ECE: Exactly-Once-Computation for Collaborative Edge in IoT using Information Centric Networking

Qian Wang, Brian Lee, Niall Murray, and Yuansong Qiao

***Abstract*—Exactly-once data processing/delivery can be guaranteed in traditional big data processing systems, e.g. Apache Flink. Checkpoint is commonly used as the solution. Each operator in these systems can restart from the last successfully saved state whenever a failure happens. It is not necessary to restore the logical job graph onto the same device(s) in traditional datacentre scenarios with powerful servers close to each other. However, the datacentre oriented solutions are not suitable for IoT collaborative edge computing scenarios. The logical job graph is tightly coupled to the physical topology in IoT networks. Data processing task(s) cannot be placed at a random edge device to recover from a network failure as it needs to evaluate the benefits of transmitting data versus processing/aggregating the data. To address the above challenges, this paper proposes an Information Centric Networking based solution and correspondent protocols to provide Exactly-once-computation for the Collaborative Edge in IoT (ECE). It contains a job execution scheme to deliver IoT jobs with exactly once data computation guarantee and a recovery procedure to dynamically change the IoT job execution graph while experiencing link failures. The protocol also provides a checking procedure on data state (received/un-received and computed/un-computed) to prevent any data loss or duplicated data processing due to the updated job graph. A data identification approach based on the job graph is devised to support the ECE functionality. A testbed has been developed on ndnSIM and the simulation results have verified the feasibility and scalability of ECE design. It also evaluates the overhead incurred by the ECE protocol to guarantee exactly once data computation.**

***Index Terms*—Collaborative Edge Computing, Exactly-once-computation, Internet of Things (IoT), Information Centric Networking (ICN)**

I. INTRODUCTION

THE Internet of Things (IoT) [1] enabled smart systems thrive in diverse areas. All of them rely on sensing devices to capture a vast amount of raw data from the physical world as the first step. Edge computing [2] [3] proposed to be complementary of Cloud Computing, has proved its ability to boost IoT Big Data processing by placing computation at the proximity of data sources. Researchers [4] further demonstrate that computation-intensive tasks, e.g. image processing and speech recognition, can benefit from the synergy of multiple edge devices than offloading to a single edge server. It is defined as collaborative edge computing [5] [6] which distributes data computation to multiple edge devices and coordinates them working together to complete the whole job(s).

Many complex IoT applications invoke the collaborative edge computing framework for better performance, such as the collaborative cross-edge analytics to preprocess training data for artificial intelligence (AI) IoT [7] and the hierarchical federated learning system with partial model aggregation deployed on edge servers [8]. Fruitful studies in this area have focused on optimizing resource usage and task deployment, handling network failures during job execution is not the main concern in their works. In fact, it may result in data loss or duplicated data transmission and/or processing if a network failure happens during the edge collaboration, which could end with wrong processed results or trained models.

This paper addresses the challenge of guaranteeing exactly-once-computation on the same data in collaborative edge scenarios. Existing works related to this topic is very scarce. Initial attempts utilise the checkpoint scheme to save the state of an IoT task into Docker images [9], concerning task migration from one edge device to another [10] and information transfer between different tasks [11]. However, their works are limited to task execution on a single edge device. Although checkpoint based solution has been maturely developed in traditional big data processing frameworks, e.g. Apache Spark [12] and Apache Flink [13], this paper argues that the solution is difficult to be applied into 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 close to each other. In sharp contrast, the logical job graph is tightly coupled to the physical topology in IoT edge environment. 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. Then the snapshots are normally saved to a durable storage, e.g. Hadoop Distributed File System (HDFS) [14], which is not widely available in edge computing environments.

Thus, this paper identifies the following challenges to achieve exactly once computation in collaborative edge

Manuscript received February 23, 2023; revised April 17, 2023; accepted April 30, 2020. Date of publication X-2023. This work was supported by the Technological University of the Shannon under the Staff Development Programme, and Science Foundation Ireland (SFI) under Grant Number SFI 16/RC/3918, co-funded by the European Regional Development Fund.
(*Corresponding author: Yuansong Qiao)*

Qian Wang, Brian Lee and Yuansong Qiao are with Software Research Institute (SRI), Technological University of the Shannon: Midlands Midwest, Athlone, Co. Westmeath, Ireland (email: qwang@research.ait.ie; blee@ait.ie; ysqiao@research.ait.ie)

Niall Murray is with Faculty of Engineering and Informatics, Technological University of the Shannon: Midlands Midwest, Athlone, Co. Westmeath, Ireland (email: nmurray@research.ait.ie)

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3275179

computing for IoT data processing.

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 requires to decide which information of data processing is essential and sufficient to be used to recover from the failures. Then it brings the challenge on how to save the information efficiently. Unlike the data centre environments, a 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 guarantee exactly once computation on the same data. When the network connection between two edge devices fails, it breaks the original job execution graph containing the two edge devices. The downstream edge is not sure if its data has been successfully delivered to its upstream neighbour. This requires designing a scheme to utilise 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 participant in the 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 distributedly saved on edge devices.

To address the challenges, this paper designs Exactly-once-computation for Collaborative Edge (ECE) protocol which consists of a job execution procedure (to solve Challenge-1 and Challenge-3) and a job recovery procedure (to solve Challenge-2). Some basic concepts are described to facilitate the introduction of the proposed design. As a continuous work of our previous one (MR-Edge) [15], the following keeps the same: (1) a tree topology is adopted as the job execution graph, with the device issuing jobs as the root, (2) a completed job state is defined as the final job results correctly computed by edge devices following the pre-built job tree and received by the root node, and (3) all communication between devices is realized in the way based on the Information Centric Networking (ICN) [16].

ECE job execution procedure is implemented by: (i) differentiating each (raw or computed) data sample to support the storage (Challenge-1) and deletion (Challenge-3) of job processing related information, and (ii) getting a consensus among devices on who process which data samples and when to delete which information. ECE devises a data identification (ID) approach which combines the job ID it belongs to and the device/node ID that has collected/computed the data. Specifically, the job ID is set by the root node before job dissemination. The node ID is uniquely created and updated along the data computation path on the job tree in a distributed manner, from the root node to each other node.

With the ID assignment available, ECE defines two types of information to answer Challenge-1, i.e. the data sample ID and its corresponding raw/computed content. Each node on the job tree saves the defined information in a pair as one record after they process. However, each node knows what data content it has computed but has no idea of the computation progress at other nodes and whether the job has completed, which is a reaching consensus problem in a distributed system. Inspired by the two-phase commit protocol [17], ECE job execution procedure contains two phases. The Job Execute Phase distributes job requests, returns computed data results and saves essential data processing information. The Job State Commit Phase is launched periodically by the root node to notify others on the job tree of the job(s) state, i.e. completed or uncompleted. Therefore, each device can delete their local records of specific completed jobs.

ECE job recovery procedure can coexist with the job execution procedure. It empowers nodes experiencing link failures to explore an alternative route (to reach the root node) to replace the failed one. The affected nodes can resume the job execution procedure on the updated job tree. Afterwards, the nodes interact with the root node to trace back their previous data computation path to check whether any data samples are lost due to link failures. The previous data computation path is obtained by decomposing the ID of the node that has just recovered from link failures. If data losses are found, the recovered nodes re-transmit the lost data sample(s) to the root node. Otherwise, no re-transmission is arranged so that duplicated data processing can be avoided.

To the best of the authors' knowledge, this is the first work to implement the exactly once data computation in IoT collaborative edge scenarios. The contributions of this paper are summarized as below:

(1) A job tree based ID assignment approach is devised to support the storage and deletion of data processing related information. The ID format embeds the knowledge of nodes that collect or compute the data, which assists checking on data loss or duplicated computation after recovering from network failures.

(2) A job execution procedure is proposed for nodes on the job tree to achieve a consensus on data processing plan and remove of processing related information with the exactly once data computation guarantee.

(3) A job recovery procedure is designed to handle link failures happened during the job execution, aiming to dynamically update the job tree to eliminate failed links. After the job tree is updated, synchronization on the data delivery (received or un-received) and computation (processed or un-processed) state is activated among affected data sources and edge devices.

(4) Simulation experiments are developed to evaluate and compare ECE performance with a checkpoint-based

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3275179

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.

The rest of this paper is organized as following: Section II presents the related work. Section III describes the protocol design in detail. The experimental setup and evaluation results are presented in Section IV. Section V concludes the paper and discusses the future work.

II. Related Works

A. Collaborative Edge in IoT

Various IoT applications benefit from the collaborative edge computing framework, such as less production order delivery time in industrial IoT [18] by self-organized task mechanism among multi-robots, a trustworthy framework for smart cities [19] and an edge assisted data monitoring system to minimize response latency and reduce cloud workload [20].

Despite the extensive research works on IoT edge computing, little work has considered guaranteeing exactly once data delivery and processing.

The solution proposed in [21] improves the message queue systems, e.g. Kafka [22] and RabbitMQ [23], to ensure exactly once processing through a consumer side protocol. All messages are stored in a shared databased and a state transition graph is introduced on each message to control access and operation. IoTEF [24] 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.

As described in the introduction section, checkpoint-based approach is applied to save the state of an IoT task as a container image in [10] [11] to facilitate task migration and restarting. However, the job execution is undertaken by a single edge device in these works. The traditional big data processing frameworks, e.g. Apache Spark [12] and Apache Flink [13], have employed the checkpoint-based schemes to achieve exactly once processing. However, the solution is not suitable for IoT edge environments. The main reason is that the logical job graph is tightly coupled with the physical job graph in IoT networks. The gain of data processing versus data transmission should be considered when mapping the logical job graph into the physical devices.

