1-15-2020

ICedge: When Edge Computing Meets Information-Centric Networking

Spyridon Mastorakis
University of Nebraska at Omaha, smastorakis@unomaha.edu

Abderrahmen Mtibaa
University of Missouri-St. Louis

Jonathan Lee
Duke University

Satyajayant Misra
New Mexico State University

Recommended Citation

S. Mastorakis, A. Mtibaa, J. Lee and S. Misra, "ICedge: When Edge Computing Meets Information-Centric Networking," in IEEE Internet of Things Journal.

ICedge: When Edge Computing Meets Information-Centric Networking

Spyridon Mastorakis, Abderrahmen Mtibaa, Jonathan Lee, and Satyajayant Misra

***Abstract*—In today's era of explosion of Internet of Things (IoT) and end-user devices and their data volume, emanating at the network's edge, the network should be more in-tune with meeting the needs of these demanding edge computing applications. To this end, we design and prototype Information-Centric edge (*ICedge*), a general-purpose networking framework that streamlines service invocation and improves reuse of redundant computation at the edge. *ICedge* runs on top of Named-Data Networking, a realization of the Information-Centric Networking vision, and handles the "low-level" network communication on behalf of applications. *ICedge* features a fully distributed design that: (i) enables users to get seamlessly on-boarded onto an edge network, (ii) delivers application invoked tasks to edge nodes for execution in a timely manner, and (iii) offers naming abstractions and network-based mechanisms to enable (partial or full) reuse of the results of already executed tasks among users, which we call "compute reuse", resulting in lower task completion times and efficient use of edge computing resources. Our simulation and testbed deployment results demonstrate that *ICedge* can achieve up to $50\times$ lower task completion times leveraging its network-based compute reuse mechanism compared to cases, where reuse is not available.**

I. Introduction

The currently underway Internet of Things (IoT) revolution and the significant growth of mobile end-user devices is resulting in a significant increase in the number of devices, and correspondingly significant growth in data volumes and computation needs at the network edge. Most of these edge devices will require high-bandwidth and low-latency remote processing of the data they generate in order to take real-time actions. To this end, the existing cloud computing model has been proven to be inadequate [33]. The future calls for networks with pervasive IoT device deployments (*e.g.,* smart cities, autonomous vehicles, smart homes, smart grid, augmented/virtual reality) powered by edge computing services that bring computation from the cloud closer to users [31]. The market share for edge computing has shown constant growth, with a predicted annual growth rate of 27% until 2023 to more than $9 Billion [20]. New applications, such as augmented/virtual reality and interactive games, with increasingly larger amounts of generated data and extremely low-latency communication requirements, pose new challenges for the design and instrumentation of the network infrastructure, even with edge computing [33].

In edge computing research, advancements thus far have focused on application-related problems, such as task scheduling [31], segmentation [5], [6], and energy consumption [28], [24]. These advancements are the driving force behind edge computing. In this paper, we look at another facet of the problem. We "shed light" on the networking aspects of edge computing, investigating whether the underlying network and the functions it offers can enhance the performance of edge computing applications. We argue that the network architecture itself, as well as networking frameworks, are vital enablers of edge computing and their design ought to be revisited in the context of the varied needs of edge applications.

In line with this assertion, and to enable applications to take advantage of edge computing resources, for the invocation and execution of computational services[1] (*e.g.,* image or video annotation, map navigation) offered by Compute Nodes (CNs) at the edge (*i.e.,* edge servers), we present *Information-Centric edge* (*ICedge*). *ICedge* is a network-based edge computing framework designed to handle the "low-level" communication details on behalf of applications. *ICedge* interacts with the underlying network to ensure that: (i) users can seamlessly connect to an edge network through *any* operational CN and discover available compute resources/services, (ii) user-invoked tasks are distributed to CNs that have adequate resources for their execution, and (iii) results of previous computations are reused among multiple users to minimize execution/completion time. At the same time, *ICedge* features a fully distributed and general-purpose design, which can be used by any edge computing application.

