits for replies. In detail, U4 replies “no” after it receives the answer from its downstream neighbour R7 which does not choose U4 as the SelectedUpstream. Both F5 and R6 return “yes”. After R3 gets all replies from its neighbours, it checks and summarises the information in the BJT-Table. Finally, R3 inserts a row (i.e. U1 - /F5/R6) in the JT-Table for further job dissemination and empties the BJT-Table. Similarly, after the job tree of U4 (e.g. JobTreeID: U4) is built, the JT-Table of F5, R6 and R7 is updated

to contain records for two job trees, i.e. U1 and U4. R3 does not join the job tree of U4 so that its JT-Table has no change.

Mappers maintain their JT-table as "JobTreeID– SelectedUpstream" pairs. One mapper may receive multiple BuildJobTree Interests from different reducers/forwarders. When a mapper receives a new BuildJobTree Interest, it retrieves the JobTreeID and checks its JT-Table as the first step, shown in Figure 13. If there has been a record for the same tree and this Interest is not from the saved SelectedUpstream, the mapper replies "no" to this neighbour. If the received JobTreeID is a new one, the mapper selects one reducer or forwarder as the upstream for each job tree. The last step of the mappers is to insert the SelectedUpstream to its JT-Table. Mappers currently do not discover downstream neighbours during job trees construction because this thesis assumes mappers process the data from all connected sensors.

4.3 Multiple Jobs Execution

The root users can trigger the job execution procedure after their job tree is ready. Each user's job is called a ComputingJob (CJ) Interest and written as (h), which is slightly different from the one proposed in **Contribution I**. In detail, */JobNeighbours* are the neighbours' name stored in the JT-Table. */JobTreeID* is used to retrieve the information from JT-Table. */JobID* is created by the root user for each issued job. It is worth to mention that the job tree ID remains unchanged once the computational job tree is built and users could issue multiple jobs on the job tree. The rest of the CJ Interest (i.e. */MapFunction/ReduceFunction/contentFilter)* is the actual job content and defined by the root user.

/JobNeighbours/JobTreeID/JobID/MapFunction/ReduceFunction/ContentFilter (h)

Two tables are designed to guarantee the correctness of running multiple jobs. Pending Job Table (PJ-Table) stores "JobID – PendingData" pairs. This table keeps track of the pending data associated with each job. Task Function Table (TF-Table) maintains the "JobID – Reduce/Map Functions" pairs. It saves the reduce and map functions specific to each job. The job execution procedure follows a reverse direction compared to the job dissemination process. The CJ Interest is sent by root users and traverses through intermediate reducers and forwarders until it reaches the mappers. The PJ-Table and TF-Table ensure the proper execution and coordination of multiple jobs within the computational job trees.

The reducers maintains both PJ-Table and TF-Table. When a reducer receives a CJ Interest, it performs five steps as listed in Figure 13:

1. Decomposing the Interest to obtain the JobTreeID, JobID and the job content (defined functions and required data).

2. Parsing the job content to extract the specific reduce function and stores in its TF-Table.

3. Searching the JobTreeID in its JT-Table in order to obtain its JobNeighbours for the current job tree. Then it rewrites the received CJ Interest by adding its own JobNeighbours at the beginning (named as subCJ Interest for clarity) and send it out.

4. A new row is inserted into its PJ-Table for every subCJ Interest in the format of "JobID - 0". The content "0" will be updated when corresponding Data packet is received. The PJ-Table is created for every job so that different Data packets can be appropriately processed.

5. When all JobNeighbours reply for the subCJ Interests of the same job, the reducer retrieves corresponding reduce function in its TF-Table, runs the function on

all received data and then returns the processed result to its SelectedUpstream or the root user.

The steps for forwarders to support multiple jobs execution are similar to the described workflow of reducers. There are two differences to be noted. One is that forwarders have no TF-Table as they do not need to run user-defined functions on received data. The other is that forwarders aggregate multiple subCJ for the same job into one and then return to their SelectedUpstream.

When a job is received by the mappers, they firstly decompose the CJ Interest to get the user-defined map function and store it in their TF-Table. Every mapper gathers the data from connected sensors and run the map function to process the data. The outputs of mappers are key-value pairs and returned to their SelectedUpstream. All mapper data is further processed by the reducers at each level of the job tree. The root user gets the final result(s) returned from its JobNeighbours.

4.4 Evaluation

A series of tests have been executed to verify the feasibility of the proposed design. All simulations are performed on ndnSIM [80] and a network generator BRITE [81] is utilized to create the topology of the IoT network. A fixed number of twenty reducers and forwarders is employed (Node Id 0-19) during the tests. Any of them may act as a user node to issues jobs. To measure the changes of network traffic, this thesis adjusts the proportion of reducers/forwarders within the network as well as the number of mappers connected to each reducer and forwarder.

Two types of data transmission speed are set, each characterised by a combination of bandwidth and delay. One type operates at a speed of 250 kbits per second with a 10-millisecond delay between a mapper and a reducer/forwarder, based on the Zigbee

protocol [82]. The other has a speed of 54 Mbits per second with a 1-millisecond delay between reducers and forwarders using the IEEE 802.11 parameter [83].

4.4.1 *Feasibility Test*

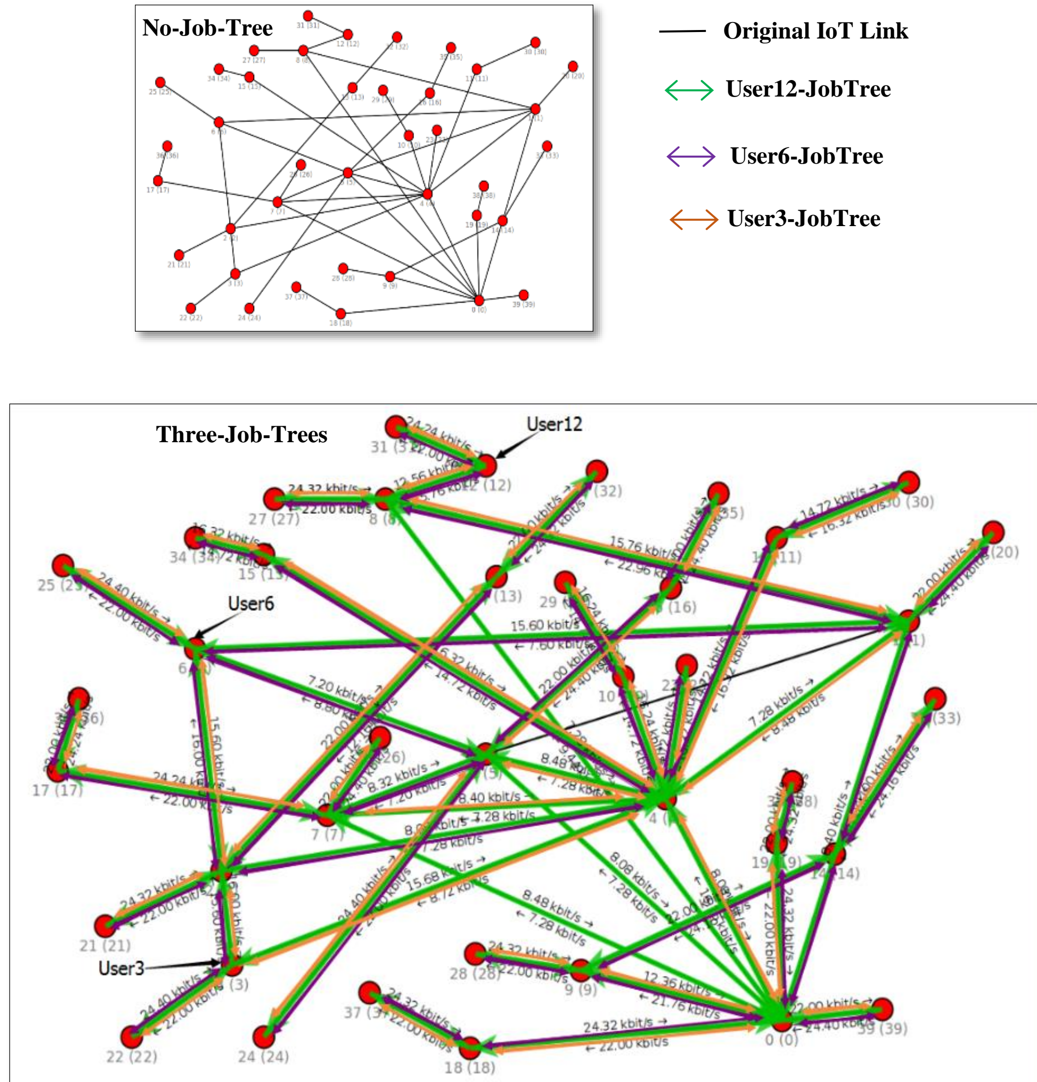


Figure 14 Network Topology and Generated Job Trees

To assess the job execution on the proposed design, this thesis configures three root user nodes (Node 3, 6 and 12), five forwarders (Node 15, 16, 17, 18 and 19) and the rest twelve nodes as reducers. Each root user starts issuing jobs at different times, specifically Node 12 starts at 0th second, Node 6 at 3rd second and Node 3 at 6th second. It is easy to observe that the original IoT network (No-job-Tree) generated

by BRITE exhibits loops in Figure 14. The objective is to evaluate whether the proposed solution can decompose the network into a tree-based graph to avoid loops. For the scenarios with multiple jobs running, the tests will validate the protocol's ability to separate different jobs, process corresponding data and return accurate results is examined.

Figure 14 depicts the traffic on the IoT links when executing from zero to three jobs. All IoT nodes run the proposed protocol to build a unique job tree for each root user. Consequently, only the edges on corresponding job tree are used to forward Interest/Data to their job neighbours. The green, purple, and orange lines with arrows represent the job tree edges for user-12, user-6, and user-3, respectively. The presence of a black line indicates that it is not utilized by any of the current three jobs. After building the three job trees, the simulation continues for 100 seconds with a frequency of one Interest per second per user. The results demonstrate that the proposed design successfully completes all jobs by returning the correct results to each root user.

4.4.2 Network Traffic Comparison

To evaluate the performance of the proposed design, a comparison study called Central-User-benchmark is designed to make the root user directly request all the sensory data and centralised process the data by itself. Node 3, 6 and 12 are still the root user nodes in the Central-User-benchmark test cases and they send Interest (i) to request data. The effects of different proportions of reducers/forwarders on the network traffic are also investigated. Four combinations are considered: 5Reducer-15Forwarder (abbreviated as 5R-15F for clarity), 10Reducer-10Forwarder (10R-10F), 15Reducer-5Forwarder (15R-5F) and 20Reducer-0Forwarder (20R-0F). Each reducer and forwarder connect with five mappers. The tests are conducted five times, randomly select 5, 10 and 15 reducers for the 5R-15F, 10R-10F and 15R-5F setting

respectively. The CJ Interest used for tests is expressed as command (j). As mappers are assumed to gather data from IoT sensors, four data sizes produced by the mappers are simulated: 25, 100, 500 and 1000 bytes respectively. Moreover, this thesis sets different compression levels for the reducer functions in the tests to evaluate the impact of compression rates on the network traffic. Assuming the received data size on each node is N, the output after applying the reduce function is set to 0.25N, 0.5N, 0.75N and N. Each test lasts for 100 seconds with the Interest sending frequency of 1 per second. Every mapper generates a data sample for each received Interest.

/SensorName (i)

/JobNeighbours/JobTreeID/JobID/Map(k,v)->(k+1,v*2)/Reduce(v1,v2)>(v1+v2)/allSensor (j)

The results of network traffic are summarised in Figure 15. It is evident that the Central-User-benchmark test cases generate significantly higher network traffic compared with any test of the proposed design. In the Central-User-benchmark approach, each sensory data requires a separate Interest to be sent by the user node, and the same number of Data packets is transmitted back through the entire network to the user node. This process results in a sharp increase in network traffic. Along with the sensory data size grows from 25 to 1000 bytes, the data transmission by the Central-User-benchmark approach shifts from almost 8000 to 40000 kilobytes, shown as red-colour columns in the figure.

When the sensory data size is fixed and the data compression rate is the same, having more reducers leads to a decrease in IoT data traversing the network. For example, given a sensory data size of 25 bytes and a data compression rate of 0.25N, leveraging 20R-0F setting significantly diminishes network traffic by approximately 52%, reducing transmitted data to around 3800 kilobytes compared with roughly

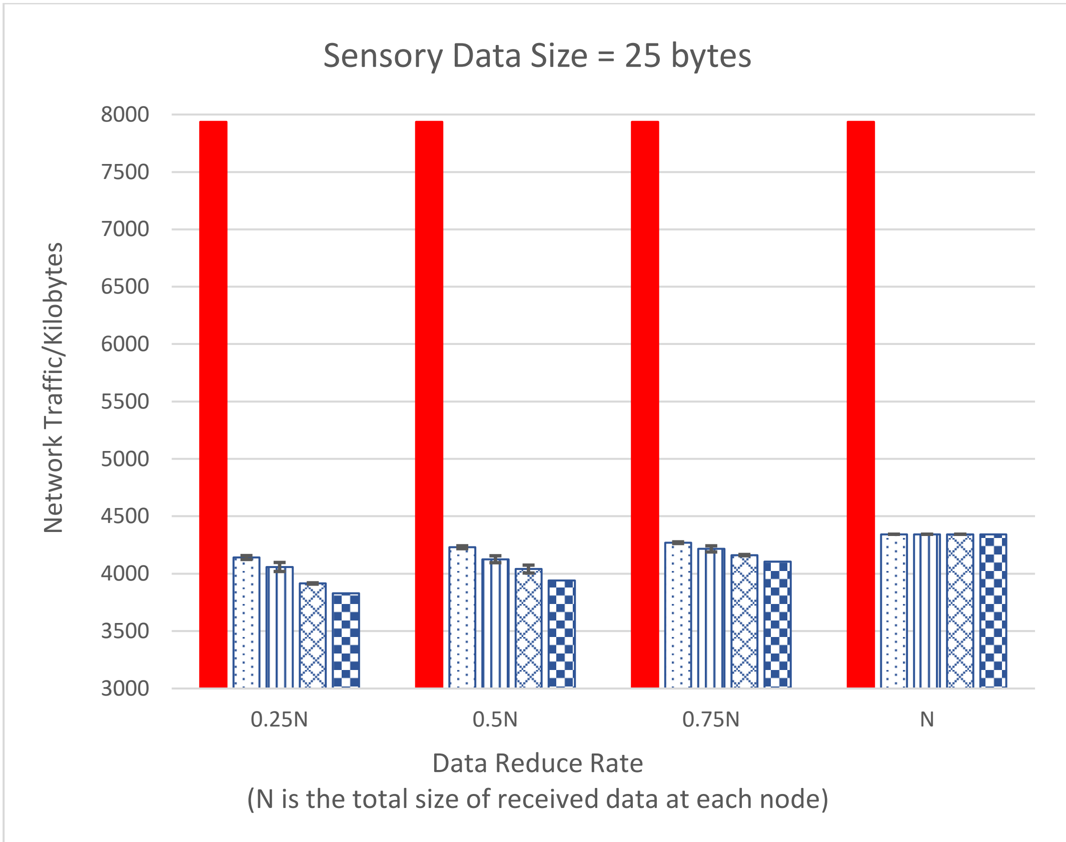
8000 kilobytes network traffic generated in the Central-User-benchmark approach. This network load saving further increases to 59% when the sensory data size grows to 1000 bytes with the same data reduce rate 0.25N. This stands in stark contrast to the Central-User-benchmark approach, which transmits roughly 8000 kilobytes of data. Furthermore, network traffic can be decreased by employing more efficient data processing functions with the same number of reducers and a fixed sensory data quantity. For instance, observing the test case with the sensory data size of 500 bytes and 15R-5F setting, the generated network traffic decreases from about 167000 to 11700 kilobytes with the data reduce rate changes from 0.75N to 0.25N.

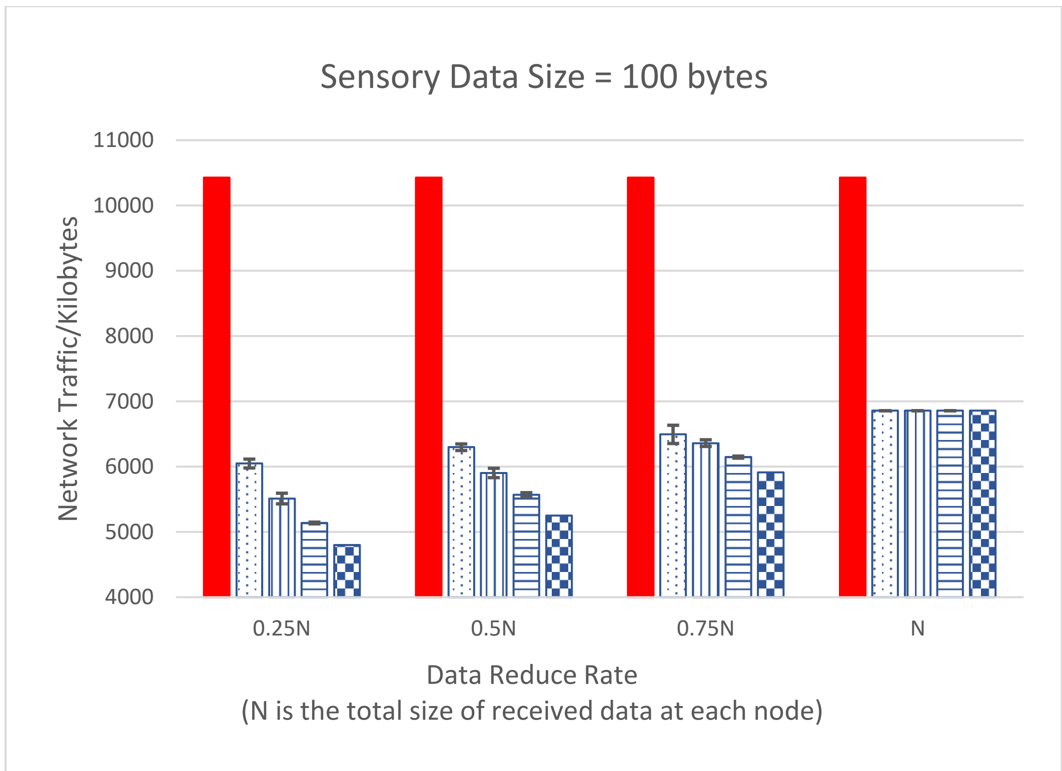
Additionally, the test results validate that the proposed design effectively lowers network traffic even when reducers exclusively perform data aggregation, indicated by a data reduce rate of N. This reduction is attributed to the design's ability to aggregate multiple data packets into one, resulting in a smaller number of packets transmitted within the network and less packet overhead.