B. Distributed Consensus Protocol

To achieve exactly once computation in IoT collaborative edge, it is necessary to obtain a consensus on the data computation plan among the edge devices. The two-phase commit protocol [17] [25] is widely used in distributed systems to coordinate all parties to agree or abort an action. The two phases are 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 summarized as follows. In the commit-request phase, the coordinator sends a message to all participants asking to commit. 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. This paper is inspired by the two-phase commit protocol, which defines a Job Execute Phase for disseminating and executing jobs (i.e. the commit-request phase) and a Job State Commit Phase to commit the job completion state only if all nodes returning computed job results correctly (i.e. the commit phase).

C. Named Data Networking (NDN) Basics

The proposed design is implemented upon the NDN [26] architecture to meet the data/information centric nature of IoT applications. NDN uniquely identifies each data/content with a specific name and uses the name to retrieve and forward data. The naming is hierarchically constructed in NDN. For example, the first reading value of the humidity sensor in room 1 of the SRI office in the TUS campus can be named as /TUS/SRI/room1/humidity/reading1.

Communication in NDN is achieved by exchanging two packets: Interest and Data. A content consumer sends an Interest carrying the name of the desired data. A matched data/content is embedded in the Data packet and returned to the consumer in the reverse path of the Interest. This paper defines specific Interest naming for different phases of the protocol to support its functionalities of the respected phase.

NDN routers maintain three tables to facilitate data lookup and forwarding [27]. The first one is Content Store (CS) which caches the Data locally. If a matched Data is found in the CS of a NDN router, the Data is returned by the router directly. The second is Forwarding Information Base (FIB) which provides the name-based routing information. When a router receives an Interest packet, it will first check its CS. If it fails to find a matched Data, the router looks up its FIB to forward the Interest to the next hop matching the naming of Interest packet. The third table is Pending Interest Table (PIT). A router saves all received Interests waiting for the matched Data packet in its PIT. Each PIT entry includes the name of the Interest and all interfaces from which the Interest(s) is received. When multiple Interests for the same data are requested, the router only forwards the first one towards the data source. When a Data arrives, the router finds the matching PIT entry and returns the Data to the corresponding interface(s). Afterwards, the router deletes the PIT entry and caches the Data in its CS.

D. ICN based Edge Computing for IoT

The original design of ICN supports in-network data forwarding and caching while lacks the in-network processing functionality. To tackle this issue, the paper [16] assumes all edge nodes are able to process data and then the final execution placement depends on the trade-off between the data transmission and computing resource cost. Edge-ICN [28] facilitates the deployment of ICN in large network scale by leveraging SDN technology. The architecture proposed in [29] explores ICN-featured forwarding strategy to dynamically deploy edge services based on the service popularity. The main difference between this paper and the above works is to ensure the exactly once data computation in a distributed manner in the ICN style.

III. Protocol Design

This section presents ECE solution and its potential application scenarios.

A. Target IoT Scenarios

The proposed computing framework can serve many IoT applications requiring sensory data dispersed across a large area. While the IoT data is transmitted from data sources to the final job processor, the intermediate nodes (e.g. edge, network, and cloud devices) along the path may contribute their resources to execute computational task over the data passing through them. A hierarchical edge structure is usually formed to organize edge nodes with different powers undertaking (sub) tasks that matches their capabilities. Such as a four-layer fog computing architecture for big data analytic in smart cities [30], a three-tier edge computing paradigm for intelligent warehouse system [31] and a multi-layer IoT-Fog-Cloud continuum [32] with coordinated management strategies. In these systems, IoT end devices at the bottom layer could use Zigbee or Wi-Fi [33] to communicate with the edge server in their area. The communication between hierarchical edge servers (e.g. base stations and access points) can be achieved through LTE or 5G [34].

This paper is an improvement of our previous work MR-Edge, i.e. a MapReduce based computation framework for IoT edge computing environments [15]. The concerned computation jobs are those requiring processing the data from multiple static IoT end devices, such as temperature sensors and speed sensors on the road. The intermediate nodes that can process the data are called reducers which run user-defined reduce function on received data, whereas those cannot process the data but can forward the data are called forwarders. The stub nodes of IoT edge networks are called mappers, which connect with multiple sensors. They take raw sensing data as input and run user-defined map functions on the data.

B. ECE Protocol Overview and Assumptions

Fig.1 presents the relationship of the five phases in the design. Normal job operation is not disturbed by recovering from failures. The definition of each phase is listed as below:

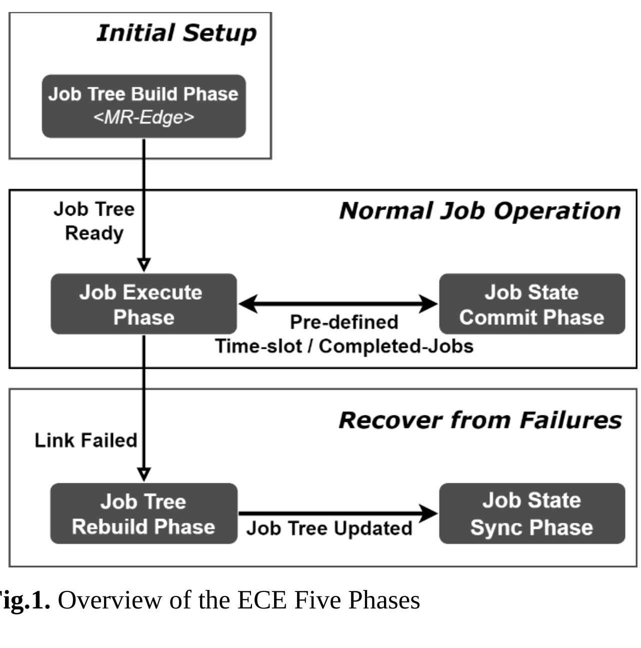


Fig.1. Overview of the ECE Five Phases

- **Job Tree Build Phase** forms a job tree with each new user as the root and the user could issue multiple jobs on its job tree.
- **Job Execute Phase** disseminates jobs requests, returns computed results and saves intermediate state of job processing.
- **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 link failures and the updated job tree cause neither data losses nor duplicated data computations.

As ECE is built upon NDN, the communication between nodes in all phases is achieved by exchanging the NDN Interest and Data. Different Interest naming schemes have been designed to facilitate the functionalities at each phase. The job tree is created using the shortest path algorithm of the NDN routing protocol. Additional metrics (e.g. link bandwidth [35], energy-efficiency [36]) can be considered when creating the job tree to optimize the performance, which is beyond the scope of this paper. The paper is also aware that the capabilities of the massive IoT devices are significantly different. Describing their resources and selecting the appropriate ones for IoT jobs [37] are not the main concern in this paper. Moreover, the protocol currently is limited to execute stateless jobs [38] whose output is solely based on its input, not the intermediate computational states. Specifically, the same computation on the same data can be undertaken by any capable edge devices. The computed result is only related to the number of input values rather than the order of them. To this end, the data computation can be recovered from a changed job tree due to link failures.

The following sections will describe each phase in detail.

C. Job Tree Build Phase

A tree topology is built with a user node (sink node) as the root node before it issues jobs in the proposed framework. This procedure is called the Job Tree Build Phase. The job tree is formed based on the NDN routing table which employs the shortest path algorithm. Every node has its own table so that it knows how to reach a specific node from itself. However, a node may have no idea of the routing information of other nodes. All nodes need to exchange their information to form a tree, which is achieved by sending NDN Interest and replying NDN Data packets.

A BuildJobTree Interest is defined for the Job Tree Build Phase and written as below:

/NeighborName/BuildJobTree/JobTreeID/UpstreamNodeName (a)

Where: (1) */NeighborName* is the name of each neighbour of the current node. (2) */BuildJobTree* is the identifier to trigger the procedure of building job trees. (3) */JobTreeID* is the combination of the name of the root node and a random number. (4) */UpstreamNodeName* is the name of the current node, which is used to for the downstream neighbours to identify the sender of this Interest.

The sink node initiates this phase by creating and sending a BuildJobTree Interest. The reducers and forwarders modify the "*UpstreamNodeName*" part and then forward it to their neighbours, until reaching the mappers. After received a

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3275179

BuildJobTree Interest, each node checks its own routing table and selects the neighbour on the shortest path to the sink node as its upstream node on the current job tree. Replying the BuildJobTree Interest starts from mappers to reducers and forwarders, and finally to the sink node. The result is that each node has a record of "JobTreeID – JobNeighbors" locally. The information is used for disseminating job(s) later. The job tree construction completes when the sink node receives all replies from its neighbours. More details of the job tree building steps can be found in MR-Edge [15] paper.

D. Job Execute Phase

The Job Execute Phase starts when the job tree is ready. It contains two steps, the first is node ID allocation that is proposed in this paper to differentiate each data sample. It is the fundamental support of the exactly once data computation feature. The second step is job dissemination and execution, which is the same procedure as in MR-Edge [15]. An improvement is made during the job execution compared with MR-Edge, which saves the intermediate state of job processing on edge devices. The aim of this design is to deal with link failures happening during job execution.

- ID Allocation and Maintenance

The data content identification is challenging. One may argue that each data content can be uniquely identified by using a NDN name as the ID. The problem of directly using NDN names is that it cannot reveal which node(s) has(ve) computed the data sample. Thus, it is hard to check the data computation state after recovering from link failures so that fails to guarantee the exactly once computation on the same data.

For two nodes connected by the same edge on the job tree, we call the one closer to the sink node as the upstream node, the other as the downstream node for clarity in the rest of the paper. When a link failure happens during the data transmission, the downstream node may not be sure if 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 local cached data had been delivered before retransmission 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.

ECE embeds the information of data provider and data computing nodes into the ID of each data content during the job execution as the solution. To identify each data content in the network, this paper firstly assigns a global ID for each node based on the shortest path of the job tree. ID allocation is launched before issuing any job requests. As mapper nodes are the data sources in the proposed design, they label each of their returned data with their node ID plus the job tree ID created by the user/sink node. Data samples from different nodes can only be computed by reducers if they have the same job tree ID to ensure the computation correctness. The ID of a computed data content consists of its reducer's global ID plus the job tree ID. Whenever a link failure happens, the affected node can use the data sample ID(s) to trace back the computation records of its provided data content, such that the node can inquire the computation state of its data content, i.e. whether received and computed correctly.