ICedge runs on top of the Named-Data Networking (NDN) architecture [37], a popular realization of the Information-Centric Networking (ICN) paradigm [36]. We argue that edge computing is inherently service-centric; a user, seeking some service from the edge network, can obtain it from any of the many CNs offering the service [25]. To this end, the NDN architecture (and the ICN paradigm in general) inherently matches the objective of edge applications with the use of application-defined naming at the network layer.

To deliver its functionality and serve edge computing application needs as a generic network-based edge framework, *ICedge* needs to overcome the following challenges:

- How to make an edge network, privy to minimal network configuration information, aware of necessary information

Spyridon Mastorakis is with the University of Nebraska, Omaha (email: smastorakis@unomaha.edu).
Abderrahmen Mtibaa is with the University of Missouri-St. Louis (email: amtibaa@umsl.edu).
Jonathan Lee is with Duke University (email: jonathan.h.lee@duke.edu).
Satyajayant Misra is with the New Mexico State University (email: misra@cs.nmsu.edu).

[1]In the rest of this paper, we use the terms "services" and "tasks" interchangeably to refer to the users' requests for computation in an edge network.

(*e.g.,* edge service utilization) so that application requests can be delivered to the appropriate CN(s) (among all the available CNs) offering a service, in a truly distributed manner?
- How to help the network adapt to the highly dynamic conditions of an edge ecosystem (*e.g.,* highly variable rates of user requests, CN load, and/or CN failures)?
- How to offer distributed network-based mechanisms to reuse the results of previously executed computational tasks to satisfy new, reusable user requests?

Contributions: *ICedge* is a fully distributed framework that provides mechanisms for: (i) seamless user on-boarding onto an edge network, allowing them to discover the offered services (Section V), (ii) dynamic "learning" and building of network paths to the CNs that offer each service (Section VI-C), and (iii) adaptive forwarding of service requests to CNs based on the current network conditions (*e.g.,* CN load, latency, energy consumption) (Section VI-D). To the best of our knowledge, *ICedge* is the *first network-based framework* that offers named-based compute reuse abstractions to applications in a fully distributed manner (Section VII). Finally, we implement an *ICedge* prototype, which we evaluate through simulations (Section VIII) and testbed deployment of a real-time face detection application (Section IX). Our evaluation results demonstrate that *ICedge* achieves up to $50\times$ lower task completion times through its network-based compute reuse mechanism compared to cases of no reuse. Compared to an IP-based solution consisting of one or more task dispatchers, *ICedge* achieves $1.27 - 2.32\times$ lower overheads with $1.06 - 1.33\times$ lower task completion times.

Section II presents the motivational examples for our work. Section III presents the background and related work in the area. Section IV presents the system model and assumptions. Section V presents the distributed service discovery, while Section VI details the network-aware service invocation mechanism. Section VII details the mechanisms for compute reuse. Section VIII presents our simulation based evaluation of *ICedge* and Section IX presents results from an experimental proof-of-concept deployment. Section X concludes our paper.

II. Motivation

We motivate our work by discussing how a network-based edge framework, such as *ICedge*, could serve the needs of edge computing applications and enhance user quality of experience (QoE).

Let us consider a scenario, where visitors of the Louvre museum would like to see as many exhibits as possible. For a given painting, say the Mona Lisa by Leonardo da Vinci, visitors take pictures of the painting using a smartphone (edge) application in order to learn more about it. In such a scenario, several visitors will be requesting *similar* computational tasks, in the sense that they may take pictures of the Mona Lisa from different angles or with different shades and ask for similar information, such as the history of the painting.

The Louvre counts over 10 M visitors annually [1], which roughly translates to 28 K visitors daily. Even if only half of the visitors in a day request further information via a picture of the of Mona Lisa, *14 K requests for similar information will be re-executed daily if there is no mechanism to exploit reuse of previously executed tasks*. This number balloons if we consider that the Louvre contains more than 410 K exhibits. Assuming that visitors visit on average 20% of the exhibits in a given day, the number of tasks (*i.e.,* requests for exhibit information) may exceed one billion every day, which represents more than 30 K requests for similar information per second during the Louvre's admission hours.