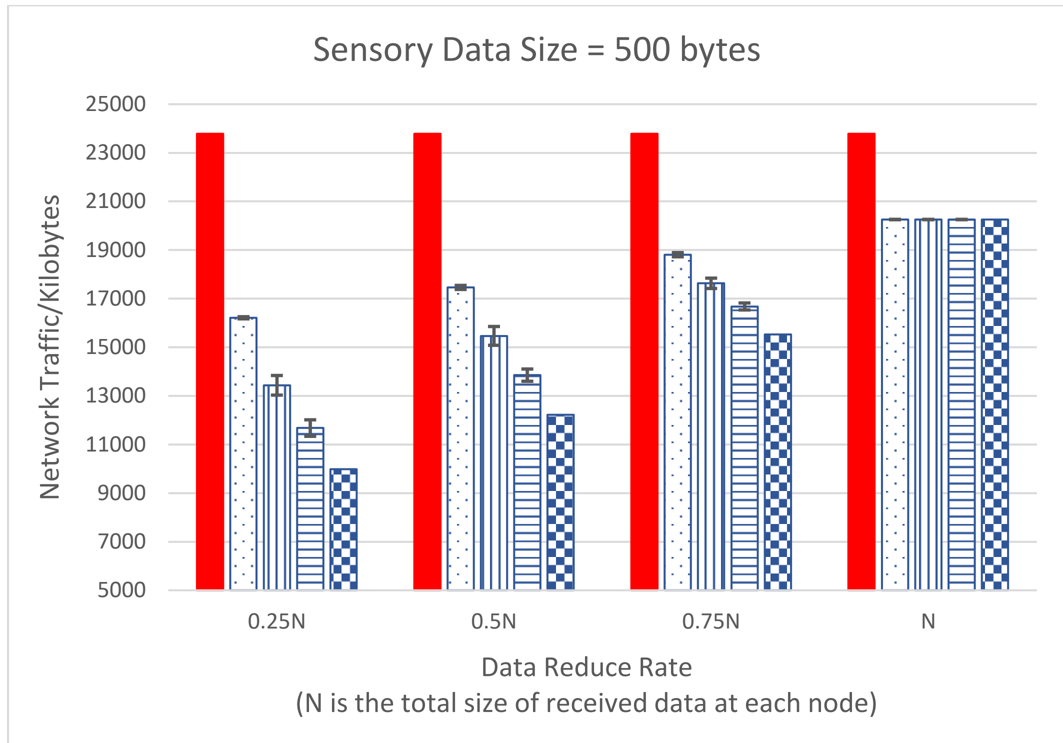
Although employing more reducers does lead to a reduction in network traffic, the magnitude of this reduction is not consistently significant. The network load saving while increasing the number of reducers is decided by the ratio between the payload size (sensor readings) and the NDN packet header size (mainly the name of the content). During all tests, the Interest naming for job dissemination is roughly 100 bytes. When the sensory data size is smaller than the NDN packet header, e.g. test cases having sensory data size of 25 bytes, the packet overhead constitutes a major portion of the network traffic that cannot be compressed or reduced. However, in scenarios where the sensory data size significantly surpasses the NDN packet header (e.g., sensory data size of 1000 bytes), the benefits of activating more reducers become notably pronounced. These tests aim to explore the potential effects of

different reducer numbers and data reduction rates on the network traffic for completing the same computation job. The results show that the optimisation of the job deployment needs to consider the reducer number, location, and the data reduction rate of the job. Optimisation is the future work of this thesis.

The proposed design demonstrates a reduction of 10% ~59% (affected by the number of reducers and data reduce rate) in network traffic in all test cases compared with the Central-User-benchmark solution. This highlights the effectiveness of the proposed design in enabling IoT data processing through the collaboration of edge nodes while significantly reducing the volume of transmitted data within the IoT network.


Sensory Data Size = 25 bytes
Network Traffic/Kilobytes
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
0.25N
0.5N
0.75N
N
Data Reduce Rate
(N is the total size of received data at each node)


Sensory Data Size = 100 bytes
Network Traffic/Kilobytes
11000
10000
9000
8000
7000
6000
5000
4000
0.25N
0.5N
0.75N
N
Data Reduce Rate
(N is the total size of received data at each node)



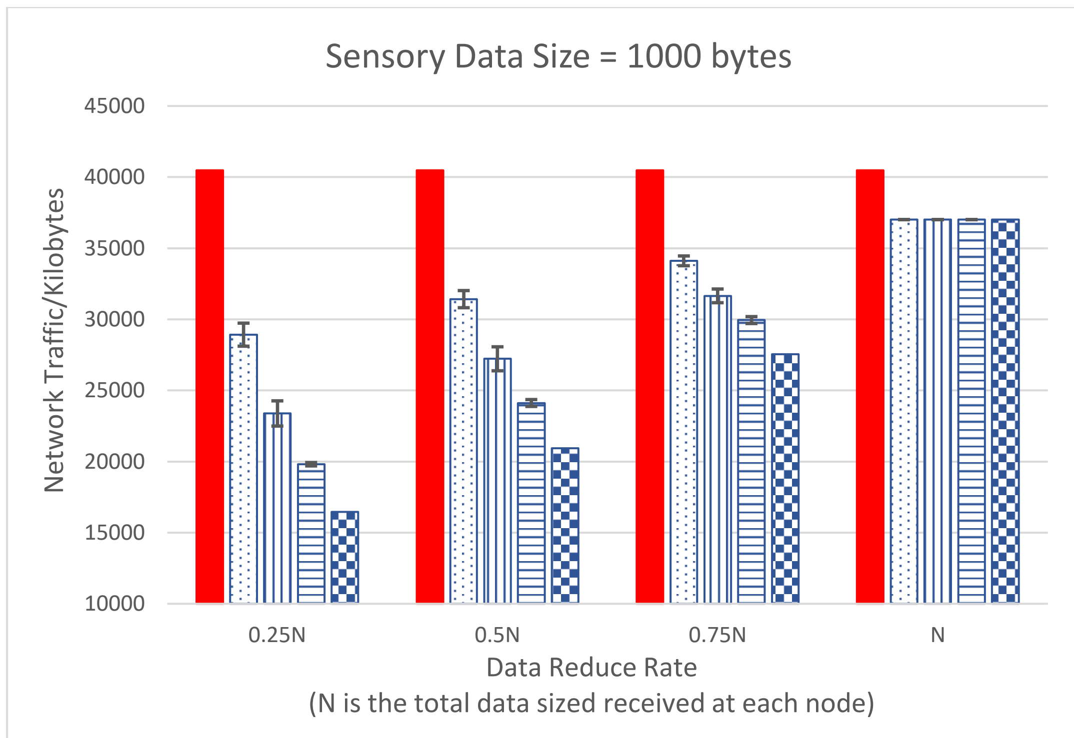


Figure 15 Network Traffic Comparison

4.5 Summary

The proposed design takes into consideration the varying computational resources of edge nodes and aims to organize them in a cooperative manner to support multiple IoT applications simultaneously. Two types of roles for the nodes are defined based on the computational capabilities of edge devices: processing-capable nodes including mappers and reducers and forwarding-only nodes acting as forwarders.

The mappers are responsible for running user-defined map functions on sensory data. They gather data from connected sensors and process it according to the specified map function. The forwarders, on the other hand, focus on data aggregation without performing any processing. Their role is to aggregate data from multiple sources and forward it to the appropriate destinations. The reducers are capable of running user-defined reduce functions on the data received from the mappers, forwarders, or other reducers. A shortest path tree is built for each job to enable collaboration among edge nodes and satisfy the compute-once requirement for the same data. To execute multiple jobs simultaneously, the proposed design incorporates application layer functionality to manage job trees and job processing related information.

The feasibility of the proposed design is verified through tests conducted on the ndnSIM simulator. The test results demonstrate that the proposed solution significantly reduces IoT network traffic compared to centralized processing.

5 Protocol for Exactly Once Data Computation

This chapter presents the third contribution of this thesis, which focuses on improving the proposed framework with exactly once computation guarantee. While exactly once data processing/delivery has been maturely developed in traditional big data processing systems, e.g. Apache Flink [18], the unique requirements and resources constraints of IoT collaborative edge computing scenarios pose challenges for applying existing datacentre-oriented solutions. In this context, this thesis identifies three challenges in ensuring exactly once data computation in IoT collaborative edge computing and propose a five-phase protocol to address them. Simulation experiments are developed to evaluate and compare the performance of the proposed design with a checkpoint-based benchmark solution.

5.1 Motivation

The proposed framework (**Contribution II**) distributes data computation among multiple edge nodes and relies on their collaboration to complete users' jobs. In fact, IoT network connections may fail between two edge devices during job execution. It could result in data loss or duplicated data transmission and/or processing, which is not tolerable by IoT applications with exactly once data computation requirements.

Existing works in this area are scarce and initial endeavours have explored the use of checkpoint schemes as a solution [20] [21]. However, these works are limited to task execution on a single edge device. Although checkpoint based solutions have

been well-developed in traditional big data processing frameworks like Apache Spark [28] and Apache Flink [18], this thesis argues that these solutions are not suitable for IoT scenarios. Firstly, it is not necessary to restore the logical graph onto the same device(s) in traditional data centre scenarios with powerful servers in proximity. In sharp contrast, the logical job graph is tightly coupled to the physical topology in IoT edge environments. Data processing task(s) cannot be placed at a random edge device to replace the previous failed one as it needs to evaluate the benefits of transmitting data versus processing/aggregating the data. Secondly, the traditional checkpoint approach requires the system to take a snapshot of each operator's state periodically and save them to a durable storage, e.g. HDFS [71], which is not widely available in edge computing environments.

Thus, this thesis identifies the following challenges to achieve exactly once computation in IoT collaborative edge computing scenarios.

Challenge-1. Backup essential data processing information in distributed edge nodes. Edge collaboration can be interrupted by IoT network failures due to unstable network connections and IoT device mobility. It is crucial to decide which information of data processing is essential and sufficient to recover from the failures. Then it brings the challenge of how to save this information efficiently. Unlike datacentre environments, central storage for the essential information is not practical in IoT edge scenarios. As edge computing is proposed to complement cloud computing to deal with the high volume/velocity/variety of data produced by massive amounts of IoT devices, it is preferable to distribute the information storage on the edge.

Challenge-2. Handle network failures during edge collaboration while guaranteeing exactly once computation on the same data. When the network

connection between two edge devices fails, it breaks the original job execution graph that includes these devices. The downstream edge device cannot be certain if its data has been successfully delivered to its upstream neighbour. This requires designing a scheme to utilize the information described in **Challenge-1** to repair the job execution graph to resume normal data processing. It also needs to check whether any data has been lost or duplicate processed due to the network failure.

Challenge-3. Limited storage space at edge devices. Only capable edge devices can participate in collaborative edge computing for IoT applications. The burden of edge devices becomes heavier if they need to process data meanwhile store relevant information. Thus, the information described in **Challenge-1** cannot be saved on edge devices permanently. As edge devices cooperate with each other to complete each IoT job, one edge device randomly deletes some information at its local storage may affect the whole job processing procedure. For example, the job cannot be recovered from the failures described in **Challenge-2** if the information saved on edge devices has been deleted before the failure happens. As a result, it brings the challenge on how to assess whether the job state related information is out-of-date/of-no-use and then how to clean the information saved on edge devices.

To address these challenges, this thesis designs a protocol consisting of a job execution procedure to solve **Challenge-1** and **Challenge-3** and a job recovery procedure to solve **Challenge-2**. Additionally, a job tree based ID assignment approach is devised to provide fundamental support for the two procedures.

5.2 Five-phase Protocol Design

The proposed protocol consists of five phases and their relationship is shown in Figure 16. It is important to note that the normal job operation is not disrupted by recovering from failures. The definition of each phase is listed as below:

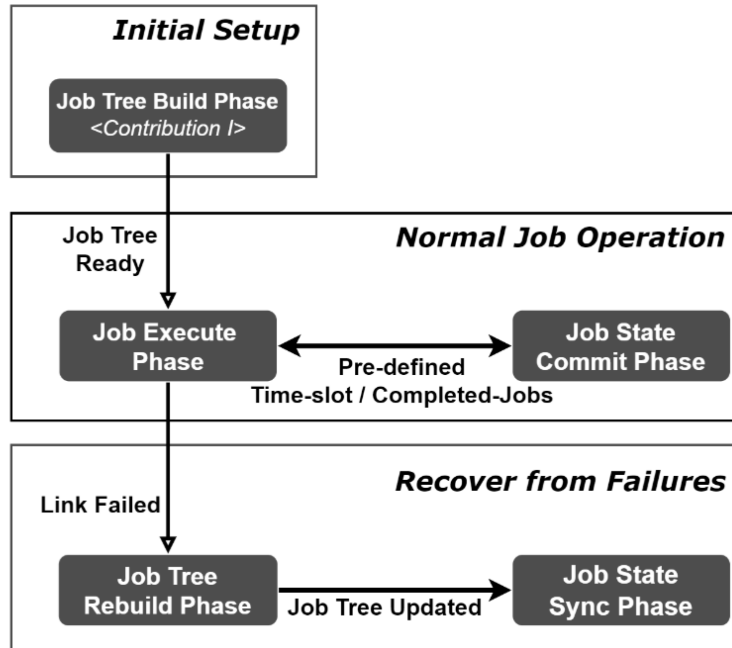


Figure 16 Five Phases of the Proposed Protocol

- **Job Tree Build Phase** forms a job tree with each new user as the root and the user can issue multiple jobs on its job tree.
- **Job Execute Phase** disseminates jobs requests, returns computed results and saves intermediate state while executing the job.
- **Job State Commit Phase** periodically clears intermediate state of completed jobs on edge devices.
- **Job Tree Rebuild Phase** updates the job tree to eliminate failed link(s) when network failures happen.

- **Job State Sync Phase** ensures that link failures and the updated job tree cause neither data losses nor duplicated data computations.

As the proposed framework is built upon NDN, the communication between nodes in all phases is accomplished by exchanging the NDN Interest and Data packets. To facilitate the functionalities at each phase, different Interest naming schemes have been designed. The job tree is created using the shortest path algorithm of the NDN routing protocol, which determines the optimal path for data transmission and computation. Additional metrics, such as link bandwidth [85] and energy efficiency [86], can be considered when constructing the job tree to optimize performance, which is beyond the scope of this thesis. The proposed protocol currently is limited to execute stateless jobs [87] whose output is solely based on its input, not the intermediate computational states. Consequently, the same computation on the same data can be undertaken by any capable edge devices. The computed result is only determined by the number of input values and is independent of their order. Therefore, the data computation can be successfully recovered from a changed job tree due to link failures.

The following sections describe each phase in detail.

5.2.1 Job Tree Build Phase

A tree topology is built with a user node (sink node) as the root before it issues jobs. The job tree construction protocol remains the same as proposed in **Contribution I** (chapter 3.3.2). Upon completion of this phase, each node on the job tree maintains a local record of “JobTreeID – JobNeighbours” information for subsequent phases of the protocol.

5.2.2 Job Execute Phase

The Job Execute Phase commences once the job tree has been established. It

comprises two steps. The first step is node ID allocation that is proposed in this thesis to differentiate each data sample. It serves as the foundation for achieving exactly once data computation. The second step is job dissemination and execution, which follows the procedure outlined in **Contribution II** (chapter 4.3). However, an improvement has been made during the job execution compared with **Contribution II**. Specifically, the intermediate state of job processing is now saved on edge devices. The design enhancement aims to handle link failures that may occur during job execution.

5.2.2.1 ID Allocation

The data content identification is challenging. While it may seem feasible to use NDN names as IDs to uniquely identify each data content, this approach does not reveal which node(s) have computed the data sample. As a result, it becomes difficult to verify the data computation state after recovering from link failures, thus failing to guarantee exactly once computation on the same data.

For two nodes connected by the same edge on the job tree, we designate the one closer to the sink node as the upstream node and the other as the downstream node for clarity in the rest of this thesis. When a link failure happens during data transmission, the downstream node may not be sure whether the data has been successfully delivered. After the downstream node rejoins the job tree by connecting to a different upstream node, it needs to check if the locally cached data had been delivered before retransmitting to ensure exactly once computation. This become more complicated when the data delivered to the previous upstream node is still under transmission/processing in the job tree.

To address these challenges, the proposed protocol incorporates the information of data provider and data computing nodes into the ID of each data content during the job execution. To identify each data content in the network, this thesis firstly assigns a global

ID for each node based on the shortest path of the job tree. ID allocation takes place before any job requests are issued. As mapper nodes are the data sources in the proposed design, they label each of their returned data samples with their node ID plus the job ID created by the user/sink node. Data samples from different nodes can only be computed by reducers if they have the same job ID to ensure the computation correctness. The ID of a computed data content consists of its reducer's global ID plus the job ID as a reference of the computation path for this data. Whenever a link failure happens, the affected node can use the data sample ID(s) to trace back the computation path of its provided data content. This enables the node to inquire about the computation state of its data content, specifically whether received and computed correctly.

An AssignID Interest is designed to assign node ID and it is written as (k). Where, */JobNeighbour* is the name of a neighbour obtained in the Job Tree Build Phase. */JobTreeID* is created by the sink node when sending the job tree building request. */NodeGlobalID* represents the actual global ID assigned to the corresponding job neighbour and it is construed as below.

/JobNeighbour/JobTreeID/NodeGlobalID (k)

The upstream node assigns a unique identifier (e.g. a number) to each of its downstream nodes as a local ID. The records of local IDs are only maintained at each upstream node. Since each node on the job tree has a distinct path between itself to the sink node, a tree-path-based global ID of each node is constructed by accumulating the local IDs along the path from the sink node to itself.