An AssignID Interest is designed to assign node ID and it is written as (b). 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* is the actual global ID assigned to the corresponding job neighbour and it is construed as below.

/JobNeighbour/JobTreeID/NodeGlobalID (b)

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 unique path between itself to the sink node, a tree-path-based global ID of each node is constructed by accumulating the local IDs on the path from the sink node to itself. 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, which is done by concatenating the local ID of a downstream node at the end of the global ID of the current reducer/forwarder, separated by a hyphen. The reducers and forwarders assign global IDs to their neighbours using the AssignID Interest. The mappers are the leaf nodes of the job tree and consequently they 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 is for a downstream neighbour, in a tuple <downstream job neighbour name, its local ID, its global ID>.

The global ID allocation is undertaken hop by hop starting from the sink node and reaching all the nodes on the job tree. An ACK message is replied from the mappers, in the reversed path of ID allocation, and finally returns to the sink node. To this end, the sink node knows that the ID allocation procedure is complete and it is ready to issue jobs.

Fig.2 presents an example to explain how the ID allocation procedure works. An IoT network topology is shown in Fig.2 (a) with the original connections between the nodes. The numbers inside each circle are used to represent their NDN name respectively. 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 (described in Section C). The NDN name of a node keeps the same no matter which role it acts in ECE protocol.

Assume that node 0 wants to issue a job, it becomes the sink node or user node in the design. It firstly sends the BuildJobTree Interest to the network, resulting in the job tree shown in Fig.2 (b). The solid lines in the figure indicate original network links currently being used on the job tree. The nodes with numbers 8 – 14 labelled with a green colour are the mappers for the current job. Other nodes may act as a reducer or forwarder according to their computing capabilities and the number of downstream neighbours. For instance, node 1 becomes a reducer (in red colour) because it receives data samples from multiple neighbours on the job tree, and it is currently capable of computing these data. Node 6 is a forwarder (in yellow colour) because it connects with 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.

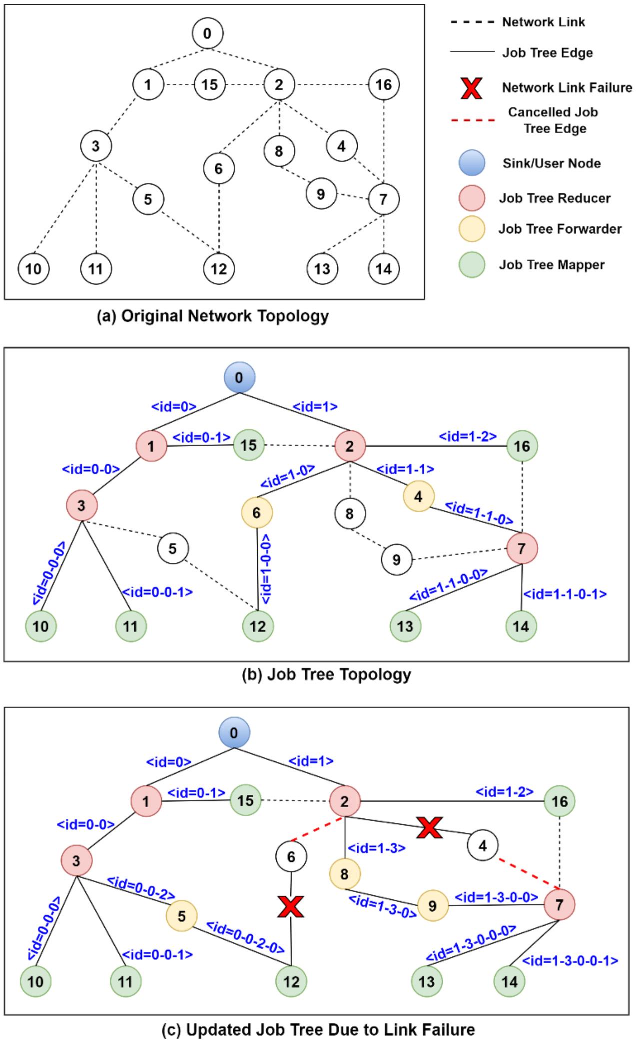


Fig.2. Illustration of ID Allocation of ECE Protocol

When the job tree is ready, node 0 as the sink node assigns the local ID to its job neighbours, i.e., 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 Fig.2 (b) with blue text. Node 1 and node 2 continue the global ID assignment by creating global IDs for their 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, consequently, 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 the mappers receive their global ID. The blue texts in Fig.2 (b) presents 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 explain the details, the path on the established job tree in Fig.2 (b) with the nodes: 10/11 -> 3 -> 1 -> 0 is chosen as an example. Fig.3 (a) shows the respective ID table of the sink node 0 and reducer 1 and 3. The first column of the ID table saves the NDN name of each downstream node, abbreviated 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 direct connected nodes (one is the upstream and the other is the downstream) and is supervised by the upstream node.

The mappers save their global ID and uses the received job ID (sent by the sink node) to label each data they produced, for example, the incremental sequence numbers attached to node 10 and 11 shown in Fig.3 (a). Only data content and its ID are returned during 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.

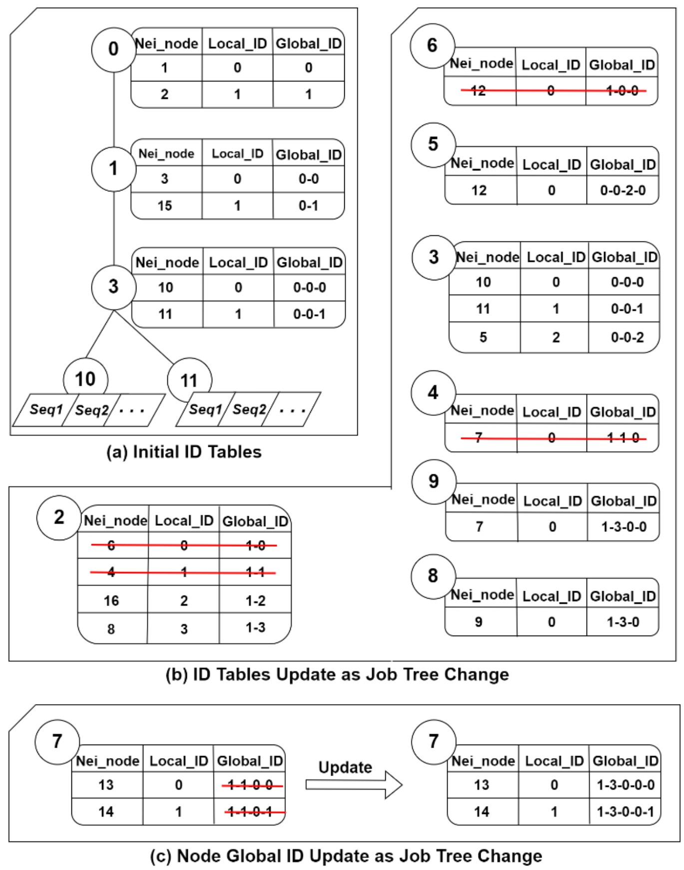


Fig.3. Illustration of ECE Node ID Tables

- Job Dissemination and Execution

When ID allocation is complete, the sink node can send computation tasks by using the ComputingJob Interest which is defined and written as (c).

/JobNei/JobTreeID/JobID/MapFunc/ReduceFunc/ContentFilter (c)

Where, */JobNei* is the name of the neighbour obtained in the Job Tree Build Phase, */JobTreeID* is created by the sink node in the Job Tree Build Phase, which is used to identify the job and to retrieve the corresponding job neighbours in case multiple jobs co-existing in the network, */JobID* is constructed by the sink node for each issued job. The sink node can send multiple jobs on the built job tree. The rest parts of the Interest

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3275179

(*/MapFunc/ReduceFunc/contentFilter)* are defined by each sink node, which describes the functions to process the data and the desired data content.

Every job is sent by the sink node, traverses the intermediate reducers and forwarders and finally reaches the mappers. The procedure of job execution is in the reverse direction of job dissemination. The */ContentFilter* section specifies the data that should be computed by the job. The mappers firstly decompose the ComputingJob Interest to retrieve the user-defined map function. They run the */MapFunc* to process captured data and then return to their selected upstream node. All mapper data is further processed by the reducers at each level of the job tree through the */ReduceFunc*.

For example, a job of counting temperature values in the range of 20-30 Celsius in the Engineer Building can be written as:

/map(x=>(x,1))/reduceByKey((y1, y2)=>(y1+y2))/content(EngineerBuilding/temperatureSensor)

The content filter specifies target data sources for this job, i.e., all temperature sensors in the Engineer Building. Each selected sensor acts as a mapper, which runs the Map function to process its reading and returns the data content in the format of (temperature-reading, 1). The temperature reading of each sensor is treated as the key and the value "1" is the appearance of the temperature reading for this job. Intermediate reducers receive key-value pairs from its job neighbour. They run the Reduce function to add values with the same key.

The sink node gets 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 as that 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.

Two tables are designed to aid ECE nodes to log the data computation state of each job in case link failures happen during executing tasks. The first table is called the Job State (JS) Table that is managed by the sink node. The table is useful to check completed job ID(s) in the Job State Commit Phase to clear corresponding information saved at edge nodes. The sink node creates a record of each issued job request and checks the corresponding received computation results. The job state is saved as a pair of "JobID – State (Completed/Uncompleted)". A completed job means that each edge node on the job tree has finished its processing on the issued job request and final computed result has been correctly delivered to the sink node, which ensures the reliability of the data delivery and computation. More detailed protocol is described in the Job Tree Build Phase. The second table is the Computation Record (CR) Table which saves the job tree ID and the data received from downstream neighbours for the job (abbreviated as dataContent) with its corresponding ID (abbreviated as dataID). Each record in the CR Table is in the form of "JobTreeID – DataID – dataContent". All reducers, forwarders and mappers maintain a CR Table locally. Each of them inserts a record to its CR Table after returning or forwarding the computed/produced data to its upstream node.

E. 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 completed. The Job State Commit Phase is designed to achieve the goal.

The sink node notifies its job neighbours of the specific job ID(s) that have completed in in the Job Execute Phase so that ECE nodes can clear the corresponding saved information. The JobCompleted Interest for this phase is defined as (d) and it can be sent periodically depending on the job requirements, e.g. every 30 seconds or every 10 completed jobs. This paper 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 creates the JobCompleted Interest. Intermediate reducers and forwarders forward this Interest until it reaches mappers.

/JobNeighbour/JobTreeID/CompletedJobID(s) (d)

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. */CompletedJobID(s)* is the successfully computed job ID(s) summarized by the sink node to inform others on the job tree.

As a result, all the ECE nodes achieve the consensus of the completed tasks they have participated and they no longer need to maintain the history records of the completed job(s), e.g., the cached computed data content at reducers and the saved previously captured data samples at mappers. It helps to release resources and space for the edge devices engaged in the data processing. In contrast, the intermediate processing state of tasks should be saved if nodes receive no notifications from the sink node. An ACK procedure is employed to response the JobCompleted Interest, which is initiated by the mappers and traverses in the reverse path of the JobCompleted Interest and finally reaches the sink node as the end of the Job State Commit Phase.

F. Job Tree Rebuild Phase

ECE nodes experiencing link failures can initiate the Job Tree Rebuild Phase to recover. If there is only one neighbour in the original IoT network, i.e. the current upstream node, the node must check the link regularly until it recovers. For instance, as shown in Fig.2 (b), node 13 only has one neighbour (node 7) on the network. Here we focus on the case that the nodes have other paths connecting to the sink node besides the one just failed.

A failed link affects two neighbouring nodes. To help explain the design, the upstream node is defined as the Previous-Upstreamer and the downstream node is defined as Rebuilder. For example, if the link between node 12 and node 6 in Fig.2 (c) is disconnected, node 6 is the Previous-Upstreamer and node 12 is the Rebuilder. The Job Tree Rebuild Phase is always initiated by the Rebuilder. This paper assumes that the link condition is detected by periodically exchanging HELLO messages between the neighbouring nodes, which is a widely used scheme in routing protocols. The following procedure is adopted whenever 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: Rebuilder has other neighbour(s)

A RebuildJobTree Interest is defined as (e) and (f) with a slight difference for this case. Interest (e) is sent by the Rebuilder and Interest (f) is used for the neighbours of the Rebuilder to forward the rebuilding request when needed. The meaning of each part of the Interest is: (1) */NeighbourName* is 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* is to indicate the job tree of interest, and (5) */UpstreamNodeName* is the name of the upstream neighbour of the Rebuilder.

/NeighborName/RebuildTree/RebuilderName/JobTreeID (e)

/NeighborName/RebuildTree/UpstreamNodeName/JobTreeID (f)

If the Rebuilder finds any neighbour(s), it sends a RebuildJobTree Interest (e) to each of its neighbours. A node receives the RebuildJobTree Interest and parses the content. Two scenarios may happen after the node extracts the JobTreeID in the Interest and checks whether it is already on the job tree.

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, also inserts the record to its ID table as introduced in section D. It then 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 received and notifies the other neighbours to withdraw its rebuilding requests.

Scenario-II: the node is not on the job tree with requested JobTreeID. The node re-writes the RebuildJobTree Interest as (f) and forwards it to its neighbours, which repeats the above procedure to process the Interest. If a node has no neighbours available, it directly replies "Rebuild-Rejected". Note that, the mappers are defined as not responsible for disseminating or forwarding jobs to others due to their limited resources and capabilities. Therefore, when a mapper receives a RebuildJobTree Interest, it refuses the request by replying a "Rebuild-Rejected" message even though it is 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.

The Rebuilder can re-enter the Job Execute Phase after receiving its new global ID. Meanwhile, the Rebuilder launches the Job State Sync Phase to make sure neither data losses nor data duplications are caused by the link failure, which is described in next sub-section. If the Rebuilder is connecting downstream nodes on the job tree, it needs to update their global IDs by notifying them with the ChangeID Interest defined as (g). The interest includes three parts: (1) */JobNeighbour* is the name of a neighbour on the job tree, (2) */JobTreeID* is to specify the affected job tree in case multiple job trees coexist, and (3) */ChangeID(NodeGlobalID)* is to inform the downstream neighbours the new ID assigned for the specific job tree.

/JobNeighbor/JobTreeID/changeID(NodeGlobalID) (g)

Case 2: Rebuilder has no other neighbour(s)

If the Rebuilder cannot find any neighbours, it needs to notify its downstream neighbour(s) to search for a new path to reach the sink node. This design aims to reduce the number of nodes affected by link failures as less as possible.

A ChangePath Interest is defined for this case and it is written as (h). 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 (h)

Each downstream neighbour of the Rebuilder becomes a new Rebuilder when it receives 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. When 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 will regularly checks with the Rebuilder to get updates of the failed links (whether it is recovered).

Two examples of link failures are illustrated in Fig.2 (c). The following steps are the rebuilding procedure for the job tree edge between node 4 and node 2 failed.

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. As node 9 is not on the requested job tree, it re-writes the RebuildJobTree Interest and sends to its neighbours. Node 8 takes the same action as node 9 and gets a "Rebuild-ok" message from node 2. Node 9 then 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 Fig.3 (b).

Step-4: Node 7 receives its new global ID and notifies its downstream neighbours on the job tree, i.e., node 13 and node 14, with a corresponding changed global ID by sending the ChangePath Interest. The ID table of node 7 is updated as shown in Fig.3 (c). Meanwhile, 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.

G. 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

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3275179

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 procedure is to synchronize the data computation state starting with the sink node, traversing the reducers or forwarders on the previous path (before link failures), until reaching the Previous-Upsteamer of the Rebuilder. Note that the newly arrived data (after the link failure) from the downstream nodes to the Rebuilder node will be processed as normal, and therefore, this phase can coexist with the Job Execute Phase.

A JobSync Interest is defined for the Job State Sync Phase, as shown in (i). The meaning of each part of the Interest is as follows. */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. */JobSync* is the identifier for the Job State Sync Phase. */RebuilderGlobalID* is the global ID of the Rebuilder. */JobTreeID* is to indicate the specific job tree in case multiple job trees running at the same time. */JobID/DataID* contains the ID(s) of data-samples for specific job to be checked.

/SinkNodeName/DataCheck/RebuilderGlobalID/JobTreeID/JobID/DataID (i)

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: The sink node parses the JobSync Interest to get the */JobID*. It firstly checks whether the task has completed. If a task is marked as completed, it means that all the data content has been correctly computed and received, and consequently the data-samples to be checked is not affected by the Rebuilder's link failure. The sink node can reply a "DataSample-Received" message to the Rebuilder, which indicates that the Job State Sync Phase has finished. If the sink reducer finds that the task state of the *JobID* is uncompleted, it means that the corresponding job execution is still ongoing and the sink node requires more information to answer 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, 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 has been received, the sink node replies a "DataSample-Received" message to the Rebuilder, which indicates that the Job State Sync Phase has finished. Otherwise, the sink node replies "DataSample-Not-Received" and asks the Rebuilder to resend those 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. This requires to decompose the global ID of the Rebuilder to obtain the next hop node to reach the Previous-Upstreamer of the Rebuilder. As described in section D, the global ID of a node consists of its upstream neighbours' global IDs separated by hyphens. The sink node is the starting point of each individual path on the job tree. Therefore, it extracts the first sub-ID (the number before the first hyphen) to find the next destination node to forward the Interest. The sink node compares the sub-ID with all the assigned local IDs in its ID table. The node with a matched local ID is the next hop node (named as NextHop for clarity) to forward the JobSync Interest.

As the downstream nodes require further information to parse the message, the sink node creates a new Interest named ForwardJobSync, as defined in (j). 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 (j)

Step-3: The NextHop node extracts the *RebuilderGlobalID* and *DataID* after having received the ForwardJobSync Interest. It then checks each data sample ID in the *DataID* in its CR Table. For each data sample, if it is received, it means either this node is 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 replies a "Data-received" message for each received data sample to the node (either the sink node or an upstream NextHop) that has sent the ForwardJobSync Interest.

If the *DataID* is not found, the NextHop node searches the *RebuilderGlobalID* in its ID table. If the *RebuilderGlobalID* is found, it means the ForwardJobSync Interest has reached the Previous-Upstreamer of the Rebuilder. The NextHop node replies "DataSample-Not-Received". If the NextHop fails to find the *RebuilderGlobalID* in its ID table, it rewrites 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. Repeating step-3 until a NextHop node finds the *RebuilderGlobalID* matching 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.

Fig.4 presents an example for 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 data computation flow in the

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3275179

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 task 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 revises 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. To this end, 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 forwards the reply from node 6 to node 0.

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

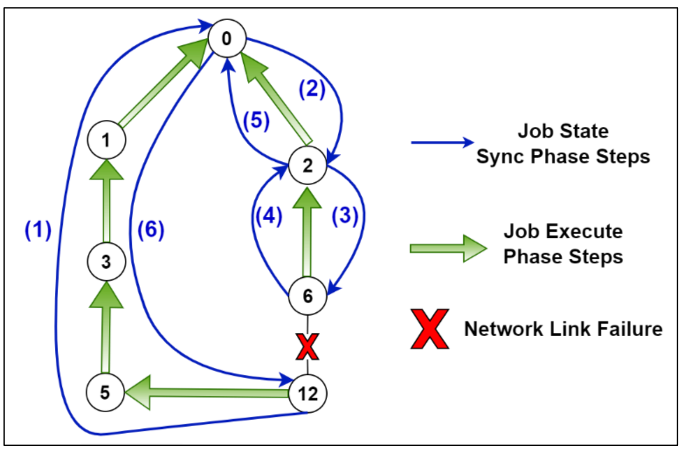


Fig.4. Procedure of ECE Job State Sync Phase

H. Overhead Analysis of ECE

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

- Network Traffic

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 exchanged 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 ECE-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 ECE-RSC phases, including the RebuildJobTree, the ChangePath, the JobCompleted, the JobSync and the ForwardJobSync Interests introduced in previous sub-sections.