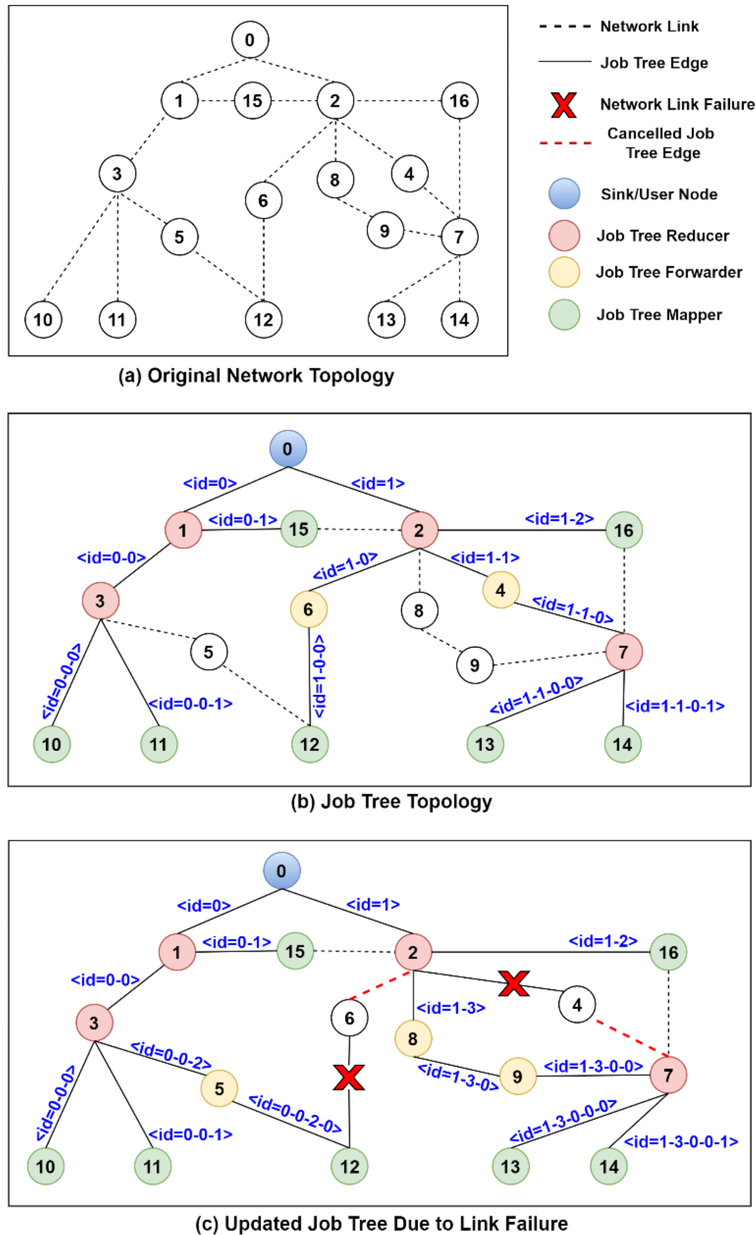


Figure 17 Illustration of ID Allocation

The sink node assigns the global node ID to its neighbours, which is the same as the nodes' local ID as the sink node has no upstream node. The intermediate reducers and forwarders receive their global ID from their upstream node and then allocate global IDs to their downstream nodes. This allocation is achieved by concatenating the local ID of a downstream node at the end of the current reducer/forwarder's global ID, separated by a hyphen. The reducers and forwarders assign global IDs to their neighbours using the AssignID Interest. The mappers, being the leaf nodes of the job tree, only receive the

global ID from their upstream node. All upstream nodes maintain an ID table to save the global and local ID of its downstream neighbours. Each record in the table corresponds to a downstream neighbour, in a tuple of <downstream job neighbour name, its local ID, its global ID>.

The global ID allocation process occurs hop by hop, starting from the sink node and reaching all the nodes on the job tree. An Acknowledgement (ACK) message is replied from the mappers in the reversed path of ID allocation, and ultimately returned to the sink node. Hence, the sink node confirms the completion of ID allocation procedure, and it is ready to issue jobs.

Figure 17 illustrates an example to demonstrate the ID allocation procedure. An IoT network topology is shown in Figure 17 (a) with the original connections between the nodes. The numbers inside each circle represent their NDN name. For instance, “13” is the NDN name of the node 13 and node 1 uses “13” as the “NeighbourName” when constructing the BuildJobTree Interest during the Job Tree Build Phase. The NDN name of a node keeps the same no matter which role it acts in the proposed framework.

Assuming that node 0 wants to issue a job, it acts as the sink node (user node) in the design. It firstly sends the BuildJobTree Interest to the network, resulting in the job tree shown in Figure 17 (b). The solid lines in the figure indicate original network links currently being utilized on the job tree. The nodes labelled with numbers 8 – 14 and highlighted in green, serve as the mappers for the current job. Other nodes may function as reducers or forwarders according to their computing capabilities and the number of downstream neighbours. For instance, node 1 becomes a reducer (in red colour) since it receives data samples from multiple neighbours on the job tree, and it is currently capable of computing these data. Node 6 acts as a forwarder (shown in

yellow) because it connects to only one mapper (node 10). Node 5 does not join the job tree as none of the nodes selects it as the neighbour for sending data to node 0.

Once the job tree is prepared, node 0 as the sink node assigns the local ID to its job neighbours, namely node 1 and node 2. Recursively, every upstream node assigns a number (for simplicity, starting from 0) to each of its downstream neighbour as the local ID. Node 1 receives 0 as its global ID and node 2 receives 1 as its global ID as illustrated in Figure 17 (b) with blue text. Node 1 and node 2 continue the global ID assignment by creating global IDs for their respective downstream neighbours. Specifically, node 1 assigns the local ID 0 to node 3 and local ID 1 to node 13. Then node 1 concatenates node 3's local ID to its own global ID separated by a hyphen symbol, resulting in the global ID of node 3 is 0-0. Similarly, node 15 obtains 0-1 as its global ID. Node 2 assigns local ID 0, 1, 2 to its neighbour node 6, 4, and 16 respectively, and consequently the corresponding global IDs for node 6, 4 and 16 are 1-0, 1-1 and 1-2 respectively. All the intermediate reducers and forwarders follow this rule to allocate a global ID to their neighbours until all mappers receive their global IDs. The blue texts Figure 17 (b) indicates each node's global ID sent by its upstream node on the job tree.

All the upstream nodes create and maintain an ID table to save the details of the assigned local and global IDs. To provide a more detailed explanation, let's consider the path on the established job tree shown in Figure 17 (b) with the nodes: 10/11 -> 3 -> 1 -> 0. Figure 18 (a) illustrates the corresponding ID tables of the sink node 0, reducer 1 and reducer 3. The ID table consists of three columns. The first column saves the NDN name of each downstream node, labelled as "Nei_node". The second and last column are the local ID and global ID of the downstream node. The local ID is only known between two directly connected nodes, where one node acts as the

upstream node and the other as the downstream node. The upstream node assigns and maintains the local ID.

The mappers in the job tree store their global IDs and use the received job ID (sent by the sink node) to label each data they produce, for example, the incremental sequence numbers attached to node 10 and node 11 in Figure 18 (a). Only data content and its ID are returned in the Job Execute Phase. The global ID of a node is used to check whether the data it has produced or computed is affected by link failures.

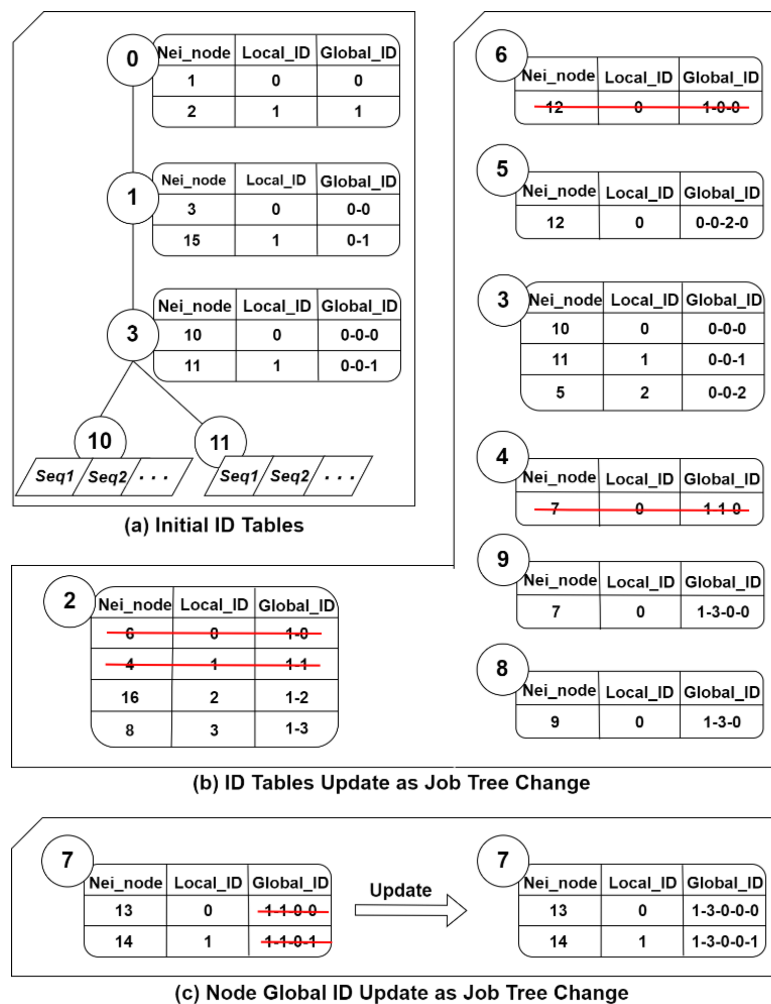


Figure 18 Illustration of Nodes' ID Table

5.2.2.2 Job Dissemination and Execution Procedure

Once the ID allocation is complete, the sink node can send computational jobs by using the CJ Interest. The job processing procedure remains the same as proposed in **Contribution II** (chapter 4.3).

The sink node receives the computed result(s) returned from its job neighbours and perform the final computation, which indicates the completion of the current job. The data processing/computing requirement of an exactly once job is defined follow: all the mapper data requested by the sink node is retrieved and each data sample is computed exactly once on the way to the sink node.

To facilitate the logging of data computation states in case of link failures during job execution, two tables are designed in the proposed protocol. The first table is called the Job State (JS) Table that is managed by the sink node. The table assists in checking the completed job ID(s) in the Job State Commit Phase, allowing the corresponding information stored at edge nodes to be cleared. For each issued job request, the sink node creates a record in the JS Table and checks the received computation results. The job state is saved as a pair of "JobID – State (Completed/Uncompleted)". A completed job signifies that each edge node on the job tree has completed its processing of the issued job request and the final computed result has been correctly delivered to the sink node. This ensures the reliability of the data delivery and computation.

The second table is the Computation Record (CR) Table which saves the job tree ID, the data received from downstream neighbours for the job (abbreviated as dataContent) and its corresponding ID (abbreviated as dataID). Each record in the CR Table is in the form of "JobTreeID – DataID – dataContent", which allows tracking and referencing the data received from downstream neighbours for a specific

job. All nodes on the job tree maintain a CR Table locally. After returning or forwarding the computed/produced data to its upstream node, each node inserts a record into its CR Table.

5.2.3 Job State Commit Phase

As IoT edge devices are resource-constraint, the intermediate state (saved in the JS Table and CR Table) of job execution cannot be stored permanently. Meanwhile, the saved information can only be cleaned if the correspondent task has been completed. The Job State Commit Phase is introduced to achieve this goal.

During this phase, the sink node notifies its neighbours on the job tree about the completion of specific job IDs that occurred in the Job Execute Phase so that all nodes on the job tree can clear the corresponding saved information. To achieve this, the JobCompleted Interest, denoted as (1), is defined. Where, */JobNeighbour* represents the name of a neighbour obtained in the Job Tree Build Phase. */JobTreeID* is created by the sink node when sending the job tree building request. */CompletedJobID(s)* is the successfully computed job ID(s) summarized by the sink node to inform others on the job tree.

/JobNeighbour/JobTreeID/CompletedJobID(s) (1)

The frequency of sending JobCompleted Interests can be determined by the sink node based on the job requirements or the resource constraints of the edge nodes. For example, it can be sent periodically every 30 seconds or after a certain number of completed jobs. This thesis assumes that the sink node is aware of the resource constraints of the edge nodes and then decides the frequency of sending the JobCompleted Interests accordingly. The sink node generates the JobCompleted Interest, which is then forwarded by intermediate reducers and forwarders until it reaches mappers.

As a result, all nodes on the job tree reach a consensus regarding the completed jobs in which they have participated. Thus, they no longer need to maintain historical records of the completed jobs. For example, reducers can clear the cached computed data content and mappers can discard the previously captured data samples. This action helps to release resources and space for the edge devices engaged in the data processing. In contrast, the intermediate processing state of jobs should be saved if nodes receive do not receive any notifications from the sink node. An ACK procedure is employed to response the JobCompleted Interest, which is initiated by the mappers and follows the reverse path of the JobCompleted Interest until it reaches the sink node, indicating the end of the Job State Commit Phase.

5.2.4 Job Tree Rebuild Phase

Any nodes on the job tree that experience link failures can initiate the Job Tree Rebuild Phase to recover. However, in cases where there is only one neighbour in the original IoT network, i.e., the current upstream node, the node must continuously monitor the link until it is restored. For example, referring to Figure 17 (b), node 13 has only one neighbour, node 7, in the network. This thesis focuses on scenarios where nodes have alternative paths connecting them to the sink node, in addition to the failed link.

A failed link affects two neighbouring nodes. To facilitate the explanation of the design, the upstream node is defined as the Previous-Upstreamer and the downstream node is called Rebuilder. For example, if the link between node 12 and node 6 in Figure 17 (c) is disconnected, node 6 becomes the Previous-Upstreamer and node 12 becomes the Rebuilder. The Job Tree Rebuild Phase is always initiated by the Rebuilder. This thesis assumes that the link condition is detected by periodically exchanging HELLO messages between neighbouring nodes, which is a commonly

used scheme in routing protocols. The following procedure is adopted when a link failure is detected.

The Rebuilder checks if it has other neighbours on the original IoT network, excluding the Previous-Upstreamer and its child nodes. Two cases are designed according to the checking result.

Case 1: The Rebuilder has other neighbour(s)

A RebuildJobTree Interest is defined as (m) and (n), each serving a specific purpose. The Rebuilder sends Interest (m), while the neighbours of the Rebuilder forward the rebuilding request using the Interest (n). The meaning of each part of the Interest is: (1) */NeighbourName* refers to the name of each neighbour of the Rebuilder found in the original IoT network, (2) */RebuildTree* is the identifier for the Job Tree Rebuild Phase, (3) */RebuilderName* is the NDN name of the Rebuilder, (4) */JobTreeID* indicates the job tree of interest, and (5) */UpstreamNodeName* denotes the name of the upstream neighbour of the Rebuilder.

/NeighborName/RebuildTree/RebuilderName/JobTreeID (m)

/NeighborName/RebuildTree/UpstreamNodeName/JobTreeID (n)

If the Rebuilder finds any neighbour(s), it sends a RebuildJobTree Interest (m) to each of its neighbours. Upon receiving the RebuildJobTree Interest, a node extracts the JobTreeID in the Interest and checks whether it has already joined on the job tree. The following two scenarios may occur:

Scenario-I: the node has joined the job tree with the requested JobTreeID. The node assigns a local and global ID to the downstream neighbour that sends the RebuildJobTree Interest. It also inserts the corresponding record into its ID table, following the procedure described in the Job Execute Phase (chapter 5.2.2.1).

Subsequently, the node replies a "Rebuild-OK" message with the assigned global ID. If multiple "Rebuild-ok" messages are received, the Rebuilder node always chooses the first one and notifies other neighbours to withdraw its rebuilding requests.

Scenario-II: the node is not on the job tree with requested JobTreeID. The node rewrites the RebuildJobTree Interest as (n) and forwards it to its neighbours, who repeats the above procedure to process the Interest. If a node has no available neighbours to forward the Interest, it directly replies a "Rebuild-Rejected" message. Note that, mappers are defined as not responsible for disseminating or forwarding jobs to others due to their limited resources and capabilities. Hence, when a mapper receives a RebuildJobTree Interest, it refuses the request by replying a "Rebuild-Rejected" message even if it is currently working on the job tree. Finally, if the Rebuilder receives "Rebuild-Rejected" messages from all its neighbours, it takes the same action as defined in Case 2.

Once the Rebuilder receives its new global ID, it can re-enter the Job Execute Phase. Simultaneously, the Rebuilder initiates the Job State Sync Phase to ensure that neither data losses nor duplications occur due to the link failure, as described in Chapter 5.2.5. If the Rebuilder is connected to downstream nodes on the job tree, it notifies them of their global ID change by sending the ChangeID Interest (o). The Interest includes three parts: (1) */JobNeighbour* is the name of a neighbour on the job tree, (2) */JobTreeID* is used to specify the affected job tree in case multiple job trees coexist, and (3) */ChangeID(NodeGlobalID)* informs the downstream neighbours about the new ID assigned for the specific job tree.

/JobNeighbor/JobTreeID/changeID(NodeGlobalID) (o)

Case 2: Rebuilder has no other neighbour(s)

If the Rebuilder is unable to find any neighbours, it needs to inform its downstream

neighbour(s) to search for an alternative path to reach the sink node. This design aims to minimize the number of nodes affected by link failures.

A ChangePath Interest is defined for this case and it is written as (p). In the Interest, */JobNeighbour* is the name of a neighbour used to disseminate jobs in the Job Execute Phase, */ChangePath* is the identifier to notify the downstream neighbours to alter the path for reaching the sink node, */JobTreeID* is to specify the affected job tree in case multiple job trees coexist.

/JobNeighour/changePath/JobTreeID (p)

Each downstream neighbour of the Rebuilder becomes a new Rebuilder upon receiving the ChangePath Interest, which is named as downstream-Rebuilder for clarity. A new round of Job Tree Rebuild Phase is initiated for each downstream-Rebuilder. If the downstream-Rebuilder successfully finds a new path on the job tree, it should notify the Rebuilder by replying a "Leave-tree" message. This notification helps the Rebuilder to maintain its downstream neighbours for the specific job once it recovers from the link failure and re-enters the Job Execute Phase. Any downstream-Rebuilders that have failed to find an alternative path need to regularly check with the Rebuilder to get updates of the failed links (whether it is recovered).

Two examples of link failures are illustrated in Figure 17 (c). The following steps outline the rebuilding procedure for the link failure occurred between node 4 and node 2.

Step-1: Node 4 as a Rebuilder finds that no other neighbours exist except the current upstream node 2 and the current downstream node 7 on the job tree. It notifies node 7 by sending a ChangePath Interest.

Step-2: Node 7 becomes a downstream-Rebuilder and sends the RebuildJobTree

Interest to its neighbouring node 9 and 16.

Step-3: Node 16 is already on the requested job tree, but it replies "Rebuild-Rejected" as it is a mapper. Node 9, not being on the requested job tree, rewrites the RebuildJobTree Interest and sends it to its neighbours. Node 8 takes the same action as node 9 and gets a "Rebuild-ok" message from node 2. Node 9 replies to node 7 after it receives the "Rebuild-ok" message and its global ID from node 8. Details of the nodes' ID table are presented in Figure 18 (b).

Step-4: Node 7 receives its new global ID and notifies its downstream neighbours on the job tree, namely node 13 and node 14, by sending the ChangePath Interest with the corresponding changed global ID. The ID table of node 7 is updated as shown in Figure 18 (c). Moreover, node 7 notifies node 4 of the path change result. Node 4 can re-join the job tree by connecting node 7 as the upstream node if needed.

5.2.5 Job State Sync Phase

The Job State Sync Phase aims to prevent any violations of the exactly once computation requirement due to the job tree changes, i.e. to avoid the local cached data in the Rebuilder to be recomputed if the data has been computed in the previous upstream node of this Rebuilder. The Rebuilder initiates this phase after it finds a new path to recover from link failures. The synchronization process begins with the sink node and traverses the reducers or forwarders along the previous path (prior to the link failures) until it reaches the Previous-Upstreamer of the Rebuilder. It is important to note that any newly arrived data from downstream nodes to the Rebuilder after the link failure will be processed as normal. Therefore, the Job State Sync Phase can coexist with the ongoing Job Execute Phase.

A JobSync Interest is defined for the Job State Sync Phase, as shown in (q). The

different parts of the Interest hold the following meanings. (1) */SinkNodeName* is the NDN name of the sink node. As the sink node gathers all computed results for each job, the Rebuilder firstly asks the sink node as the starting point. (2) */JobSync* is the identifier for the Job State Sync Phase. (3) */RebuilderGlobalID* is the global ID of the Rebuilder. (4) */JobTreeID* is to indicate the specific job tree in case multiple job trees running at the same time. (5) */JobID/DataID* contains the ID(s) of data-samples for specific job to be checked.

/SinkNodeName/JobSync/RebuilderGlobalID/JobTreeID/JobID/DataID (q)

The following steps are undertaken in this phase:

Step-1: The Rebuilder constructs the JobSync Interest and sends it to the sink node.

Step-2: Upon receiving the JobSync Interest, the sink node parses it to extract the */JobID*. It first checks if the task associated with the JobID has been completed. If the task is marked as completed, it means that all the data content has been correctly computed and received, and therefore the data samples to be checked are unaffected by the Rebuilder's link failure. In this case, the sink node can reply with a "DataSample-Received" message to the Rebuilder, indicating that the Job State Sync Phase has finished. However, if the task state of the JobID is marked as incomplete, it implies that the corresponding job execution is still ongoing, and the sink node requires more information to respond to the JobSync Interest.

The sink node further extracts the *RebuilderGlobalID* and *DataID* from the JobSync Interest. It searches the *RebuilderGlobalID* in its ID table resulting in the two cases below.

If the RebuilderGlobalID is found in the sink node's ID table, it means that the sink node is the Previous-Upstreamer of the Rebuilder. The sink node then checks

the DataID in its JS Table. If the data associated with the DataID has been received, the sink node replies with a "DataSample-Received" message to the Rebuilder, indicating that the Job State Sync Phase has finished. However, if the data sample has not been received, the sink node replies with a "DataSample-Not-Received" message and requests the Rebuilder to resend the missing data.

If the sink node fails to find the RebuilderGlobalID in its ID table, it needs to forward the JobSync Interest to the previous path of the Rebuilder before the link failure occurred. To determine the next hop node to reach the Previous-Upstreamer of the Rebuilder, the sink node decomposes the RebuilderGlobalID. As described in chapter 5.2.2.1, the global ID of a node comprises the global IDs of its upstream neighbors separated by hyphens. Since the sink node is the starting point of each individual path on the job tree, it extracts the first sub-ID (the number before the first hyphen) to identify the next destination node for forwarding the Interest. The sink node compares this sub-ID with all the assigned local IDs in its ID table. The node with a matching local ID is determined as the next hop node (referred to as NextHop) to forward the JobSync Interest.

To assist downstream nodes in parsing the message, the sink node creates a new Interest called ForwardJobSync, as defined in (r). The Interest is based on the JobSync Interest with two different components. */NextHopName* is the NDN name of the NextHop. */HopNum* is the hop number of the current node to reach the sink node on the job tree. This design assists other nodes to parse the *RebuilderGlobalID* in the ForwardJobSync Interest.

/NextHopName/DataCheck/RebuilderGlobalID/JobTreeID/JobID/DataID/HopNum (r)

Step-3: The NextHop node extracts the *RebuilderGlobalID* and *DataID* after receiving the ForwardJobSync Interest. It then checks each data sample ID in its CR

Table. If the data sample is received, it means this node is either the Previous-Upstreamer of the Rebuilder or the upstream node of the Previous-Upstreamer which has received the processed data content after the link failure. The NextHop node replies a "Data-received" message for each received data sample to the node (either the sink node or an upstream NextHop) that sent the ForwardJobSync Interest.

If the *DataID* is not found in the CR table, the NextHop node searches the *RebuilderGlobalID* in its ID table. If the *RebuilderGlobalID* is found, it idicates the ForwardJobSync Interest has reached the Previous-Upstreamer of the Rebuilder. The NextHop node replies "DataSample-Not-Received". If the NextHop node fails to find the *RebuilderGlobalID* in its ID table, it modifies the *NextHopName* and *HopNum* parts of the ForwardJobSync Interest and forwards it to the downstream NextHop. Suppose that the *HopNum* is n in the received ForwardJobSync Interest, the current NextHop node knows that the hop number of its upstream node is n so that its own hop number equals to $n+1$, which means the current NextHop node extract the $(n+1)th$ sub-ID as the local ID of the next destination node. It then finds the neighbour with the matched local ID, replacing the *NextHopName* by the neighbour's name. Step-3 is repeated until a NextHop node finds the *RebuilderGlobalID* that matches one of the neighbours' global ID in its ID table.

Step-4: If a NextHop node is neither the Previous-Upstreamer of the Rebuilder nor the one found the matched *DataID* content in its CR Table, it simply forwards received reply message.

Step-5: The sink node receives the replied message. If the message content is "DataSample-Received", the sink node forwards this message to the Rebuilder, which means the Job State Sync Phase has finished. If the message content is "DataSample-Not-Received", the sink node asks the Rebuilder to resend those data.

The Job State Sync Phase is complete when the sink node receives all the missed data-samples from the Rebuilder.

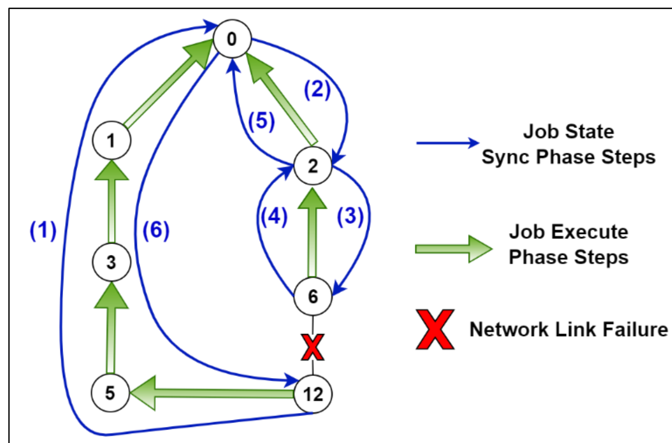


Figure 19 Procedure of Job State Sync Phase

Figure 19 presents an example to demonstrate the Job State Sync Phase. Node 12 finds a new upstream node (node 5) after the link between itself and node 6 fails. The green lines with arrows in the figure indicate the normal job flow in the Job Execute Phase. Steps of the Job State Sync Phase are the blue lines with arrows, labelled as steps (1) – (6). To explain in detail:

(1) Node 12 as the Rebuilder sends the JobSync Interest to node 0.

(2) Node 0 as the sink node checks the job ID, node global ID and data ID embedded in the Interest and does not find the corresponding records. Therefore, it constructs the ForwardJobSync Interest and sends to the next hop neighbour.

(3) Node 2 as the NextHop parses the received Interest and checks the embedded node global ID and the data sequence numbers. As it does not find matched information, it modifies the ForwardJobSync Interest and continues the forwarding process.

(4) Node 6 as the NextHop of node 2 receives the ForwardJobSync Interest. It

finds that the *RebuilderGlobalID* within the Interest matches one of its downstream neighbour's global ID. Therefore, Node 6 is the Previous-Upsteamer of node 12. It checks the corresponding data computation records and then replies.

(5) Node 2 as the intermediate NextHop node forwards the reply from node 6 to node 0.

(6) Node 0 replies to node 12 according to its received message content.

5.3 Protocol Overhead Analysis

The overhead incurred by the proposed design includes two parts. One is the computation records saved at each node and the other is the network traffic generated to handle link failures and ensure exactly once data computation.

5.3.1 Network Traffic Overhead

The network traffic transmitted in the Job Tree Build Phase and the Job Execute Phase is defined as the actual job traffic, which sends job requests and returns computed job results in the formed job tree. Extra cost besides the actual job traffic is required to deal with link failures and guarantee the exactly once computation on the same data, which includes the Job Tree Rebuild Phase, the Job State Sync Phase and the Job State Commit Phase, abbreviated as Proposed-RSC phases. For clarity and simplicity, let $X_{RSC-Interest}$ and $X_{RSC-Data}$ denotes the corresponding size of the Interest and Data packets used in the Proposed-RSC phases, including the RebuildJobTree, the ChangePath, the JobCompleted, the JobSync and the ForwardJobSync Interests introduced in previous chapters.

In the Job Tree Rebuild Phase, at most three procedures contribute to the overhead traffic. The first involves the Rebuilder searching for alternative path(s) to reconnect

with an upper stream neighbour that is already on the job tree. For example, in Figure 17 (c) node 3 is the upper stream neighbour of node 12. We call this upper stream neighbour as Joint-Upstream for clarity and use $D_{Rebuilder}^{Joint-Up}$ to represent the path distance between the Rebuilder to the Joint-Upstream. The second procedure is the notification of ID change. After the Rebuilder finds a new path to re-join the job tree, it receives a new global ID. If the Rebuilder is a mapper node, the second procedure can be ignored as mapper nodes have no downstream neighbour in the proposed design. Otherwise, the Rebuilder needs to update the global ID of all its downstream neighbours and notify them of the change. Suppose N_{child} denotes the total number of nodes on the sub-tree with the Rebuilder as the root. The number of edges traversing by the ChangePath Interest equals to the number of nodes (i.e. N_{child}) on the sub-tree. The third procedure is optional and generates additional traffic when the Job Tree Rebuild Phase involves downstream-Rebuilder node(s). For example, node 7 as a downstream-Rebuilder communicates with node 4 that is the Rebuilder in Figure 17 (c). Suppose N_{down} is the total number of downstream-Rebuilder connected to the Rebuilder and $D_{down(i)}^{Rebuilder}$ is the path distance between the $\text{downstream} - \text{Rebuilder}_{\,i}$ and the Rebuilder. To simply the overhead expression, the cost of each IoT network link is assumed to be the same and labelled as C_l. The total overhead occurred in the Job Tree Rebuild Phase (O_R) can be written as (s).

$$O_R = C_l * (D_{Rebuilder}^{Joint-Up} + N_{child} + \sum_{i=1}^{N_{down}} D_{down(i)}^{Rebuilder}) * (X_{RSC-Interest} + X_{RSC-Data}) \quad \text{(s)}$$

The overhead traffic in the Job State Sync Phase also includes three procedures at most. The first is the communication between the Rebuilder and the sink node. Let $D_{Rebuilder}^{Sink}$ denotes the path distance from the Rebuilder to the sink. If the sink node has already received the data sample(s) matched the ID(s) in the JobSync Interest,

this phase is considered finished and the remaining two procedures can be skipped. The second procedure is the enquiry between the sink node and Previous-Upstreamer of the Rebuilder or the upstream node of the Previous-Upstreamer which has received the processed data content after the link failure. Suppose $D_{Sink}^{Upstream}$ is the path distance between the sink node and the upstream node which can respond to the ForwardJobSync Interest. The third procedure is optional. It is for data re-transmission if finding any data samples missing in the previous procedures. The Rebuilder re-sends the specific data samples to the sink node. Thus, the overhead traffic in the Job State Sync Phase (O_S) can be written as (t).

$$O_S = C_l(D_{Rebuilder}^{Sink} + D_{Sink}^{Upstream} + D_{Rebuilder}^{Sink})(X_{RSC-Interest} + X_{RSC-Data}) \quad (t)$$

In the Job State Commit Phase, the sink node sends the notification to all other nodes on the job tree periodically. Suppose if N_{total} is the number of nodes on the job tree, there are (N_{total}-1) edges to transmit the Interest and Data packets in this phase. Let T_{total} denotes the time length of the current sink node issuing jobs on the job tree and t_{commit} as the frequency for the sink node to send the JobCompleted Interest. The overhead traffic in the Job State Commit Phase (O_C) can be written as (u).

$$O_C = C_l(N_{total}-1)\,(X_{RSC-Interest} + X_{RSC-Data})(T_{total}/t_{commit}) \quad (u)$$

The network traffic overhead of the proposed protocol is calculated as O_R+O_S+O_C. Observing expressions (s), (t) and (u), we can conclude three factors that affect the overhead. The first is the job tree size. Both the depth and width of the job tree decide the number of nodes required by the current job(s). A deeper and wider the job tree results in a larger value of N_{total} in equation (u), which increases the overall overhead traffic. The second factor is the pre-defined frequency for the sink node to send

notifications, i.e. t_{commit} in equation (u). For the same job running the same time on the job tree, a smaller the value of t_{commit} leads to more frequent invocations of the Job State Commit Phase. It results in a higher value of O_C which contributes to the whole overhead of the proposed protocol. The last factor is the node that experiences a link failure, i.e. the Rebuilder in the proposed design. The overhead traffic O_R in equation (s) is closely related to the number of messages that the Rebuilder sent in the Job Tree Rebuild phase, i.e. to find a new upstream node ($D_{Rebuilder}^{Joint-Up}$), to notify downstream neighbours of ID change (N_{child}) and the previous upstream neighbour of path change ($\sum_{i=1}^{N_{down}} D_{down(i)}^{Rebuilder}$). In addition, the distance between the Rebuilder and the sink node directly affects the overhead O_S in equation (t). A longer the distance requires more messages exchanged to complete the Job State Sync Phase.

5.3.2 Computation Record Storage Overhead

The intermediate state of job execution is saved at each node, with the sink node maintaining the JS Table and others having their corresponding CR Table. Let W_i represent the number of records for $node_i$ inserting into its local JS/CR Table per second and T_{clear} is the time length for waiting the notification of clearing records from the sink node. The number of records saved by all nodes for each clear-record-cycle (W_{ECE}) can be calculated as (v). It is easy to summarize that the overhead of computation record storage is decided by T_{clear}. A smaller the T_{clear} value implies that fewer records maintained by each node. However, it is worth to mention that reducing T_{clear} also results in entering the Job State Commit Phase more frequently, which increases the network traffic overhead. It is up to the sink node or IoT applications to decide the best T_{clear} value.

$$W_{ECE} = \sum_{i=1}^{N_{total}} (W_i * T_{clear}) \tag{v}$$

5.4 Evaluation

This section presents tests to verify the feasibility of the proposed design and evaluate its performance under different link failure scenarios. As the proposed design relies on a job-tree-based ID and a multiple-phase job execution scheme to assure the exactly once data computation, overhead analysis is conducted in terms of ID allocation (varying according to the tree depth), the job maintenance (occurred in Proposed-RSC phases), and intermediate state of job processing save at edge nodes.

Since there are no existing approaches specifically targeting the same problem addressed in this thesis, a benchmark solution is developed based on the checkpoint scheme. It is abbreviated as CP-Benchmark for clarity and its main idea is summarized as below:

Step-1. The sink node has the information of processing-capable devices in the network. It generates a job execution plan/graph before issuing computation tasks, which randomly picks the processing nodes and assigns data sources to subgroups accordingly. The sink node then notifies each selected processing node of the generated job graph.

Step-2. During the job execution, the sink node sends a checkpoint message periodically to all nodes on the job graph. Each node replies to the checkpoint message by sending its current state to the sink node, simulating the central and durable storage for checkpoint snapshots. The checkpoint is considered successful if all nodes report normal states. In case of any failure or error, the sink node initiates a recovery procedure to address the issue.

Step-3. In the event of a failure, the sink node randomly picks another device to replace the failed one and migrates the computation tasks on the newly selected node.

Step-4. The sink node instructs all nodes on the current job graph to rollback to the last checkpoint and restart. The process then returns to ***Step-2*** and repeats.

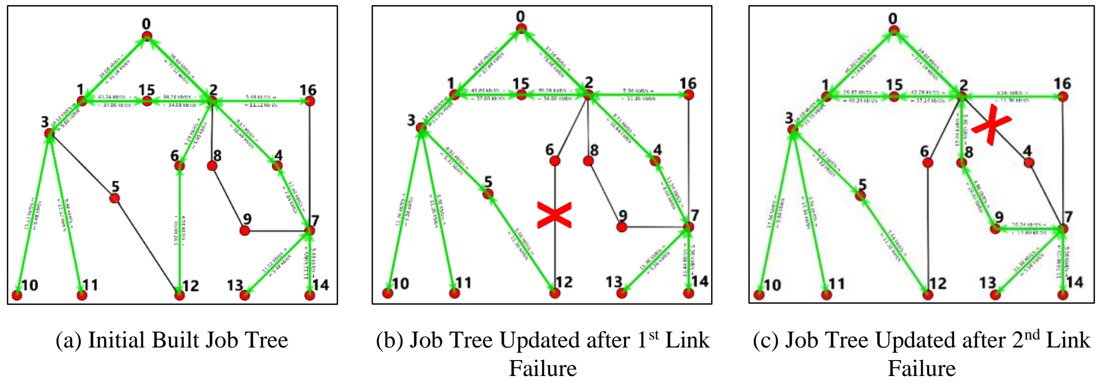
All tests are implemented on ndnSIM [80] which is specially designed for NDN. The following settings are applied to all tests: the sink/user node sends one task Interest per second. Mappers in the proposed design and CP-Benchmark data sources return a Data packet per received task Interest. Edge nodes process data samples every five seconds, which facilitates the ndnSIM simulator to capture link failure events. It can be flexibly set to meet the requirements of IoT applications. The network traffic is calculated by accumulating the number of transmitted Interest and Data packets by all nodes involved in the job tree/graph.