In the Job Tree Rebuild Phase, at most three procedures contribute to the overhead traffic. The first is that the Rebuilder searches the alternative path(s) to find an upper stream neighbour which is already on the job tree in order to re-join the job tree. For example, in Fig.2 (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 ECE design. Otherwise, the Rebuilder then 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. More traffic is generated when the Job Tree Rebuild Phase involves node(s) acting as downstream-Rebuilder(s). For example, node 7 as a downstream-Rebuilder communicates with node 4 which is the Rebuilder in Fig.2 (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 (k).

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

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 finished and the rest two procedures can be omitted. Otherwise, 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 answer 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 (l).

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

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 (m).

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

The network traffic overhead of ECE altogether is calculated as O_R+O_S+O_C. Observing equations (k), (l) and (m) 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 current job(s). The deeper and wider the job tree, the bigger the variable N_{total} in equation (m), which increases the overhead traffic. The second factor is the pre-defined frequency for the sink node to send notifications, i.e. t_{commit} in equation (m). For the same job running the same time on the job tree, the smaller the value of t_{commit}, the more rounds of the Job State Commit Phase are invoked. It results in a bigger value of O_C which contributes to the whole overhead of ECE. The last factor is the node that experiences a link failure, i.e. the Rebuilder in ECE. The overhead traffic O_R in equation (k) is tightly 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 (l). The longer the distance, the more messages exchanged to finish the Job State Sync Phase.

- Computation Record Storage

The intermediate state of job execution is saved at each ECE node, i.e. the sink node maintains the JS Table and others have their corresponding CR Table. Let W_i represent the number of records for $node_i$ to insert to its local TS/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 ECE nodes for each clear-record-cycle (W_{ECE}) can be calculated as (n). It is easy to summarize that the overhead of ECE computation record storage is decided by T_{clear}. The smaller the T_{clear} value, the less records maintained by each node. However, it is worth to mention that a smaller T_{clear} results in entering the Job State Commit Phase more frequent, which increases the network traffic overhead. It is up to the sink node or IoT applications to decide the best T_{clear} value.

$$W_{ECE} = \sum_{i=1}^{N_{total}}(W_i * T_{clear}) \quad \text{(n)}$$

IV. Evaluation and Analysis

This section presents tests to verify the feasibility of ECE and evaluate its performance under different link failure scenarios. As ECE 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 ECE-RSC phases), and intermediate state of job processing save at edge nodes.

Due to no existing approaches targeting the same problem as studied in this paper, 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 then splits the data sources into subgroups accordingly. The sink node 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 returns its current state to the sink node (to mimic the central and durable storage for checkpoint snapshots) as the reply for the checkpoint message. The checkpoint is successfully saved if the states of all nodes are normal. Otherwise, the sink node initiates a recovery procedure to fix the failure/error.

Step-3. The sink node randomly picks another device to replace the failed one and migrates the computation tasks on the new-picked node.

Step-4. The sink node asks all nodes on current job graph to rollback to last checkpoint to restart. The system jumps to ***Step-2*** to repeat.

All tests are implemented on ndnSIM [39] which is a simulator specially designed for NDN. The following settings are applied to all tests: the sink/user node sends one task Interest per second. ECE mappers/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 (bandwidth + delay) are set for the simulation: 250 Kbits per second + 10 milliseconds based on the Zigbee protocol between a mapper and a reducer/forwarder of ECE, and between a data source and a processing node of CP-Benchmark. 54 Mbits per second + 1 millisecond using the IEEE 802.11 parameter between reducers and forwarders of ECE, and between processing nodes of CP-Benchmark.

A. Feasibility of ECE

To verify if ECE functions correctly as described in the protocol design section, the network topology shown in Fig.2 (a) is created in ndnSIM. 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.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3275179

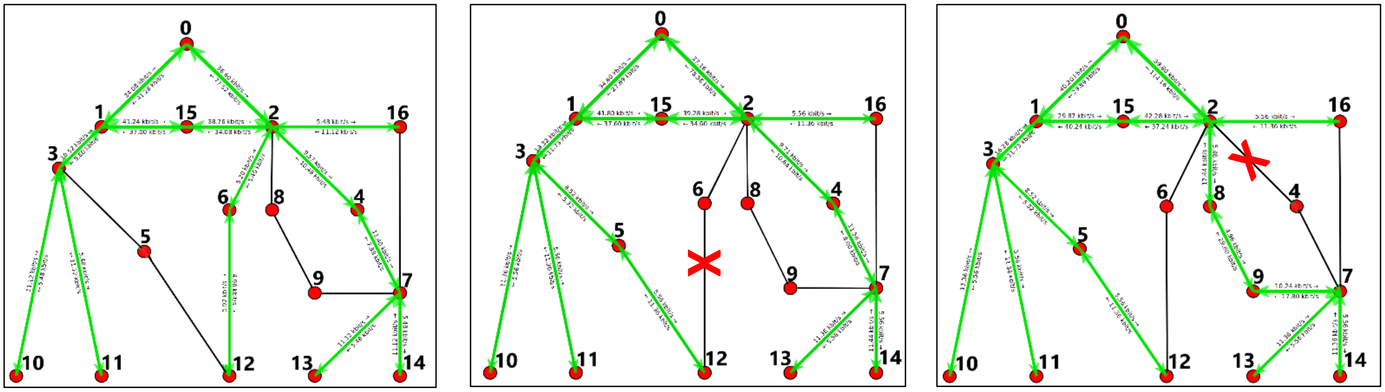

(a) Initial Built Job Tree (b) Job Tree Updated after 1st Link Failure (c) Job Tree Updated after 2nd Link Failure

Fig.5. Job Tree Built and Updated by ECE

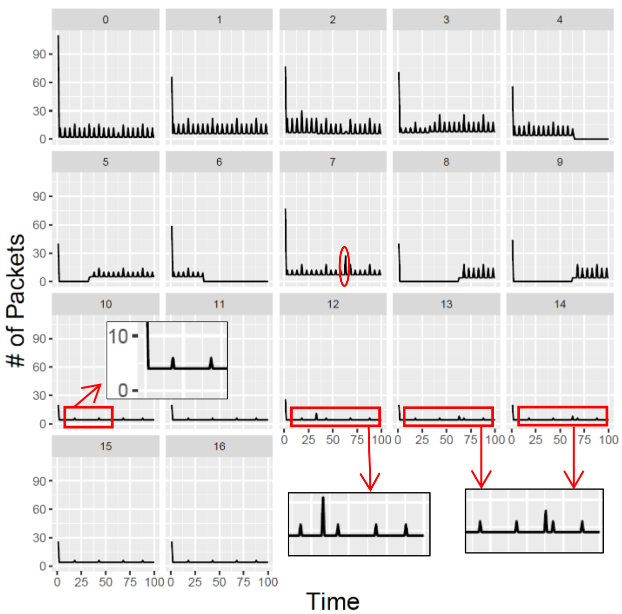


Fig.6. Traffic on ECE Nodes

The cost of all links is set to the same. The job tree is built according to the NDN routing protocol utilizing the shortest path algorithm. Link failures are defined to happen during the job execution at different moments: 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).

Fig.5 shows different job trees 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 shown 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 prove that ECE protocol can deal with link failures without suspending normal job execution procedure. Moreover, the final job result is received correctly neither with data lost nor duplicated processing.

Fig.6 reflects the transmitted traffic at each node during the test. The figures of node 10, 11, 15 and 16 have the same curve pattern, which are stable and repeat regularly. Because 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 represent the periodic JobCompleted Interest sent in the Job State Commit Phase, i.e. 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 is the extra messages of ECE-RSC to handle the first link failure. Secondly, as node 6 only has one job neighbour (node 12) and after the link between them fails, it neither receives nor returns job data. Consequently, its curve stays at 0 after the first link failure. Thirdly, node 5 is 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 adds one more job neighbour (node 5) and therefore it needs to send more ComputingJob 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. Meanwhile, node 4 notifies the link failure situation to its child neighbour node 7 so that node 7 can try to find an alternative route without being affected by the link failure. 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 shows the increased number of messages for the notification of updated global ID.

B. Network Traffic Comparison and Analysis

ECE network traffic overhead is evaluated by comparing with the CP-Benchmark. Two network topologies are created to show the performance. A job in the tests is defined as consecutively executing and completing 100 computational tasks. The sink node sends a JobCompleted Interest every 20 committed tasks in ECE test case. 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 Fig.2 (a)**

The network topology in Fig.2 (a) is created in ndnSIM for tests. Two failures are set during the job execution for ECE and CP-Benchmark respectively. Node 0 is the user node and node 10-16 are data sources. Other 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.