Two types of data transmission speed are set for the simulation, in the combination of bandwidth and delay: 250 Kbits per second plus 10 milliseconds based on the Zigbee [82] protocol between a mapper and a reducer/forwarder in the proposed design, and between a data source and a processing node for the CP-Benchmark solution. 54 Mbits per second and 1 millisecond using the IEEE 802.11 [83] parameter between reducers and forwarders for the proposed design, and between processing nodes of the CP-Benchmark.

5.4.1 Feasibility Test

To verify the functionality of the protocol design, a network topology is created in ndnSIM based on Figure 17 (a). Node 0 is configured as the user node and node 10-16 are set as mappers. Node 1-9 may act as a reducer or a forwarder or do not participate in data processing depending on their situations. The user node has a job request which consecutively issues 100 computational tasks. It also sends a JobCompleted Interest every 20 committed tasks to notify other nodes on the job tree to clear the corresponding history job records.

The links in the network topology have equal cost. The job tree is built using the NDN routing protocol with the shortest path algorithm. Link failures are defined to happen at different moments during the job execution: the first failed link is between node 6 (a forwarder) and node 12 (a mapper) and the second is between node 2 (a reducer) and node 4 (a forwarder).



(a) Initial Built Job Tree

(b) Job Tree Updated after 1st Link Failure

(c) Job Tree Updated after 2nd Link Failure

Figure 20 Job Tree Built and Updated by the Proposed_design

Figure 20 shows different job trees observed during the simulation: (a) is the initial job tree built with node 0 as the root, (b) is the updated job tree after the link between node 6 and 12 fails and (c) is the job tree after the second link failure happens between node 2 and 4. In the figures, each node is represented as a red dot, and the green lines indicate the edges on the job tree while the black ones are not currently used by the tree. The updated job trees demonstrate that the proposed protocol can handle link failures without suspending normal job execution procedure. Moreover, the final job result is received correctly without any data lost or duplicated processing.

Figure 21 reflects the transmitted traffic at each node during the test. Node 10, 11, 15 and 16 exhibit the same curve pattern, which are stable and repeat regularly. Since the four nodes are not affected by any network failures, they act as mappers to receive task requests and return data content. The peaks in their figures correspond to the periodic JobCompleted Interest sent in the Job State Commit Phase, occurring every 20

committed tasks.

After the first link failure happens, it causes more traffic for the following nodes. Firstly, the highest peak in the figure of node 12 in Figure 21 is the extra messages incurred in the Proposed-RSC phases to handle the link failure. Secondly, as node 6 only has one job neighbour (node 12) and after this link fails, it neither receives nor returns job data. Consequently, its curve remains at 0 after the first link failure. Thirdly, node 5 becomes the updated upstream job neighbour of node 12, it starts to transmit Interest and Data packets because of the rebuilt job tree. Lastly, the number of transmitted packets of node 3 increases after the first link failure because it now has an additional job neighbour (node 5), requiring it to send more CJ Interests and reply with more computed job results.

The second link failure forces node 4 to leave the job tree as it has no backup routes reaching the sink node, resulting its curve turning to 0 in Figure 21. Meanwhile, node 4 notifies the link failure to its child neighbour node 7 so that node 7 can try to find an alternative route without being affected. The rebuilt job tree enables node 7 to continue working on the job tree by adding node 8 and 9 as forwarders on the new path. Thus, the curve in the figure of node 8 and 9 respectively shows transmitted packets after the second link failure. Furthermore, the number of transmitted packets by node 7 grows as labelled by the red oval in its figure, which is the procedure initiated by node 7 to search alternative paths.

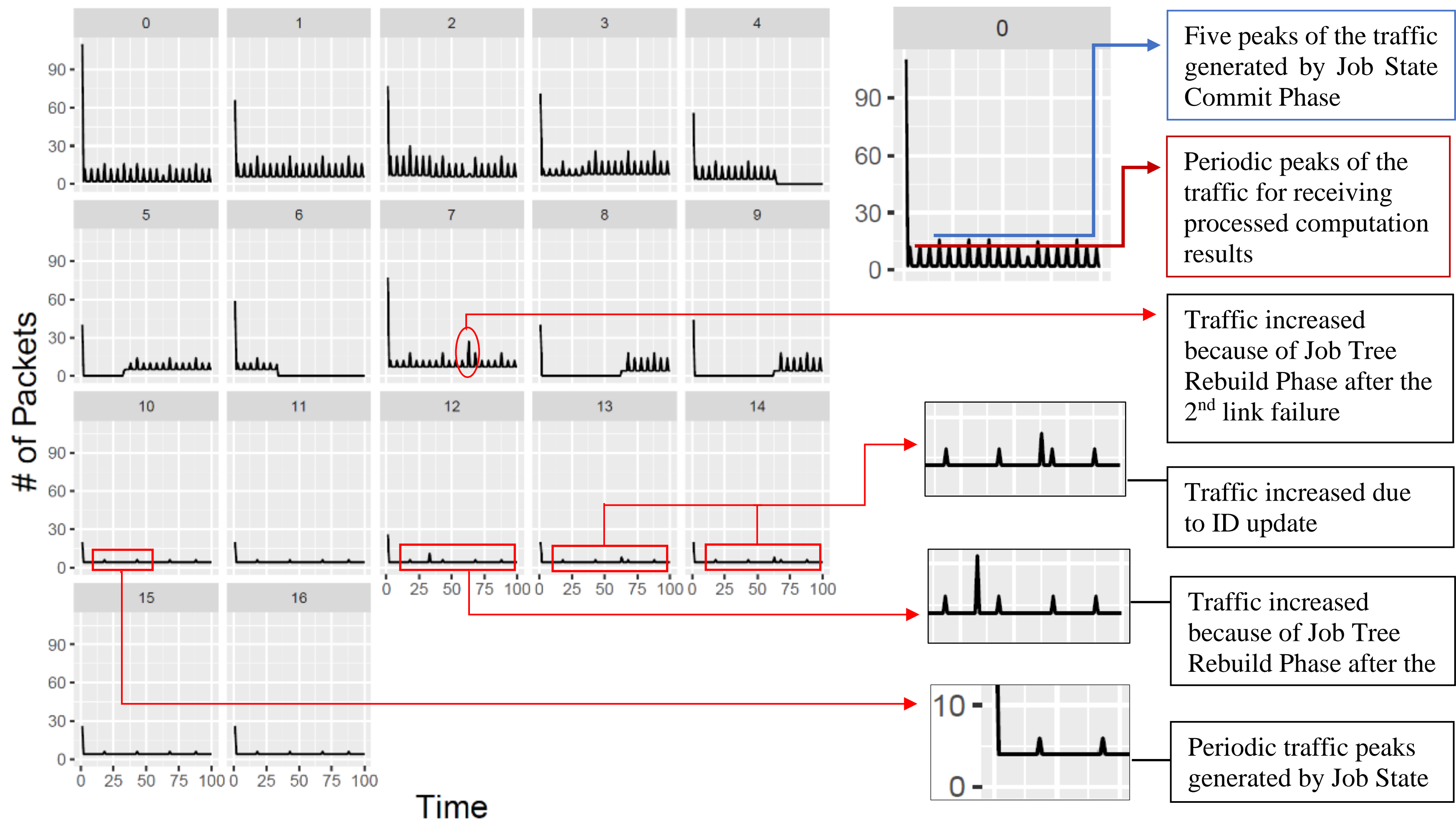


Figure 21 Traffic Generated by Nodes on Job Tree

The global ID of node 7 changes because its upstream neighbours on the job tree has been updated. It also changes the global ID of the child nodes of node 7. The highest peak in the figure of node 13 and 14 indicates the increased number of messages for the notification of updated global ID.

The second link failure forces node 4 to leave the job tree as it has no backup routes reaching the sink node, resulting its curve turning to 0 in Figure 21. Meanwhile, node 4 notifies the link failure to its child neighbour node 7 so that node 7 can try to find an alternative route without being affected. The rebuilt job tree enables node 7 to continue working on the job tree by adding node 8 and 9 as forwarders on the new path. Thus, the curve in the figure of node 8 and 9 respectively shows transmitted packets after the second link failure. Furthermore, the number of transmitted packets by node 7 grows as labelled by the red oval in its figure, which is the procedure initiated by node 7 to search alternative paths. The global ID of node 7 changes because its upstream neighbours on the job tree has been updated. It also changes the global ID of the child nodes of node 7. The highest peak in the figure of node 13 and 14 indicates the increased number of messages for the notification of updated global ID.

5.4.2 Network Traffic Overhead Analysis

To evaluate the network traffic overhead of the proposed protocol, a comparison is made with the CP-Benchmark approach. Two network topologies are created to assess the performance. In the tests, a job is defined as consecutively executing and completing 100 computational jobs. The sink node sends a JobCompleted Interest every 20 committed jobs in the test cases of the proposed design. As more network traffic is incurred by a higher checkpoint frequency, two checkpoint intervals are deployed for the CP-Benchmark tests, i.e. every 5 seconds and every 20 seconds.

- **Toy-Topology in Figure 17 (a)**

The network topology in Figure 17 (a) is created in ndnSIM for tests. Two failures are set during the job execution for the proposed design and CP-Benchmark solution respectively. Node 0 is the user node and node 10-16 are data sources. The remaining nodes act as edge devices and whether an edge node joins data processing depends on the job tree/graph generated by the protocol. CP-Benchmark randomly picks three edge nodes to undertake data processing and therefore the data sources are randomly separated into three groups.

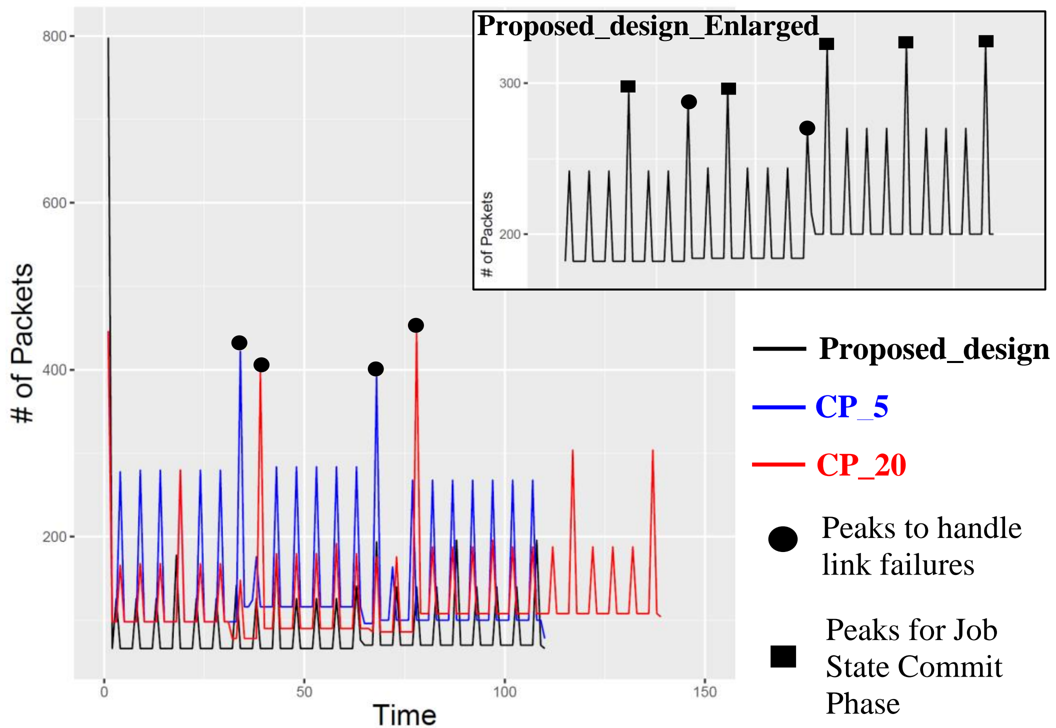


Figure 22 Network Traffic Comparison: Proposed_design Vs. CP_Benchmark

Figure 22 shows the test results, where the black curve represents the performance of the proposed design while the blue and red curve corresponds the CP-Benchmark with checkpoint interval of 5 seconds (CP_5) and 20 seconds interval (CP_20) respectively. At the beginning of the simulation, the highest peak of the proposed design is the number of messages exchanged by all nodes in the Job Tree Build Phase.

This phase incurs the most overhead in a round of job execution, as the job tree needs to be constructed for each new user node. In contrast, the CP-Benchmark approach assumes that the sink node possesses prior knowledge of network resources, resulting in a lower initial cost for generating the job graph compared to the proposed design.

During the job execution, it is evident that the network traffic generated by CP-Benchmark is consistently higher than that of the proposed design, regardless of the chosen checkpoint interval. This disparity can be attributed to the fact that CP-Benchmark does not consider the physical topology when generating the logical job plan. In the test scenario, the job graph created by CP-Benchmark assigns node 1 to process data samples from nodes 10, 13, 14, and 16, node 5 to handle nodes 11 and 12, and node 7 to manage node 15. However, transmitting raw data to edge nodes incurs a higher cost compared to directly sending data samples from the data source to the sink node. Consequently, in most cases, the distance between a data source and a processing node is longer than the direct path to the sink node, leading to increased network traffic.

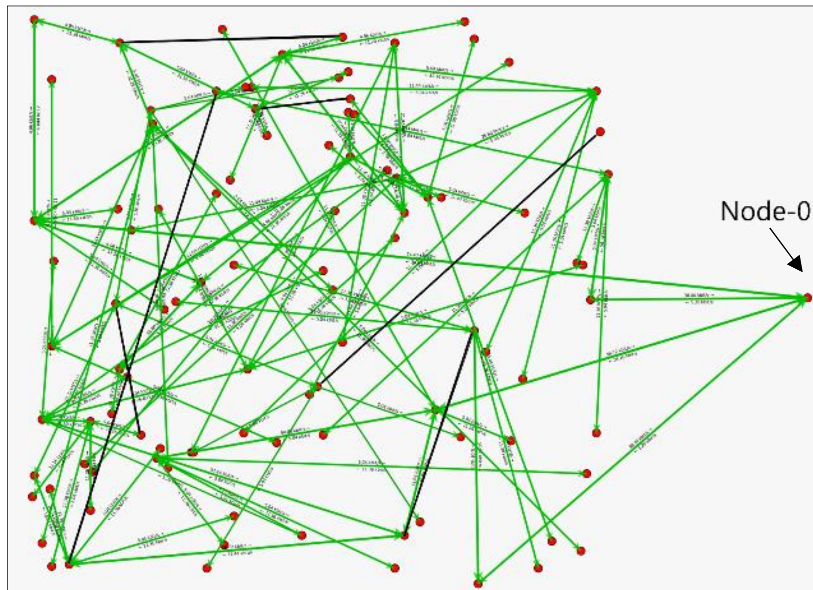
The peaks with a dot on the top of CP-Benchmark curves are the moments to handle link failures. During these instances, the network traffic spikes due to the sink node selecting another edge node for recovery and notifying all nodes on the job graph to roll back to the previous checkpoint state. The network traffic of CP-Benchmark with a 5-second checkpoint interval (blue curve) is higher than that of the 20-second interval (red curve) since checkpoint messages are transmitted more frequently throughout the job execution. Although this results in lower job execution latency and faster failure detection and recovery, the CP-Benchmark with a 20-second interval incurs a time cost approximately 30 seconds longer than both the 5-second interval and the proposed design, as indicated by the x-axis of Figure 22.

Figure 22 also includes an enlarged view of the curve of the proposed design, providing more details. The peaks with a dot at the top represent two link failures, where additional messages are exchanged to rebuild the job tree, synchronize job states and retransmit any lost data, according to the equation (s) and (t) described in previous section. The peaks with a square at the top are the moments of the JobCompleted Interest traversing all nodes on the job tree to clear historical job data, as described in equation (m). The job completion time of the proposed design is the same as CP-Benchmark with 5-second checkpoint interval.

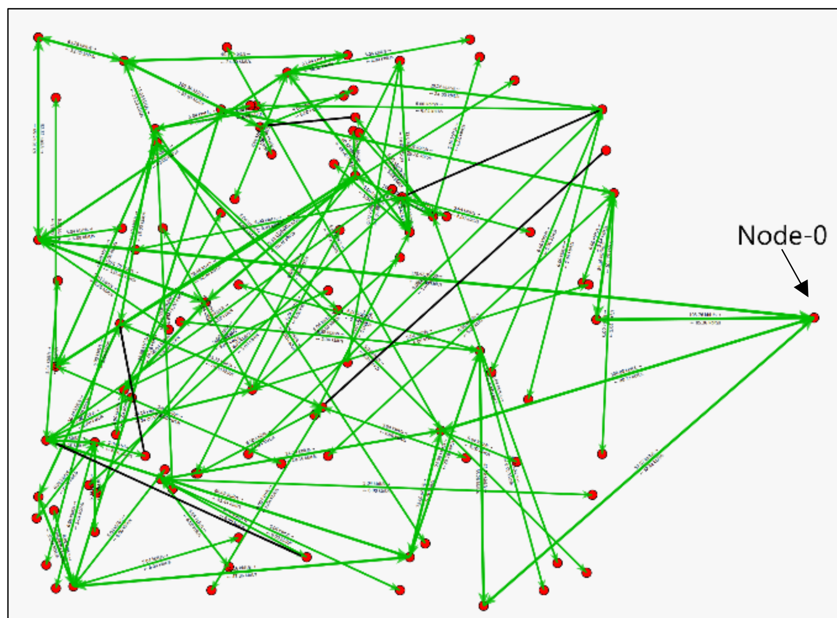
- **BRITE-Topology**

To evaluate the scalability of the proposed protocol, a network topology consisting of 100 nodes is generated by using BRITE [81] topology generator with RouterWaxman model. It is called BRITE-Topology for clarity. Node-0 is configured as the sink/user node. For the rest 99 nodes, 69 nodes (node number 31-99) act as mappers/data sources and 30 nodes serve as edge nodes. Five link failures are set during the simulation for the proposed design and the CP-Benchmark respectively.