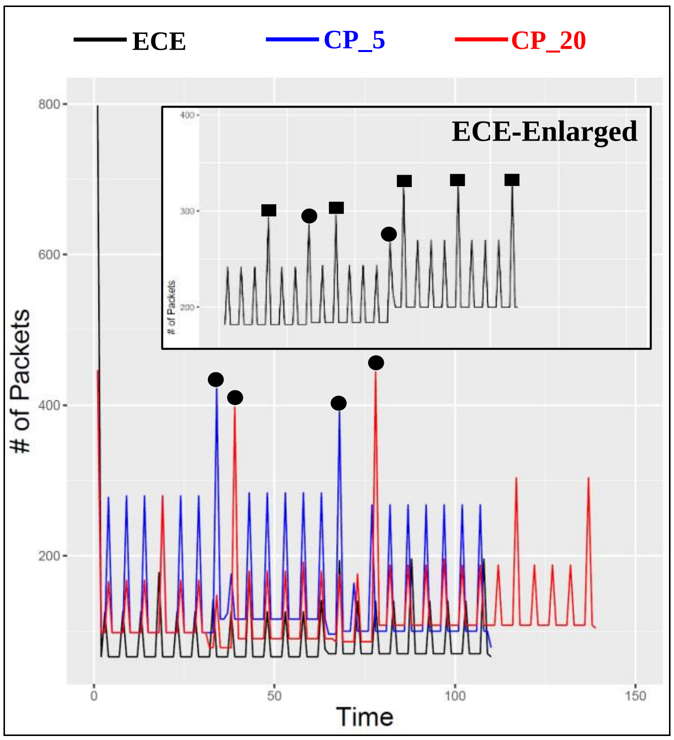


Fig.7. Network Traffic Comparison: ECE Vs. CP-Benchmark

Fig.7 shows the test results, i.e. the black curve represents ECE and CP-Benchmark with checkpoint interval in 5 seconds and 20 seconds is in blue (CP_5) and red (CP_20) respectively. At the beginning of the simulation, the highest peak of ECE is the number of messages exchanged by all nodes in the Job Tree Build Phase. The job tree is built once for every new user node, which brings the most overhead in a round of job execution. As the sink node is assumed to have the information of network resources in advance for the CP-Benchmark solution, the initial cost of generating job graph is lower than that of ECE.

When the job execution starts, it is easy to observe that the network traffic of CP-Benchmark is always above ECE no matter the setting of checkpoint interval. The main reason is CP-Benchmark takes no consideration of the physical topology when generating the logical job plan. In this test, the job graph generated by the CP-Benchmark is selecting node 1 to process data samples from node 10, 13, 14 and 16, node 5 to be responsible for node 11 and 12, and node 7 to manage node 15. The cost of transmitting raw data to edge nodes is larger than the gain of data computation or aggregation. In most cases, the distance between a data source and a processing node is longer than the path of directly sending data samples from the data source to the sink node.

The peaks with a dot on the top of CP-Benchmark curves are the moments to handle link failures. It produces more traffic than the job execution procedure because the sink node needs to pick another edge node to recover and notify all nodes on the job graph to rollback to last checkpoint state. The network traffic of CP-Benchmark with 5-second checkpoint interval (blue curve) is higher than it with 20-second interval (red curve) because checkpoint messages are transmitted more frequent during the job execution. The benefit is that the system can detect and recover from failures more quickly, which reduces job execution latency. The time cost of CP-Benchmark with 20-second checkpoint interval is approximately 30s longer than both its 5-second interval and ECE by observing the x-axis of Fig.7.

An enlarged view of ECE curve is added in Fig.7 to show more details. The peaks with a dot on the top indicate the two link failures. According to the equation (k) and (l) described in previous section, more messages are exchanged to rebuild the job tree, sync job states and retransmit lost data if any. The peaks with a square on the top are the moments of the JobCompleted Interest traversing all nodes on the job tree to clear history job data, as described in equation (m). The job completion time of ECE is the same as CP-Benchmark with 5-second checkpoint interval.

- **BRITE-Topology**

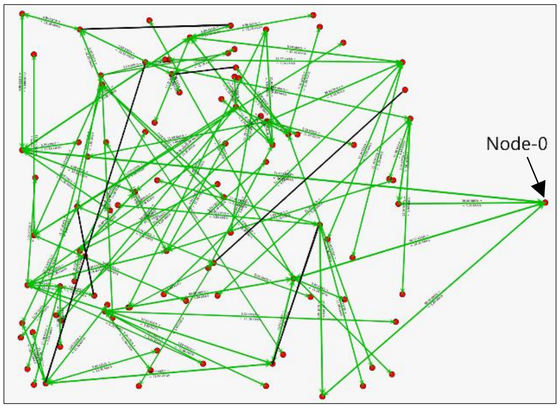
To test the scalability of ECE protocol, a network topology consisting of 100 nodes is generated by using BRITE [40] topology generator with RouterWaxman model. It is called BRITE-Topology for clarity. Node-0 is configured as the sink/user node. For the rest 99 nodes, 69 nodes (node number 31-99) act as mappers/data sources and 30 nodes are edge nodes. Five link failures are set during the simulation for ECE and CP-Benchmark respectively.

Fig.8 (a) and (b) are the corresponding job graph generated by ECE and CP-Benchmark. The red dots represent nodes, green lines with arrows are links used on the job graph and black lines are original network links that are not used by current job. ECE 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 is random in the range from 5 to 15.

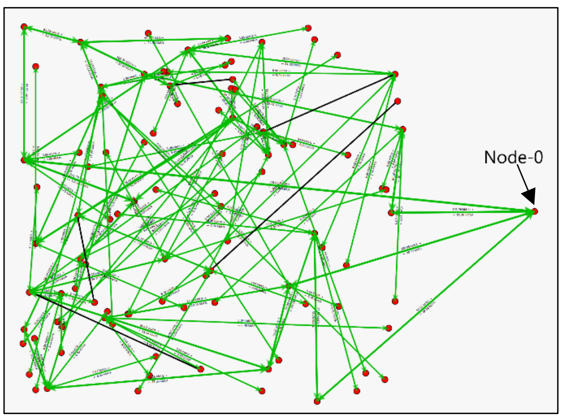
Fig.9 (a) presents the test results of ECE to complete the same job with/without failures. ECE-Exec (red curve) is the test case that no failures happen during the job execution. ECE-RSC (black curve) shows the network traffic varying with ECE to handle five failures during the job execution. Both curves have the highest peak at the initial of the test

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3275179

because of ECE nodes exchanging the routing information to build the job tree.



(a) ECE Job Tree

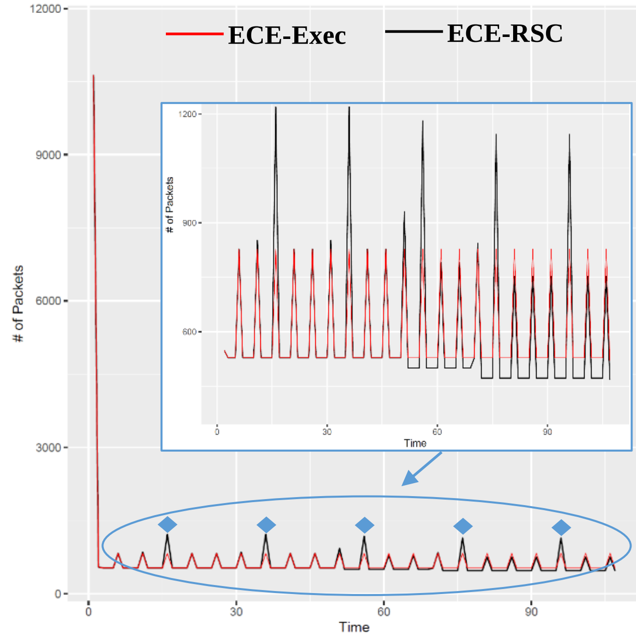


(b) CP-Benchmark Job Graph

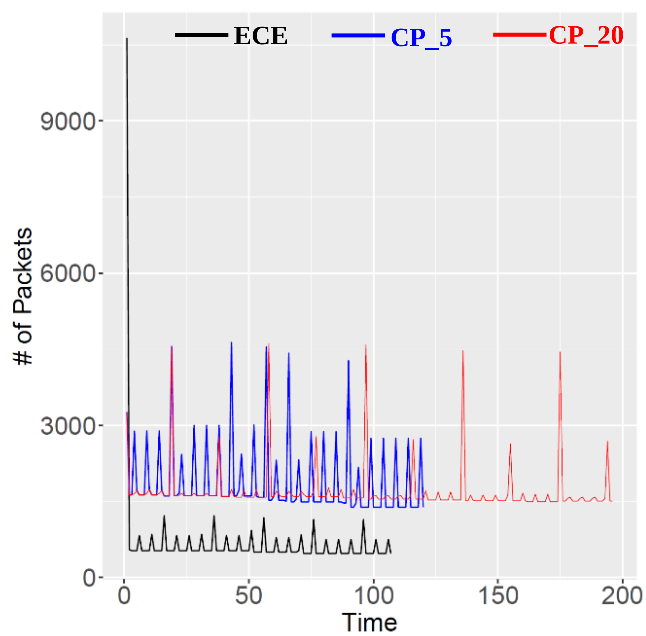
Fig.8. Job Graph on BRITE-Topology

The ECE-Exec curve goes up and down every five seconds during the whole test, which keeps the same as the frequency of reducers to process data every five seconds. The network traffic increases when the reducers return the Data packets after processing. The black curve overlaps with the red curve most time of the simulation, which proves limited extra cost incurred by ECE to achieve exactly once data computation. The peaks with a blue diamond on the top above the black curve represent the sink node sending JobCompleted Interests in the Job State Commit Phase. These peaks also contain the network traffic for ECE handling link failures, which explains the first two peaks are higher than others in the zoomed view of Fig.9 (a). Observing the network traffic, the black curve is lower than the red one from approximately 50^{th} 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 can reduce if a node that was not on the job tree becomes a reducer to aggregate multiple job data content into one Data packet.

The network traffic comparison between ECE and CP-Benchmark is shown in Fig.9 (b). CP-Benchmark with 5-second and 20-second checkpoint interval are respectively presented as the blue (CP_5) and red (CP_20) curve. ECE curve is in black, which is the same as the ECE-RSC shown in Fig.9 (a) if need to see more details. As more nodes are included in the BRITE-Topology, the cost of ECE to build the job tree grows consequently. It also results in the network traffic of CP-Benchmark increasing significantly, which always transmits more packets than ECE to complete the same job.