Figure 23 (a) and (b) display the corresponding job graph generated by the proposed design and the CP-Benchmark approach. In these figures, red dots represent nodes, green lines with arrows are links used on the job graph and black lines depict original network links that are not used by the current job. The proposed design builds the job tree with node-0 as the root. CP-Benchmark randomly selects five edge nodes to undertake data computation tasks. All data sources are split into five groups and the number of nodes in each group ranges randomly from 5 to 15.



(a) Proposed_design Job Tree



(b) CP-Benchmark Job Graph

Figure 23 Job Graph on BRITE-Topology

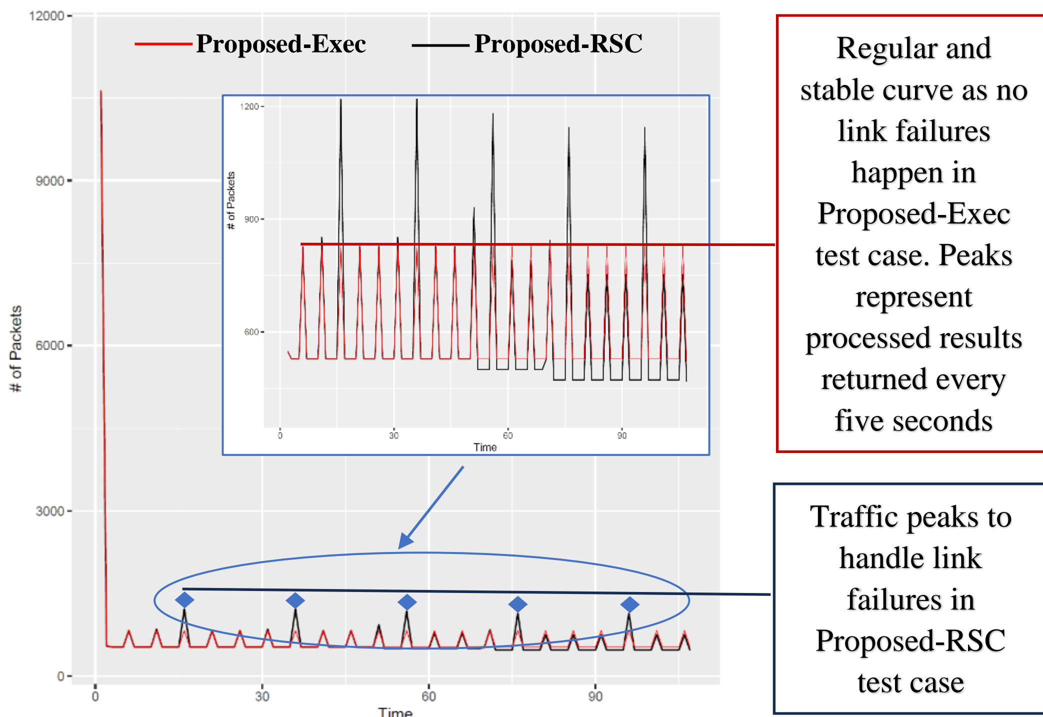
Figure 24 (a) presents the test results of the proposed design for completing the same job with/without failures. The Proposed-Exec curve (in red) represents the test case that no failures happen during the job execution. On the other hand, the Proposed-RSC curve (in black) shows the network traffic variation when the proposed design handles five failures during the job execution. Both curves have the

highest peak at the initial time of the test because of nodes exchanging the routing information to build the job tree.

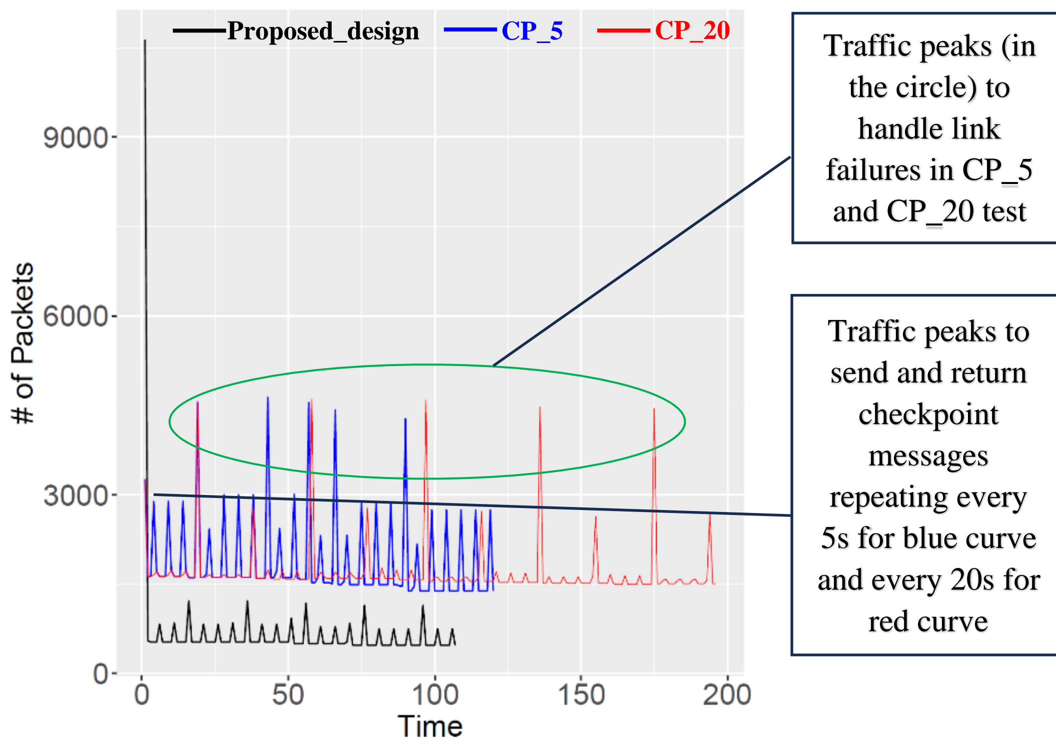
In the Proposed-Exec curve, there is a regular pattern of peaks and valleys every five seconds throughout the entire test, corresponding to the frequency at which reducers process data. The network traffic increases when the reducers return Data packets after processing. The black curve (Proposed-RSC) largely overlaps with the red curve for most of the simulation, indicating that the proposed design incurs limited additional cost to achieve exactly once data computation. The peaks with a blue diamond on top of the black curve represent the sink node sending JobCompleted Interests in the Job State Commit Phase. These peaks also contain the network traffic for the proposed design handling link failures, which explains the first two peaks are higher than the others in the zoomed view of Figure 24 (a). Observing the network traffic, the black curve is lower than the red one from approximately 50th second of the test. As link failures result in updated job trees, the number of Interest and Data packets decreases because of nodes changing their role during the job execution to aggregate multiple packets into one. For example, the number of Data packets may reduce if a node that was not on the job tree becomes a reducer and aggregates multiple job data content into a single Data packet.

The network traffic comparison between the proposed design and CP-Benchmark is shown in Figure 24 (b). CP-Benchmark with 5-second and 20-second checkpoint interval are respectively presented as the blue (CP_5) and red (CP_20) curve. The curve of the proposed design is in black, which is the same as the Proposed-RSC curve in Figure 24 (a) for more detailed information. As the number of nodes in the BRITE-Topology grows, the network traffic generated by the proposed design to build the job tree increases consequently. This increase in network traffic is even

more pronounced in CP-Benchmark, which always transmits more packets than the proposed design to complete the same job.



(a) Overhead Analysis of Proposed_design



(b) Proposed_design Vs. CP-Benchmark

Figure 24 Network Traffic Comparison on BRITE-Topology

As IoT networks grow, data transmission from data sources to processing nodes becomes a significant contributor to overall network traffic, especially if the physical topology is ignored during job assignment, as is the case in CP-Benchmark's random grouping of data sources with edge nodes. Additionally, using a checkpoint-based scheme to ensure exactly once data computation can introduce noticeable delays in job completion, as seen in the red curve in Figure 24 (b), potentially doubling the job execution time.

5.4.3 Overhead of Computation Record Storage

For evaluation purposes, the Clear-Record-Frequency (CRF) is defined as the number of completed tasks to clear all history records once. The job tree built in Figure 20 (a) is applied. In this simulation, a job is defined as consecutively executing and completing 200 tasks. The sink node sends a JobCompleted Interest with CRF values of 50, 20 and 10 respectively for the same job.

Figure 25 shows the number of records saved at each node with different CRF settings. The red curves represent CRF = 50, the green curves are for CRF = 20 and the blue ones for CRF = 10. The black lines in the figure track the network traffic associated with the job tree building and job execution processes, which are the same as the Proposed-Exec results discussed in previous section. As node 5, 8 and 9 are not on the job tree, they neither transmit job data nor save computation records.

The number of saved records can be categorized into two types: the test results of mappers (node 10-16) and the rest nodes. For the mappers, the curves exhibit a repeating pattern based on the CRF settings. With CRF = 50, the red curve increases from 0 to 50 and then drops back to 0. Similarly, the green curve rises from 0 to 20 and then returns to 0 with CRF = 20, and the blue curve follows a cycle of 0 to 10 to 0 with CRF = 10. The curves of mappers show a smooth growth pattern for all CRF

settings because mappers promptly reply to received CJ Interests and add a job computation record after returning each Data packet. The number of transmitted packets for actual job execution remains constant at 2, which includes one Data packet and one received Interest packet per second in the Job Execute Phase.

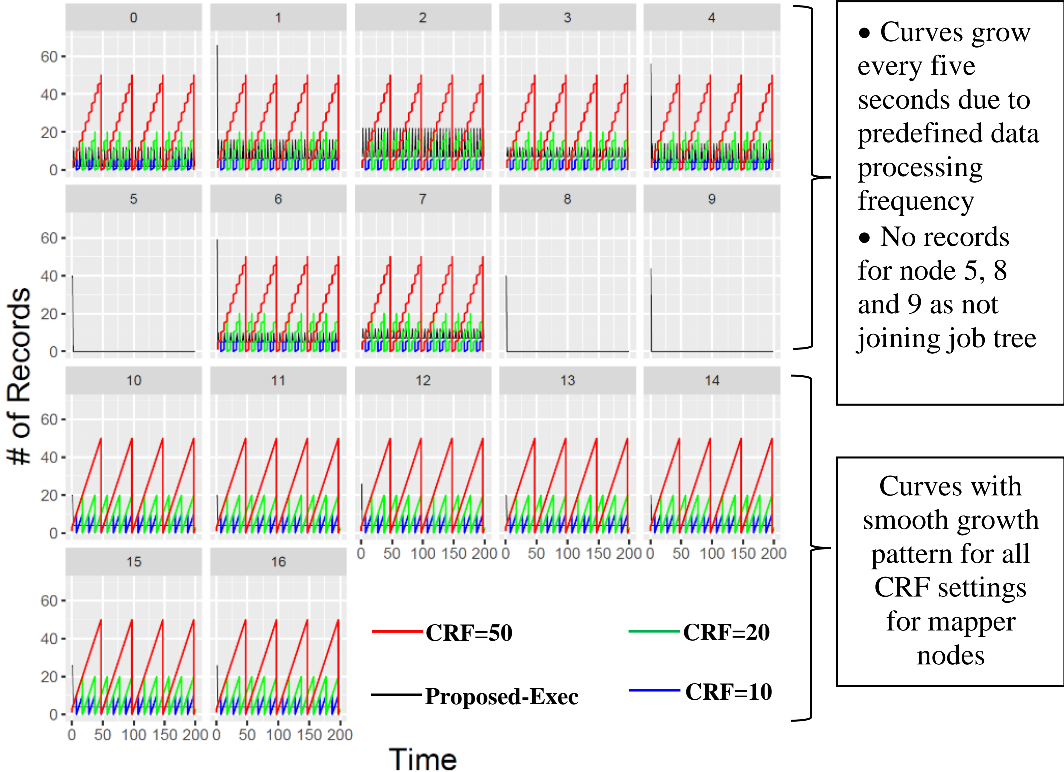


Figure 25 Overhead of Job Computation Records Storage

For the other nodes (node 0, 1-4, 6 and 7), the curves experience growth every five seconds due to the predetermined data processing frequency of reducers and forwarders. The number of saved job computation records is cleared every 10/20/50 completed jobs depending on the CRF setting. The curves representing job execution packets remain consistent and are unaffected by changes in CRF. These test results align with the conclusion stated in equation (n) from the previous section: a larger CRF value leads to a higher number of records maintained by all nodes on the job tree. The choice of the optimal CRF setting depends on the specific requirements of the IoT applications.

5.4.4 *Overhead of ID Allocation and Update*

Two network topologies are created in Figure 26 to compare the cost of ID allocation and update in the proposed design, which is influenced by the depth of the job tree. The two initial job trees, i.e. Job-Tree-A and Job-Tree-B, differ only in the number of intermediate nodes between the sink node and the mappers.

During the simulation, each job tree runs for 100 seconds and three link failures are configured at the 32nd, 62nd and 82nd second respectively. For Job-Tree-A, the failed links occur in the following temporal order: the link between node 2 and m3, the link between node 3 and m4, and the link between node 1 and m2. For Job-Tree-B, the link failures occur in the following order: the links between node 8 and m3, node 9 and m4, and node 7 and m2. The updated job trees after the three link failures are also depicted in Figure 26, with red dashed lines indicating the failed links.

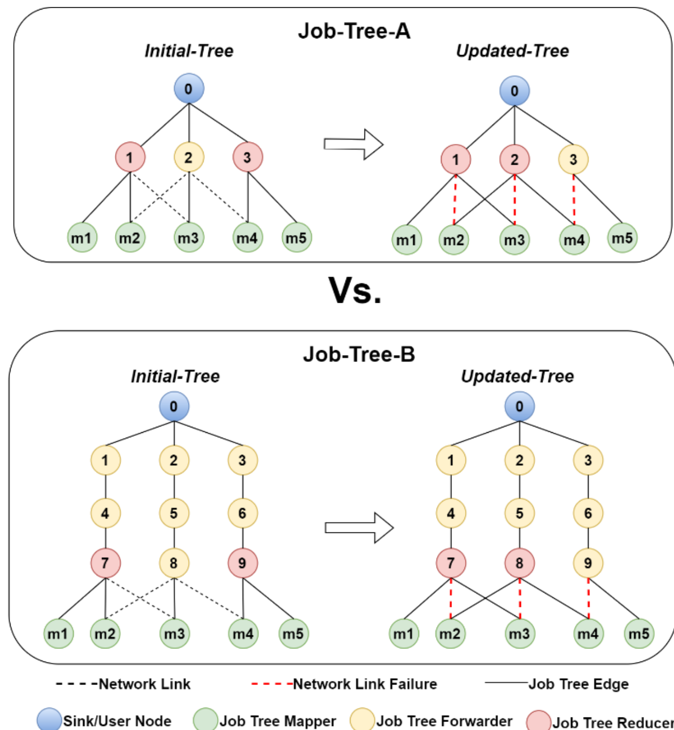
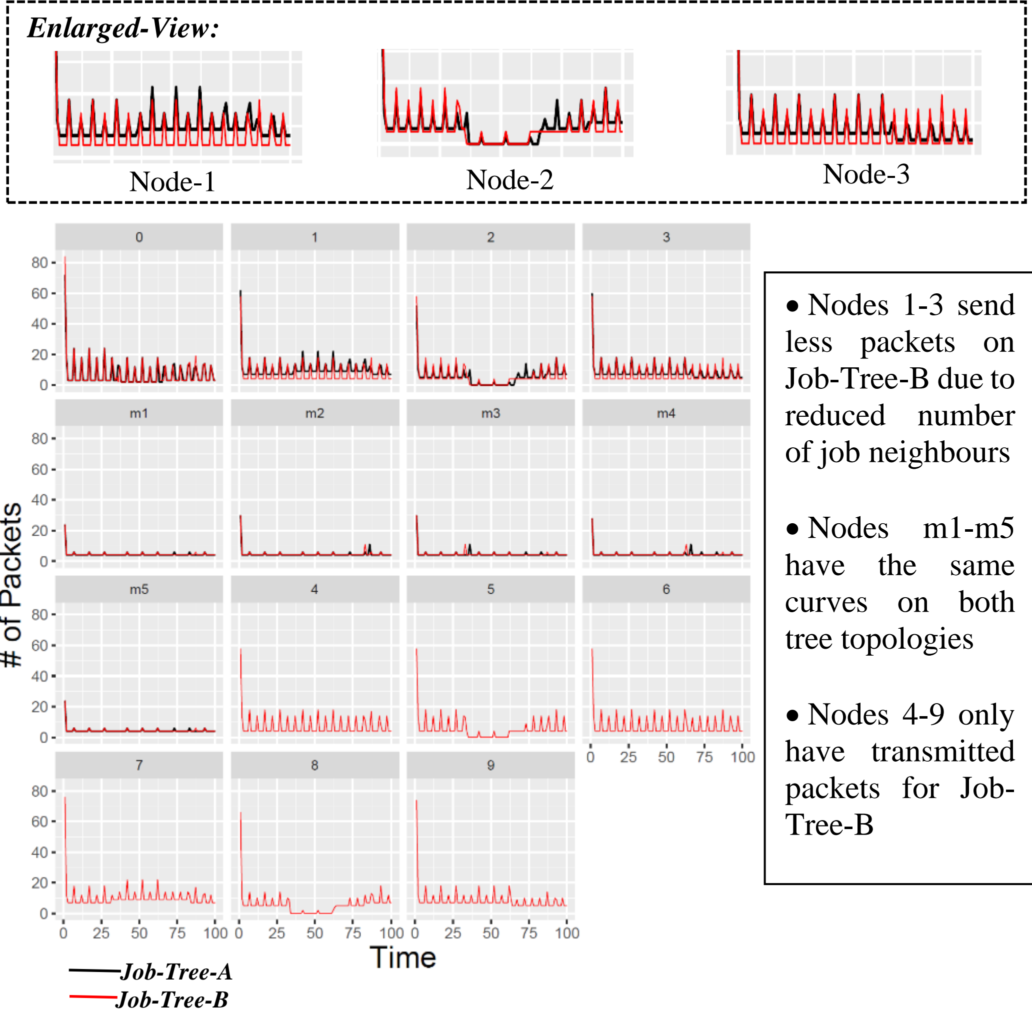
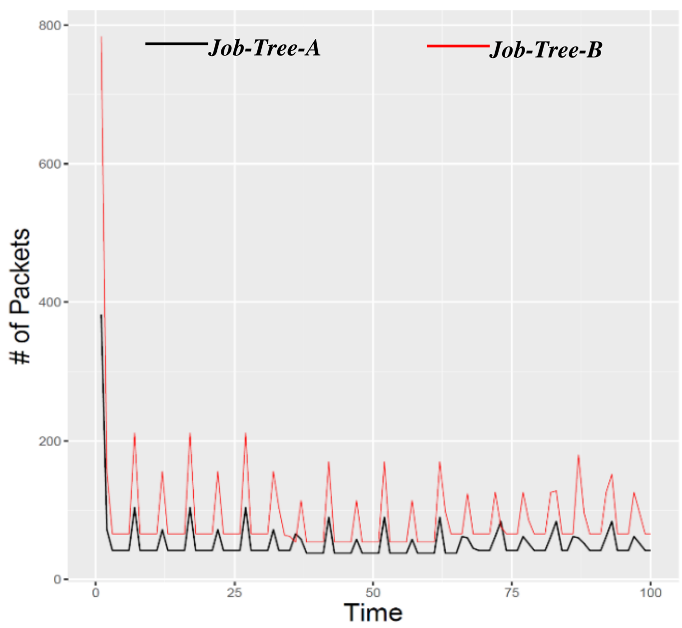


Figure 26 Node ID affected by Job Tree Depth



(a) Cost at each node to update ID



(b) Total Cost of Nodes' ID Update

Figure 27 Overhead of ID Update

The number of transmitted packets by each node over the simulation time is presented in Figure 27 (a), while the total network traffic shows in Figure 27 (b). The black curves represent the test data generated on the Job-Tree-A and the red curves are for Job-Tree-B. Nodes 4-9 only have transmitted packets for Job-Tree-B. For mapper nodes m1 and m5, the curves are the same for both job tree topologies, as the link failures have no effect on their job execution procedure. However, there is a slight difference in the number of packets for mapper nodes m2-m4. This difference arises because the transition from Job-Tree-A to Job-Tree-B introduces more traffic in the Job State Sync Phase with more intermediate nodes involved in forwarding Interest and Data packets. It worths to mention that the transmitted packets by mapper nodes m2-m4 in other phases of the proposed protocol remain the same.