(a) ECE Overhead Analysis



(b) ECE Vs. CP-Benchmark

Fig.9. Network Traffic Comparison on BRITE-Topology

With the network size increases in IoT, data transmission from data sources to processing nodes contributes a lot to the total network traffic if ignoring their physical topology during job assignment, such as CP-Benchmark randomly grouping data sources with edge nodes. In addition, it causes noticeable delay to finish the same job when using checkpoint based

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3275179

scheme to guarantee exactly once data computation, which could even double the job execution time if observing the red curve in Fig.9 (b).

C. Overhead of ECE Computation Record Storage

For the evaluation purpose, the Clear-Record-Frequency (CRF) is defined as the number of completed tasks to clear all history records once. The job tree built in Fig.5 (a) is applied. A job in this simulation is defined as consecutively executing and completing 200 tasks. The sink node sends a JobCompleted Interest with CRF = 50/20/10 respectively for the same job.

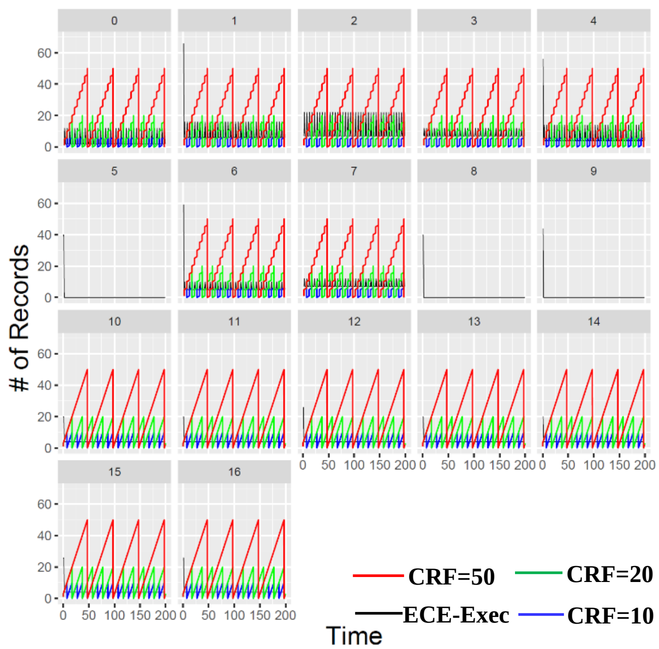


Fig.10. Overhead of Job Computation Records Storage

Fig.10 shows the number of records saved at each ECE 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 for the job tree building and job execution processes, which are the same as the ECE-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 varying with CRFs can be separated into two types. One is the test results of mappers (node 10-16). In the case of CRF = 50, the red curve repeats in a period of increasing from 0 to 50 and dropping to 0. Similarly, the green curve rises from 0 to 20 and downs to 0 with CRF = 20 and the blue curve is in a cycle of 0 to 10 to 0 with CRF = 10. The curves of mappers grow smoothly for all CRF settings because mappers reply each received ComputingJob Interest immediately. A job computation record is added after returning each Data packet. The number of transmitted packets for executing actual jobs stays at 2 no matter the CRF settings, i.e. one Data packet plus one received Interest packet per second in the Job Execute Phase.

The rest of the ECE nodes, i.e. node 0 as the sink node and node 1-4, 6 and 7 as a reducer or a forwarder, present another type of test results. All curves grow every five seconds due to the pre-defined data processing frequency of reducers and forwarders. The number of save job computation records is cleared every 10/20/50 completed jobs with corresponding CRF settings. The curves of job execution packets keep the same, which is not affected by CRF changes. The test results follow the same conclusion of the equation (n) in the previous section that the bigger CRF value the more records maintained by all ECE nodes. It depends on the specific IoT applications to decide the best CRF setting.

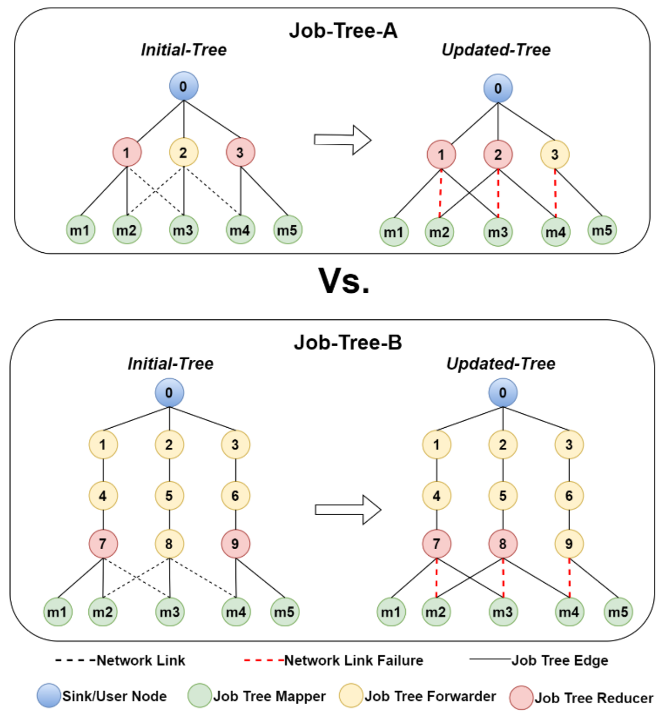


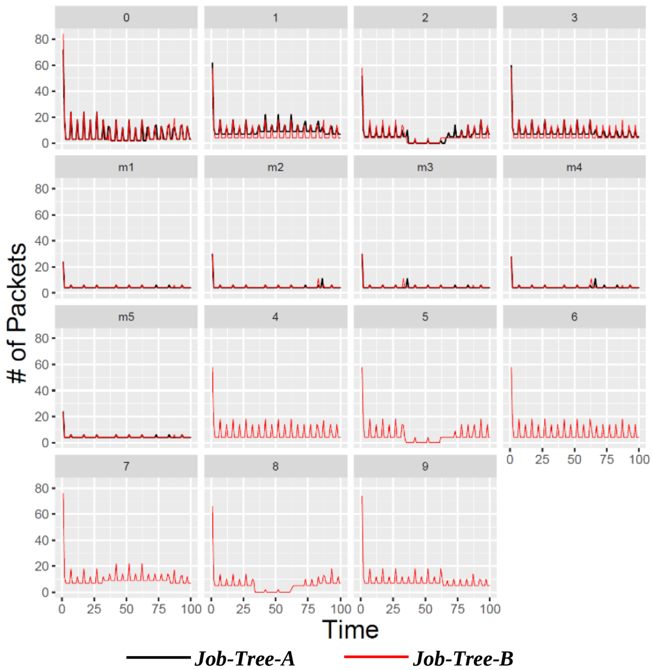
Fig.11. ECE ID affected by Job Tree Depth

D. Overhead of ECE ID Allocation and Update

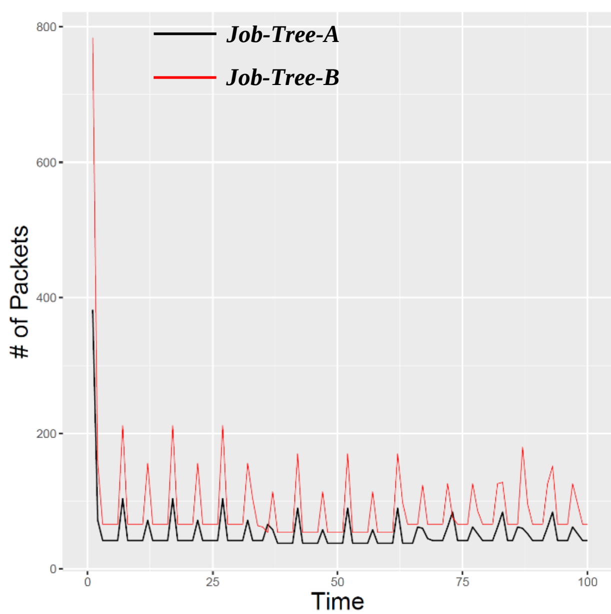
As the ECE node ID is constructed based on the path of the job tree, the depth of a job tree directly affects the cost of the initial ID allocation and as well as the ID update whenever a network failure happens. Two network topologies are created in Fig.11 as a comparative study of the cost of the ECE ID allocation and update affected by the job tree depth. The only difference between the 2 initial job trees, i.e. Job-Tree-A and Job-Tree-B, is the number of intermediate nodes between the sink node and the mappers.

The simulation runs on each job tree for 100 seconds. Three link failures are configured at 32nd, 62nd and 82nd second respectively during the simulation. For Job-Tree-A, the failed links in temporal order are 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 happened in order are the links between node 8 and m3, node 9 and m4 and node 7 and m2. The two updated job trees after the three link failures are also shown in Fig.11 with red dashed lines to indicate the failed links.

The number of transmitted packets by each node varying with the simulation time is presented in Fig.12 (a) and the total network traffic is shown in Fig.12 (b). The black curves represent the test data generated on the Job-Tree-A and the red curves are for Job-Tree-B. For node 4-9, they only have transmitted packets for Job-Tree-B.The curves of mapper m1 and m5 are the same for both tree topologies as the link failures have no effect on their job execution procedure. There is a slight difference in the number of packets in the figures of mapper m2-m4. Because changing from Job-Tree-A to Job-Tree-B only generates more traffic in the Job State Sync Phase with more intermediate nodes involving to forward Interest and Data packets. The transmitted packets by mapper m2-m4 in other ECE phases keep the same.