For node 1-3, they disseminate less job requests on Job-Tree-B compared to Job-Tree-A because they are only responsible for one downstream neighbour on Job-Tree-B. The CJ Interest in the Job Execute Phase is sent per job node so that having more downstream neighbours introduces more traffic, which is doubled with returned job Data packets.

The total cost of the entire job tree is illustrated in Figure 27 (b). The number of transmitted packets increases by approximately two times when changing from Job-Tree-A to Job-Tree-B. This increase in cost aligns with the equation (s) discussed in the previous section (chapter 5.3.1). Additionally, the cost incurred by Job-Tree-B involves nodes leaving or re-joining the job tree due to the absence of downstream neighbours or the connection of new downstream neighbour(s), indicated by the variable $D_{down(i)}^{Rebuilder}$ in equation (s). For instance, when the link between node 8 and m3 fails, m3 finds a new path through node 7 on the job tree. Simultaneously, node 8, finding no job neighbours available after losing m3, leaves the job tree by notifying

node 5 about the situation. The same actions are taken by both node 5 and 2. When the second link failure happens between node 9 and m4, m4 sends a re-join request to node 8. To this end, nodes 8, 5 and 2 initiate the re-join tree procedure one by one until receiving the reply from the sink node, indicated by the variable $D_{Rebuilder}^{Joint-Up}$ in equation (s). Thus, the number of packets transmitted to allocate and update node ID is closely tied to the tree topology as well as the specific node experiencing the link failure.

5.5 Summary

Collaborative edge computing is a data processing paradigm that employs multiple edge devices cooperating with each other to execute jobs for IoT applications. However, ensuring exactly once data computation in such scenarios is a challenge due to potential IoT network connection failures. These failures can lead to data losses or duplicated data transmissions and computations, which violate the exactly once computation requirement.

This thesis proposes a five-phase protocol as a solution, which is built upon the novel ICN architecture. The Job Tree Build Phase constructs a job graph in the form of a tree, with the sink/user node as the root. This phase is executed before running any jobs. The Job Execute Phase disseminates job requests and returns the computed job results in the form of NDN Interest and Data packets. Whenever a network failure happens during the job execution, the Job Tree Rebuild Phase and the Job State Sync Phase are invoked to update the job graph and ensure no data affected by the failures. Finally, the Job State Commit Phase is designed to notify all nodes on the job tree about the completed jobs and to perform any necessary cleanup. A set of tests have been performed to show the feasibility and scalability of the proposed protocol. This thesis also analyses the overhead associated with ID assignment and computation

information storage.

This thesis assumes that the completion time of the proposed protocol is shorter than the failure frequency of nodes. Consequently, the proposed protocol updates the job tree to eliminate failed links even in cases where links frequently transition between active and inactive states. This, however, leads to additional overhead to maintain the job tree. Future work will focus on refining the proposed solution to effectively handle lossy network types.

6 Conclusion and Future Work

6.1 Conclusion

This thesis presents an ICN based collaborative edge computing framework for IoT data processing. The motivation behind this research is twofold. Firstly, with the rapid growth of IoT network, there is a large amount of and will be more data produced and exchanged by IoT devices. Processing data at the edge can help aggregate or filter data, reducing traffic volume before transmitting it to the cloud. Secondly, IoT applications usually require data to be processed at intermediate nodes situated between the user node and the data sources. The end-to-end communication model of current Internet does not adequately align with this requirement. Therefore, this thesis employs ICN as the underlying network support to enable IoT data communications and computation. Additionally, ICN fits well with the information-centric nature of IoT applications, whose users prefer more on content and service acquisition than establishing connections between multiple devices.

There are three contributions arising from the research work of this thesis.

Contribution I: Proposed a functional architecture for IoT collaborative edge computing and its ICN-based implementation for executing MapReduce jobs [22] [23].

The first research question of the thesis aims to explore the essential functional units required to support in-network data processing for IoT edge environments. With the increasing availability of computational resources in edge and IoT devices such as mobile phones, CCTV cameras and edge servers at base stations, there is an opportunity to offload data processing tasks from cloud servers to the edge. This can

help alleviate the burden on centralized servers and enable more efficient and timely data processing in IoT applications. These functional units may include data aggregation, filtering, analytics, and other computational capabilities. The goal is to identify the specific tasks and operations that can be performed within the network itself, closer to the data source, rather than relying solely on centralized cloud servers.

To address this research question, this thesis designs a functional architecture for IoT edge data computing to identify the responsibilities of edge devices by considering their heterogeneity nature. There are three components in the proposed architecture. The first component is the Computation Manager, responsible for administrating data processing and managing network devices. The second component is the Computation Executors, which refers to specific edge devices capable of contributing to the computational service. The third component is the Function Repository designed to save the processing functions, which can be maintained by a single device or deployed in the network in a distributed way. The functionality of each role is designed to undertake appropriate part of data processing and to cooperate with each other to complete the whole computation task.

To achieve efficiency, distributed data processing is more promising for big datasets than the centralized processing by a single server. For example, MapReduce framework is a powerful and popular tool in traditional big data processing. Thus, this thesis applies the MapReduce idea into the proposed design. The following research efforts contribute to the development of the proposed functional architecture for MapReduce job execution based on an ICN architecture (i.e., NDN).

Firstly, a NDN naming scheme is devised to express required data and processing logic in each user's request. More critically, the naming scheme assists job dissemination and function acquisition. Secondly, a computational job tree

construction protocol is proposed organize edge nodes for collaborative job execution. This protocol involves selecting appropriate edge nodes and distributing computational jobs among them. Thirdly, a job execution protocol is designed to parse users' requests and return the processed results. Experimental studies have been conducted to validate the proposed design. The results of these experiments demonstrate reduced network traffic compared with central-processing benchmark tests.

Contribution II: Developed the protocol to execute multiple MapReduce jobs on the proposed framework with the consideration of resource constraints on edge devices [25].

The second research question is to investigate the protocol that supports the simultaneous deployment and execution of multiple MapReduce jobs on the proposed framework, considering the resource constraints of heterogeneous edge devices, as some edge nodes have the computing resources to process data while others do not.

Broadly, this thesis divides edge devices into two types: processing-capable (acting as a mapper or reducer) and forwarding-only (acting as a forwarder) with the assumption that the procedure of matching the computational resource need of a job with the available computing resources at the devices has been completed. Each defined job contains the user-defined Map and Reduce tasks and desired datasets. Processing-capable nodes possess sufficient computing resources to perform a portion of current job. Specifically, the mappers directly connect with sensors/data sources, and they run user-defined map function on sensory data. The reducers execute the user-defined reduce function on the received data from neighbour nodes. Forwarding-only nodes, i.e. forwarders, do not need to parse or execute functions within the Interests. Their role is to receive multiple Data packets for the same job

tree, aggregate all received data samples into a single Data packet and then returns.

To enable collaboration between the processing-capable and forwarding-only edge nodes to complete each job, a job maintenance scheme is devised. The application layer functionalities are developed to support multiple job tree construction, e.g. managing job neighbours for different job trees, and to maintain multiple jobs running at the same time, e.g. ensuring processed results returned to the corresponding root/user node.

Contribution III: Developed the protocol to guarantee exactly once data computation on the proposed framework [27].

Fruitful research studies flourish in IoT collaborative edge computing area, with the focus on optimizing resource usage and task deployment. However, guaranteeing exactly once data computation in this context has not been thoroughly considered in edge computing scenarios. This is a critical aspect as network failures during edge collaboration can result in data loss or duplicated data transmission/processing, jeopardizing the exactly once computation requirements. Therefore, the third research question aims to address this challenge by achieving exactly once data computation in the proposed framework. This thesis identifies three specific challenges and proposes a five-phase protocol as a solution:

- Challenge-1. Backing up essential data processing information in distributed edge nodes.
- Challenge-2. Handling network failures during collaborative edge computing while guarantee exactly once computation on the same data.
- Challenge-3. Limited storage space at edge devices.

The proposed solution consists of five phases organised into two separate

procedures. The job execution procedure solves Challenge-1 and Challenge-3. It contains the Job Tree Build Phase, which constructs a tree-based job graph with the sink/user node as the root. The Job Execute Phase distributes job requests, returns computed data results, and saves essential data processing information in the form of "data sample ID - corresponding raw/computed content". The Job State Commit Phase is periodically launched by the root node to notify other nodes on the job tree about the state of the job(s), whether completed or uncompleted. This allows each device to delete their local records of specific completed jobs.

The job recovery procedure includes two phases and can coexist with the job execution procedure. It addresses Challenge 2 and is invoked when a network failure occurs during job execution. The Job Tree Rebuild Phase enables nodes experiencing link failures to explore alternative routes to reach the root node and replace the failed connections. Subsequently, the Job State Sync Phase synchronizes the data computation state, starting from the sink node and tracing back the previous data computation path to identify any lost data samples due to link failures.

Simulation experiments are developed to evaluate and compare the performance of the proposed design with a checkpoint-based benchmark solution, in terms of network traffic and job execution time. It also analyses the overhead associated with computation records storage and unique ID assignment.

6.2 Future Work

Currently there are some assumptions and limitations in the proposed design of this thesis, which points out the potential research directions in the future.

1. Improve the performance of the proposed design

The proposed ID format in ensuring exactly once computation embeds the

knowledge of all nodes that collect or compute the data, whose disadvantage is that the length of ID increases along with the job tree depth. It is worth to investigate an encoding/decoding algorithm to efficiently represent the ID information.

Computation information storage on edge devices causes more burden considering their resource constraints. The proposed design can be improved to equip with a scheme quantifying and collecting available storage space of edge nodes, which enables the sink node to decide the frequency of clearing historical records accordingly.

2. Optimize job deployment

The proposed computational job tree protocol is built on NDN routing protocol using the shortest path algorithm, which is one approach of the job deployment, i.e. selecting Computation Executor(s) closest to the required data sources. The criteria to establish a job tree should be tailored to meet the specific demands of each job. For instance, in the case of an IoT job with stringent latency requirements, the filtering of links based on available bandwidth can be employed. Metrics related to device capabilities and network resources can be integrated into the optimization objective for building the computational tree. Moreover, the partitioner function in traditional MapReduce framework can be integrated into current design, improving job processing efficiency.

3. Handle mobility of IoT devices

The proposed design assumes that IoT sensing devices are static. However, IoT applications, such as intelligent transport and vehicular networks, involve mobile devices. It results in frequent updates of the job tree in the proposed solution as IoT devices move, which incurs additional overhead to maintain the job tree. Future work will explore and develop a management scheme for IoT mobile scenarios, aiming at minimizing the cost of job tree maintenance.

Reference

[1] R. Abdmeziem and D. Tandjaoui, ‘Internet of Things: Concept, Building blocks, Applications and Challenges’. arXiv, Jan. 02, 2014. doi: 10.48550/arXiv.1401.6877.

[2] A. H. Mohd Aman, E. Yadegaridehkordi, Z. S. Attarbashi, R. Hassan, and Y.-J. Park, ‘A Survey on Trend and Classification of Internet of Things Reviews’, IEEE Access, vol. 8, pp. 111763–111782, 2020, doi: 10.1109/ACCESS.2020.3002932.

[3] ‘Internet of Things and data placement | Edge to Core and the Internet of Things | Dell Technologies Info Hub’. Accessed: Mar. 14, 2023. [Online]. Available: https://infohub.delltechnologies.com/l/edge-to-core-and-the-internet-of-things-2/internet-of-things-and-data-placement

[4] W. Yu et al., ‘A Survey on the Edge Computing for the Internet of Things’, IEEE Access, vol. 6, pp. 6900–6919, 2018, doi: 10.1109/ACCESS.2017.2778504.

[5] M. De Donno, K. Tange, and N. Dragoni, ‘Foundations and Evolution of Modern Computing Paradigms: Cloud, IoT, Edge, and Fog’, IEEE Access, vol. 7, pp. 150936–150948, 2019, doi: 10.1109/ACCESS.2019.2947652.

[6] A. Gaur, B. Scotney, G. Parr, and S. McClean, ‘Smart City Architecture and its Applications Based on IoT’, Procedia Computer Science, vol. 52, pp. 1089–1094, Jan. 2015, doi: 10.1016/j.procs.2015.05.122.

[7] A. Hazra, M. Adhikari, T. Amgoth, and S. N. Srirama, ‘A Comprehensive Survey on Interoperability for IIoT: Taxonomy, Standards, and Future Directions’, ACM Comput. Surv., vol. 55, no. 1, p. 9:1-9:35, Nov. 2021, doi: 10.1145/3485130.

[8] S. K. Fayazbakhsh et al., ‘Less pain, most of the gain: incrementally deployable ICN’, SIGCOMM Comput. Commun. Rev., vol. 43, no. 4, pp. 147–158, Aug. 2013, doi: 10.1145/2534169.2486023.

[9] Y. Wang, K. L. Man, K. Lee, D. Hughes, S.-U. Guan, and P. Wong, ‘Application of Wireless Sensor Network Based on Hierarchical Edge Computing Structure in Rapid Response System’, Electronics, vol. 9, no. 7, Art. no. 7, Jul. 2020, doi: 10.3390/electronics9071176.

[10] Y. Yu, S. Liu, P. L. Yeoh, B. Vucetic, and Y. Li, ‘LayerChain: A Hierarchical Edge-Cloud Blockchain for Large-Scale Low-Delay Industrial Internet of Things Applications’, IEEE Transactions on Industrial Informatics, vol. 17, no. 7, pp. 5077–5086, Jul. 2021, doi: 10.1109/TII.2020.3016025.

[11] B. Tang, Z. Chen, G. Hefferman, T. Wei, H. He, and Q. Yang, ‘A Hierarchical Distributed Fog Computing Architecture for Big Data Analysis in Smart Cities’, in Proceedings of the ASE BigData & SocialInformatics 2015, in ASE BD&SI ’15. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 1–6. doi: 10.1145/2818869.2818898.

[12] D. Ongaro and J. Ousterhout, ‘In Search of an Understandable Consensus Algorithm’, presented at the 2014 USENIX Annual Technical Conference (USENIX ATC 14), 2014, pp. 305–319.

[13] S. Kamburugamuve and G. Fox, Survey of Distributed Stream Processing. 2016. doi: 10.13140/RG.2.1.3856.2968.

[14] Z. Cai and T. Shi, ‘Distributed Query Processing in the Edge-Assisted IoT Data Monitoring System’, IEEE Internet of Things Journal, vol. 8, no. 16, pp. 12679–12693, Aug. 2021, doi: 10.1109/JIOT.2020.3026988.

[15] L. Liu, J. Zhang, S. H. Song, and K. B. Letaief, 'Client-Edge-Cloud Hierarchical Federated Learning', in ICC 2020 - 2020 IEEE International Conference on Communications (ICC), Jun. 2020, pp. 1–6. doi: 10.1109/ICC40277.2020.9148862.

[16] A. Javed, J. Robert, K. Heljanko, and K. Främling, 'IoTEF: A Federated Edge-Cloud Architecture for Fault-Tolerant IoT Applications', J Grid Computing, vol. 18, no. 1, pp. 57–80, Mar. 2020, doi: 10.1007/s10723-019-09498-8.

[17] M. A. P. Putra, A. R. Putri, A. Zainudin, D.-S. Kim, and J.-M. Lee, 'ACS: Accuracy-based client selection mechanism for federated industrial IoT', Internet of Things, vol. 21, p. 100657, Apr. 2023, doi: 10.1016/j.iot.2022.100657.

[18] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, 'Apache Flink™: Stream and Batch Processing in a Single Engine', Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, vol. 36, no. 4, pp. 28–38, 2015.

[19] 'Apache Kafka', Apache Kafka. Accessed: Feb. 01, 2023. [Online]. Available: https://kafka.apache.org/0102/documentation/streams/architecture

[20] P. Karhula, J. Janak, and H. Schulzrinne, 'Checkpointing and Migration of IoT Edge Functions', in Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking, in EdgeSys '19. New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 60–65. doi: 10.1145/3301418.3313947.

[21] F. Aïssaoui, G. Cooperman, T. Monteil, and S. Tazi, 'Smart scene management for IoT-based constrained devices using checkpointing', in 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), Oct. 2016, pp. 170–174. doi: 10.1109/NCA.2016.7778613.

[22] Q. Wang, B. Lee, N. Murray, and Y. Qiao, 'IProIoT: An in-network processing framework for IoT using Information Centric Networking', in 2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN), Jul. 2017, pp. 93–98. doi: 10.1109/ICUFN.2017.7993754.

[23] Q. Wang, B. Lee, N. Murray, and Y. Qiao, 'MR-IoT: An information centric MapReduce framework for IoT', in 2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC), Jan. 2018, pp. 1–6. doi: 10.1109/CCNC.2018.8319184.

[24] J. Dean and S. Ghemawat, 'MapReduce: simplified data processing on large clusters', Commun. ACM, vol. 51, no. 1, pp. 107–113, Jan. 2008, doi: 10.1145/1327452.1327492.

[25] Q. Wang, B. Lee, N. Murray, and Y. Qiao, 'MR-Edge: a MapReduce-based Protocol for IoT Edge Computing with Resource Constraints', in 2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC), Jan. 2019, pp. 1–6. doi: 10.1109/CCNC.2019.8651855.

[26] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, 'Wireless sensor networks: a survey', Computer Networks, vol. 38, no. 4, pp. 393–422, Mar. 2002, doi: 10.1016/S1389-1286(01)00302-4.

[27] Q. Wang, B. Lee, N. Murray, and Y. Qiao, 'ECE: Exactly Once Computation for Collaborative Edge in IoT using Information Centric Networking', IEEE Internet of Things Journal, pp. 1–1, 2023, doi: 10.1109/JIOT.2023.3275179.

[28] 'Structured Streaming Programming Guide - Spark 3.3.0 Documentation'. Accessed: Jul. 19, 2022. [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[29] A. Afanasyev, J. Burke, T. Refaei, L. Wang, B. Zhang, and L. Zhang, ‘A Brief Introduction to Named Data Networking’, in MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM), Oct. 2018, pp. 1–6. doi: 10.1109/MILCOM.2018.8599682.

[30] L. Zhang et al., ‘Named data networking’, ACM SIGCOMM Computer Communication Review, vol. 44, no. 3, pp. 66–73.

[31] I. Polato, R. Ré, A. Goldman, and F. Kon, ‘A comprehensive view of Hadoop research—A systematic literature review’, Journal of Network and Computer Applications, vol. 46, pp. 1–25, Nov. 2014, doi: 10.1016/j.jnca.2014.07.022.

[32] N. Pansare, V. Borkar, C. Jermaine, and T. Condie, ‘Online aggregation for large MapReduce jobs’, Proc. VLDB Endow., vol. 4, no. 11, pp. 1135–1145, Aug. 2011, doi: 10.14778/3402707.3402748.

[33] F. Pianese, ‘Information Centric Networks for Parallel Processing in the Datacenter’, in 2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops, Jul. 2013, pp. 208–213. doi: 10.1109/ICDCSW.2013.67.

[34] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O’Shea, and A. Donnelly, ‘Symbiotic routing in future data centers’, in Proceedings of the ACM SIGCOMM 2010 conference, in SIGCOMM ’10. New York, NY, USA: Association for Computing Machinery, Aug. 2010, pp. 51–62. doi: 10.1145/1851182.1851191.

[35] C. Hardy, E. L. Merrer, and B. Sericola, ‘Distributed deep learning on edge-devices: feasibility via adaptive compression’, presented at the IEEE International Symposium on Network Computing and Applications, Cambridge, MA USA, Nov. 2017. doi: 10.48550/arXiv.1702.04683.

[36] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, ‘A Survey on Distributed Machine Learning’, ACM Comput. Surv., vol. 53, no. 2, p. 30:1-30:33, Mar. 2020, doi: 10.1145/3377454.

[37] M. Yu, A. Liu, N. N. Xiong, and T. Wang, ‘An Intelligent Game-Based Offloading Scheme for Maximizing Benefits of IoT-Edge-Cloud Ecosystems’, IEEE Internet of Things Journal, vol. 9, no. 8, pp. 5600–5616, Apr. 2022, doi: 10.1109/JIOT.2020.3039828.

[38] Y. Bi, G. Han, C. Lin, Q. Deng, L. Guo, and F. Li, ‘Mobility Support for Fog Computing: An SDN Approach’, IEEE Communications Magazine, vol. 56, no. 5, pp. 53–59, May 2018, doi: 10.1109/MCOM.2018.1700908.

[39] X. Qiu, L. Liu, W. Chen, Z. Hong, and Z. Zheng, ‘Online Deep Reinforcement Learning for Computation Offloading in Blockchain-Empowered Mobile Edge Computing’, IEEE Transactions on Vehicular Technology, vol. 68, no. 8, pp. 8050–8062, Aug. 2019, doi: 10.1109/TVT.2019.2924015.

[40] B. Wang, M. Li, X. Jin, and C. Guo, IEEE Access, vol. 8, pp. 46373–46399, 2020, doi: 10.1109/ACCESS.2020.2979022.

[41] B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas, and Q. Zhang, ‘Edge Computing in IoT-Based Manufacturing’, IEEE Communications Magazine, vol. 56, no. 9, pp. 103–109, Sep. 2018, doi: 10.1109/MCOM.2018.1701231.

[42] L. Yuan et al., ‘CoopEdge: A Decentralized Blockchain-based Platform for Cooperative Edge Computing’, in Proceedings of the Web Conference 2021, in WWW ’21. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 2245–2257. doi: 10.1145/3442381.3449994.

[43] H. Jin, L. Jia, and Z. Zhou, ‘Boosting Edge Intelligence With Collaborative Cross-Edge Analytics’, IEEE Internet of Things Journal, vol. 8, no. 4, pp. 2444–2458, Feb. 2021, doi: 10.1109/JIOT.2020.3034891.

[44] X. Masip-Bruin et al., ‘mF2C: towards a coordinated management of the IoT-fog-cloud continuum’, in Proceedings of the 4th ACM MobiHoc Workshop on Experiences with the Design and Implementation of Smart Objects, in SMARTOBJECTS ’18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 1–8. doi: 10.1145/3213299.3213307.

[45] B. Nour et al., ‘A survey of Internet of Things communication using ICN: A use case perspective’, Computer Communications, vol. 142–143, pp. 95–123, Jun. 2019, doi: 10.1016/j.comcom.2019.05.010.

[46] R. Ravindran, P. Suthar, D. Trossen, C. Wang, and G. White, ‘Enabling ICN in 3GPP’s 5G NextGen Core Architecture’, Internet Engineering Task Force, Internet Draft draft-irtf-icnrg-5gc-icn-04, Jan. 2021.

[47] C. Westphal et al., ‘Adaptive Video Streaming over Information-Centric Networking (ICN)’, Internet Engineering Task Force, Request for Comments RFC 7933, Aug. 2016. doi: 10.17487/RFC7933.

[48] X. Li, A. Wang, W. Wang, D. Kutscher, and Y. Wang, ‘Distributed architecture for microservices communication based on Information-Centric Networking (ICN)’, Internet Engineering Task Force, Internet Draft draft-li-icnrg-damc-01, Aug. 2023.

[49] R. Ravindran et al., ‘Design Considerations for Applying ICN to IoT’, Internet Engineering Task Force, Internet Draft draft-irtf-icnrg-icniot-03, May 2019.

[50] M. Amadeo, C. Campolo, and A. Molinaro, ‘Information-centric networking for connected vehicles: a survey and future perspectives’, IEEE Communications Magazine, vol. 54, no. 2, pp. 98–104, Feb. 2016, doi: 10.1109/MCOM.2016.7402268.

[51] M. Amadeo, C. Campolo, and A. Molinaro, ‘Multi-source data retrieval in IoT via named data networking’, in Proceedings of the 1st ACM Conference on Information-Centric Networking, in ACM-ICN ’14. New York, NY, USA: Association for Computing Machinery, Sep. 2014, pp. 67–76. doi: 10.1145/2660129.2660148.

[52] C. Gündoğan, P. Kietzmann, T. C. Schmidt, and M. Wählisch, ‘HoPP: Robust and Resilient Publish-Subscribe for an Information-Centric Internet of Things’, in 2018 IEEE 43rd Conference on Local Computer Networks (LCN), Oct. 2018, pp. 331–334. doi: 10.1109/LCN.2018.8638030.

[53] C. Gündogan, P. Kietzmann, T. C. Schmidt, M. Lenders, H. Petersen, M. Wählisch, ‘Information-centric networking for the industrial IoT’, in Proceedings of the 4th ACM Conference on Information-Centric Networking, in ICN ’17. New York, NY, USA: Association for Computing Machinery, Sep. 2017, pp. 214–215. doi: 10.1145/3125719.3132099.

[54] C. Gündoğan, J. Pfender, P. Kietzmann, T. C. Schmidt, and M. Wählisch, ‘On the impact of QoS management in an Information-centric Internet of Things’, Computer Communications, vol. 154, pp. 160–172, Mar. 2020, doi: 10.1016/j.comcom.2020.02.046.

[55] P. Kietzmann, J. Alamos, D. Kutscher, T. C. Schmidt, and M. Wählisch, ‘Delay-tolerant ICN and its application to LoRa’, in Proceedings of the 9th ACM Conference on Information-Centric Networking, in ICN ’22. New York, NY, USA: Association for Computing Machinery, Sep. 2022, pp. 125–136. doi: 10.1145/3517212.3558081.

[56] C. Tschudin and M. Sifalakis, 'Named functions and cached computations', in 2014 IEEE 11th Consumer Communications and Networking Conference (CCNC), Jan. 2014, pp. 851–857. doi: 10.1109/CCNC.2014.6940518.

[57] M. Król, K. Habak, D. Oran, D. Kutscher, and I. Psaras, 'RICE: remote method invocation in ICN', in Proceedings of the 5th ACM Conference on Information-Centric Networking, in ICN '18. New York, NY, USA, Sep. 2018, pp. 1–11. doi: 10.1145/3267955.3267956.

[58] M. Król, S. Mastorakis, D. Oran, and D. Kutscher, 'Compute First Networking: Distributed Computing meets ICN', in Proceedings of the 6th ACM Conference on Information-Centric Networking, in ICN '19. New York, NY, USA, Sep. 2019, pp. 67–77. doi: 10.1145/3357150.3357395.

[59] W. Drira and F. Filali, 'NDN-Q: An NDN query mechanism for efficient V2X data collection', in 2014 Eleventh Annual IEEE International Conference on Sensing, Communication, and Networking Workshops (SECON Workshops), Jun. 2014, pp. 13–18. doi: 10.1109/SECONW.2014.6979698.

[60] O. Ascigil, S. Reñé, G. Xylomenos, I. Psaras, and G. Pavlou, 'A keyword-based ICN-IoT platform', in Proceedings of the 4th ACM Conference on Information-Centric Networking, in ICN '17. New York, NY, USA: Association for Computing Machinery, Sep. 2017, pp. 22–28. doi: 10.1145/3125719.3125733.

[61] M. Król and I. Psaras, 'NFaaS: named function as a service', in Proceedings of the 4th ACM Conference on Information-Centric Networking, in ICN '17. New York, NY, USA: Association for Computing Machinery, Sep. 2017, pp. 134–144. doi: 10.1145/3125719.3125727.

[62] M. Amadeo, C. Campolo, A. Molinaro, and G. Ruggeri, 'IoT Data Processing at the Edge with Named Data Networking', in European Wireless 2018; 24th European Wireless Conference, May 2018, pp. 1–6.

[63] T.-D. Nguyen, E.-N. Huh, and M. Jo, 'Decentralized and Revised Content-Centric Networking-Based Service Deployment and Discovery Platform in Mobile Edge Computing for IoT Devices', IEEE Internet of Things Journal, vol. 6, no. 3, pp. 4162–4175, Jun. 2019, doi: 10.1109/JIOT.2018.2875489.

[64] S. Mastorakis, A. Mtibaa, J. Lee, and S. Misra, 'ICedge: When Edge Computing Meets Information-Centric Networking', IEEE Internet of Things Journal, vol. 7, no. 5, pp. 4203–4217, May 2020, doi: 10.1109/JIOT.2020.2966924.

[65] Z. Fan, W. Yang, F. Wu, J. Cao, and W. Shi, 'Serving at the Edge: An Edge Computing Service Architecture Based on ICN', ACM Trans. Internet Technol., vol. 22, no. 1, p. 22:1-22:27, Oct. 2021, doi: 10.1145/3464428.

[66] B. Liang, J. Tian, and Y. Zhu, 'A Named In-Network Computing Service Deployment Scheme for NDN-Enabled Software Router', in 2022 5th International Conference on Hot Information-Centric Networking (HotICN), Nov. 2022, pp. 25–29. doi: 10.1109/HotICN57539.2022.10036191.

[67] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili, 'Collaborative Mobile Edge Computing in 5G Networks: New Paradigms, Scenarios, and Challenges', IEEE Communications Magazine, vol. 55, no. 4, pp. 54–61, Apr. 2017, doi: 10.1109/MCOM.2017.1600863.

[68] T. Akidau et al., 'MillWheel: fault-tolerant stream processing at internet scale', Proc. VLDB Endow., vol. 6, no. 11, pp. 1033–1044, Aug. 2013, doi: 10.14778/2536222.2536229.

[69] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach, ‘Exactly-once delivery in a content-based publish-subscribe system’, in Proceedings International Conference on Dependable Systems and Networks, Jun. 2002, pp. 7–16. doi: 10.1109/DSN.2002.1028881.

[70] M. Roohitavaf, K. Ren, G. Zhang, and S. Ben-romdhane, ‘LogPlayer: Fault-tolerant Exactly-once Delivery using gRPC Asynchronous Streaming’. arXiv, Nov. 25, 2019. doi: 10.48550/arXiv.1911.11286.

[71] ‘HDFS Architecture Guide’. Accessed: Jan. 26, 2023. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[72] F. Sun et al., ‘Recovery-oriented Big Data Computing for Exactly Once Message Processing’, in 2019 IEEE International Conference on Big Data (Big Data), Dec. 2019, pp. 2923–2930. doi: 10.1109/BigData47090.2019.9006585.

[73] ‘Messaging that just works — RabbitMQ’. Accessed: Apr. 17, 2023. [Online]. Available: https://www.rabbitmq.com/#features

[74] ‘Overview’, Docker Documentation. Accessed: Apr. 14, 2023. [Online]. Available: https://docs.docker.com/get-started/

[75] ‘Two-phase-commit-concepts’. Accessed: Jan. 26, 2023. [Online]. Available: https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/informix-servers/14.10?topic=protocol-precommit-phase

[76] ‘Two-phase commit protocol’, Wikipedia. Mar. 24, 2022. Accessed: Jul. 22, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Two-phase_commit_protocol&oldid=1078983413

[77] ‘An Overview of End-to-End Exactly-Once Processing in Apache Flink (with Apache Kafka, too!)’. Accessed: Jun. 30, 2023. [Online]. Available: https://flink.apache.org/2018/02/28/an-overview-of-end-to-end-exactly-once-processing-in-apache-flink-with-apache-kafka-too/

[78] W. Osamy, A. M. Khedr, A. Aziz, and A. A. El-Sawy, ‘Cluster-Tree Routing Based Entropy Scheme for Data Gathering in Wireless Sensor Networks’, IEEE Access, vol. 6, pp. 77372–77387, 2018, doi: 10.1109/ACCESS.2018.2882639.

[79] W. Qiu, E. Skafidas, and P. Hao, ‘Enhanced tree routing for wireless sensor networks’, Ad Hoc Networks, vol. 7, no. 3, pp. 638–650, May 2009, doi: 10.1016/j.adhoc.2008.07.006.

[80] S. Mastorakis, A. Afanasyev, and L. Zhang, ‘On the Evolution of ndnSIM: an Open-Source Simulator for NDN Experimentation’, SIGCOMM Comput. Commun. Rev., vol. 47, no. 3, pp. 19–33, Sep. 2017, doi: 10.1145/3138808.3138812.

[81] A. Medina, A. Lakhina, I. Matta, and J. Byers, ‘BRITE: an approach to universal topology generation’, in MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Aug. 2001, pp. 346–353. doi: 10.1109/MASCOT.2001.948886.

[82] S. Marksteiner, V. J. Exposito Jimenez, H. Valiant, and H. Zeiner, ‘An overview of wireless IoT protocol security in the smart home domain’, in 2017 Internet of Things Business Models, Users, and Networks, Nov. 2017, pp. 1–8. doi: 10.1109/CTTE.2017.8260940.

[83] S. Banerji and R. S. Chowdhury, ‘On IEEE 802.11: Wireless LAN Technology’, IJMNCT, vol. 3, no. 4, pp. 45–64, Aug. 2013, doi: 10.5121/ijmnct.2013.3405.

[84] B. Wang and J. C. Hou, 'Multicast routing and its QoS extension: problems, algorithms, and protocols', IEEE Network, vol. 14, no. 1, pp. 22–36, Jan. 2000, doi: 10.1109/65.819168.

[85] Y.-K. Huang, A.-C. Pang, P.-C. Hsiu, W. Zhuang, and P. Liu, 'Distributed Throughput Optimization for ZigBee Cluster-Tree Networks', IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 3, pp. 513–520, Mar. 2012, doi: 10.1109/TPDS.2011.192.

[86] K. Dev, P. K. R. Maddikunta, T. R. Gadekallu, S. Bhattacharya, P. Hegde, and S. Singh, 'Energy Optimization for Green Communication in IoT Using Harris Hawks Optimization', IEEE Transactions on Green Communications and Networking, vol. 6, no. 2, pp. 685–694, Jun. 2022, doi: 10.1109/TGCN.2022.3143991.

[87] C. Cicconetti, M. Conti, and A. Passarella, 'A Decentralized Framework for Serverless Edge Computing in the Internet of Things', IEEE Transactions on Network and Service Management, vol. 18, no. 2, pp. 2166–2180, Jun. 2021, doi: 10.1109/TNSM.2020.3023305.