(a) Cost at each node to update ID



(b) Total Cost of ECE Nodes to update ID

Fig.12. Overhead of ECE ID Update

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

The total cost of the whole job tree is shown in Fig.12 (b). The number of transmitted packets almost increases two times when changing from Job-Tree-A to Job-Tree-B. The formulated equation (k) in the previous section can also apply here. Besides the above reasons, the cost on Job-Tree-B also involves nodes leaving (re-joining) the job tree due to no downstream neighbours (connecting new downstream neighbour(s)), indicated by the variable $D_{down(i)}^{Rebuilder}$ in equation (k). For instance, when the link between node 8 and m3 fails, m3 finds a new path via node 7 on the job tree. When node 8 finds no job neighbours available after losing m3, it leaves the job tree by notifying node 5 the situation. The same actions are taken by both node 5 and 2. When the second link failure between node 9 and m4 happens, m4 sends re-join request to node 8. To this end, node 8, 5 and 2 need to initiate the re-join tree procedure one by one until getting the reply from the sink node, indicated by the variable $D_{Rebuilder}^{Joint-Up}$ in equation (k). Thus, the number of packets transmitted to allocate and update ECE ID is closely related to the tree topology as well as the specific node that experiences the link failure.

V. Conclusion

Collaborative edge computing is a data processing paradigm which employs multiple edge devices cooperating with each other to execute jobs for IoT applications. To achieve exactly once data computation in collaborative edge computing scenarios, one of the challenges to be addressed is the network connections between edge devices may fail during the job execution. This may result in data losses or duplicated data transmission/computations, and consequently violates the exactly once computation guarantee.

This paper proposes the ECE protocol as a solution. It consists of five phases and is built upon the novel ICN architecture. The Job Tree Build Phase is launched before running any jobs and forms a tree based job graph with the sink/user node as the root of the tree. 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 is affected by the failures. Finally, the Job State Commit Phase is designed to notify all the nodes on the job tree on the completed jobs. A set of tests have been performed to show the feasibility and scalability of the ECE protocol and the overhead associated with ID assignment and computation information storage is analyzed.

Future work includes improving ECE with a device capability aware algorithm to build/maintain the job tree for different IoT applications considering the resource constraints, device heterogeneity, energy consumption and mobility of edge devices. As the proposed design is built upon ICN, the naming scheme and/or name resolution may be improved to support more types of IoT jobs, e.g. filtering data sources and/or selecting edge devices.

ACKNOWLEDGMENT

This work has emanated from research conducted with the financial support of the Technological University of the Shannon under the Staff Development Programme, and Science Foundation Ireland (SFI) under Grant Number SFI 16/RC/3918, co-funded by the European Regional Development Fund.

REFERENCES

[1] S. Balaji, K. Nathani, and R. Santhakumar, "IoT Technology, Applications and Challenges: A Contemporary Survey," *Wirel. Pers. Commun.*, vol. 108, no. 1, pp. 363–388, Sep. 2019, doi: 10.1007/s11277-019-06407-w.

[2] 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.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016, doi: 10.1109/JIOT.2016.2579198.

[4] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili, "Collaborative Mobile Edge Computing in 5G Networks: New Paradigms, Scenarios, and Challenges," *IEEE Commun. Mag.*, vol. 55, no. 4, pp. 54–61, Apr. 2017, doi: 10.1109/MCOM.2017.1600863.

[5] Y. Sahni, J. Cao, and L. Yang, "Data-Aware Task Allocation for Achieving Low Latency in Collaborative Edge Computing," *IEEE Internet Things J.*, vol. 6, no. 2, pp. 3512–3524, Apr. 2019, doi: 10.1109/JIOT.2018.2886757.

[6] Q. Wang, Q. Wang, H. Zhu, and X. Wang, "Enabling Collaborative Computing Sustainably Through Computational Latency-Based Pricing," *IEEE Trans. Sustain. Comput.*, vol. 5, no. 4, pp. 541–551, Oct. 2020, doi: 10.1109/TSUSC.2020.2980133.

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

[8] 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.

[9] "Overview," *Docker Documentation*, Apr. 14, 2023. https://docs.docker.com/get-started/ (accessed Apr. 14, 2023).

[10] 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.

[11] 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.

[12] "Structured Streaming Programming Guide - Spark 3.3.0 Documentation." https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html (accessed Jul. 19, 2022).

[13] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng.*, vol. 36, no. 4, pp. 28–38, 2015.

[14] "HDFS Architecture Guide." https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (accessed Jan. 26, 2023).

[15] 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.

[16] 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.

[17] "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

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

[19] B. Wang, M. Li, X. Jin, and C. Guo, "A Reliable IoT Edge Computing Trust Management Mechanism for Smart Cities," *IEEE Access*, vol. 8, pp. 46373–46399, 2020, doi: 10.1109/ACCESS.2020.2979022.

[20] Z. Cai and T. Shi, "Distributed Query Processing in the Edge-Assisted IoT Data Monitoring System," *IEEE Internet Things J.*, vol. 8, no. 16, pp. 12679–12693, Aug. 2021, doi: 10.1109/JIOT.2020.3026988.

[21] 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.

[22] "Apache Kafka," *Apache Kafka*. https://kafka.apache.org/0102/documentation/streams/architecture (accessed Feb. 01, 2023).

[23] "Messaging that just works — RabbitMQ." https://www.rabbitmq.com/#features (accessed Apr. 17, 2023).

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

[25] "Two-phase-commit-concepts," Oct. 21, 2022. https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/informix-servers/14.10?topic=protocol-precommit-phase (accessed Jan. 26, 2023).

[26] L. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73.

[27] "Named Data Networking: Motivation & Details," *Named Data Networking (NDN)*. https://named-data.net/project/archoverview/ (accessed Jan. 26, 2023).

[28] N. Fotiou, V. A. Siris, G. Xylomenos, G. C. Polyzos, K. V. Katsaros, and G. Petropoulos, "Edge-ICN and its application to the Internet of Things," in *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, Jun. 2017, pp. 1–6. doi: 10.23919/IFIPNetworking.2017.8264880.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] L. Davoli, L. Belli, A. Cilfone, and G. Ferrari, "From Micro to Macro IoT: Challenges and Solutions in the Integration of IEEE 802.15.4/802.11 and Sub-GHz Technologies," *IEEE Internet Things J.*, vol. 5, no. 2, pp. 784–793, Apr. 2018, doi: 10.1109/JIOT.2017.2747900.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3275179

[34] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, “A Survey on Mobile Edge Computing: The Communication Perspective,” *IEEE Commun. Surv. Tutor.*, vol. 19, no. 4, pp. 2322–2358, 2017, doi: 10.1109/COMST.2017.2745201.

[35] Y.-K. Huang, A.-C. Pang, P.-C. Hsiu, W. Zhuang, and P. Liu, “Distributed Throughput Optimization for ZigBee Cluster-Tree Networks,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 3, pp. 513–520, Mar. 2012, doi: 10.1109/TPDS.2011.192.

[36] 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 Trans. Green Commun. Netw.*, vol. 6, no. 2, pp. 685–694, Jun. 2022, doi: 10.1109/TGCN.2022.3143991.

[37] M. Li, F. R. Yu, P. Si, W. Wu, and Y. Zhang, “Resource Optimization for Delay-Tolerant Data in Blockchain-Enabled IoT With Edge Computing: A Deep Reinforcement Learning Approach,” *IEEE Internet Things J.*, vol. 7, no. 10, pp. 9399–9412, Oct. 2020, doi: 10.1109/JIOT.2020.3007869.

[38] C. Cicconetti, M. Conti, and A. Passarella, “A Decentralized Framework for Serverless Edge Computing in the Internet of Things,” *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 2, pp. 2166–2180, Jun. 2021, doi: 10.1109/TNSM.2020.3023305.

[39] “ndnSIM Documentation — ndnSIM documentation.” https://ndnsim.net/current/ (accessed Sep. 27, 2022).

[40] “BRITE: Boston university Representative Internet Topology gEnerator.” https://www.cs.bu.edu/brite/ (accessed Jan. 12, 2023).

Qian Wang is a Ph.D. candidate with the Software Research Institute (SRI) at Technological University of the Shannon: Midlands Midwest, Ireland. She received her B.Sc. in Electronic Information Science and Technology from Shaanxi University of Science, China and Technology in 2012 and M.Sc. in Ubiquitous Computing from Trinity College Dublin, Ireland in 2013. Her research interests include Internet of Things, Edge Computing and Information Centric Networking.

Brian Lee is the director of the Software Research Institute (SRI) at Technological University of the Shannon: Midlands Midwest, Ireland. He is a Science Foundation Ireland (SFI) Funded Investigator in the SFI CONFIRM Smart Manufacturing Centre. He received his Ph.D. in Computer Science from Trinity College Dublin in 2004. He is a member of IEEE (Communications, Computer and Robotics and Automation societies) and ACM). His research interests include Computer Security (Access Control, Network Security, Security Analytics) and Programmable Networking and Edge Computing.

Niall Murray (Member, IEEE) is currently a Senior Lecturer with the Faculty of Engineering and Informatics, Technological University of the Shannon: Midlands Midwest, Ireland. He is also the Founder, in 2014, and the Principal Investigator (PI) in the truly Immersive and Interactive Multimedia Experiences (tIIMEx) Research Group in AIT. He is a Science Foundation Ireland (SFI) Funded Investigator (FI) with the Confirm Centre for Smart manufacturing and an FI with the SFI Adapt Centre for AI enabled Digital Content. He is an Associate PI with the Enterprise Ireland Funded Technology Gateway COMAND. His current research interests include immersive and multisensory multimedia communication and applications, multimedia signal processing, quality of experience, and wearable sensor systems (further information available at: www.niallmurray.info).

Yuansong Qiao is a Senior Research Fellow in the Software Research Institute (SRI) at Technological University of the Shannon: Midlands Midwest, Ireland. He is a Science Foundation Ireland (SFI) Funded Investigator in the SFI CONFIRM Smart Manufacturing Centre. He received his Ph.D. in Computer Applied Technology from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2008. He is a member of IEEE (Robotics and Automation Society and Blockchain Community) and ACM (SIGCOMM and SIGAI). His research interests include Future Internet Architecture, Blockchain Systems, Robotic Control and Coordination, and Edge Computing Intelligence.