We argue that *ICedge* can help applications utilize the available edge resources to: (i) achieve low end-to-end latency and high availability, thus being able to serve users under highly dynamic conditions (*e.g.,* visitors' demand that may rapidly and unpredictably change for one or more exhibits during a day), as required by edge computing applications [30], and (ii) enable reuse of previous computation at the edge. We emphasize that *ICedge* is orthogonal to existing edge computing research advancements (*e.g.,* scheduling, resource management, compute optimization). *ICedge* complements and benefits such advancements by handling the low-level communication complexity with the underlying edge network. *ICedge* can also facilitate the design and implementation of new edge computing applications via its rich networking semantics with minimal development effort.

Why is *ICedge* built on top of an ICN substrate? ICN provides a context-aware network substrate that uses application-defined naming as the identifier for communication purposes. This shared identifier between applications and the network enables edge applications to semantically represent computational tasks, making them directly visible to the underlying network. Consequently, the ICN network forwards tasks from users to CNs based on their names, allowing for an adaptive forwarding behavior, where tasks/requests for edge services are dispatched to the CNs that can execute them. Once users receive the computation results, these results are natively cached in the network for low latency access by users requesting the same computation in the future.

Building on top and extending the ICN abstractions, *ICedge* achieves: (i) adaptation to the highly dynamic edge conditions by assessing to which CN to forward tasks for execution based on a variety of objectives (*e.g.,* CN load, energy consumption, CPU and/or memory capability), (ii) seamless reuse of data and computation within the edge network and at the CNs, and (iii) a fully distributed operation paradigm, allowing the edge network to be highly available for edge applications, without relying on centralized coordination.

III. Background and Related Work

A. Edge Computing Research

Different from cloud computing, various solutions for computation offloading to less powerful surrogate machines [8], known as cyber-foraging [4], have been proposed. Solutions like Clone Cloud [5] and MAUI [6] alleviate the load on the distant Cloud through an edge computing paradigm. Edge computing research has expanded rapidly to multiple other areas, including scheduling algorithms [13], task abstraction mechanisms [5], as well as resource management [19] and energy consumption [26] designs. Moreover, fully reusing the results of another computational task (*e.g.,* annotating the same image) is rather rare in the edge computing context [11]. The partial reuse of results (stemming from similar computational tasks, *e.g.,* annotating similar images) among users or applications has been purely considered from an application point of view by mostly applying machine learning and optimization techniques [12], [11]. The direction of leveraging partial reuse of results among users over a network with multiple CNs remain largely unexplored.

While most of this research considers application-specific scenarios, we investigate a direction orthogonal to these areas: a network-based framework that can be used by applications for the invocation of edge services. Such a framework complements the research outcomes mentioned above.

B. Named-Data Networking

The Named-Data Networking (NDN) architecture [37] offers a receiver-driven communication model, where consumer applications send requests for named data, called *Interest packets*. An Interest consists of the name of the requested data and other optional elements defined by consumers. Producer applications receive Interests and send the data back to the consumer(s) that requested it. A data packet consists of the data name, the content, and carries the producer's signature which binds the data name to the actual content.

NDN is based on the following three fundamental ideas: **(i) identifying packets at the network layer through semantically meaningful application-defined names**–NDN itself does not name the data, but rather carries packets that contain application-defined names (*e.g.,* existing network applications use "names" in the format of urls); **(ii) securing data directly**–each network-layer data packet carries the signature of the producer, which secures the data at rest and in transit over the network; and **(iii) a stateful name-based forwarding plane**–each Interest is forwarded based on its name by NDN routers, leaving state at each router. The corresponding data packet uses this state to follow the same path as the corresponding Interest back to the consumer(s), satisfying/consuming the Interest state at each router.

To realize such a name-based stateful forwarding plane, NDN routers are equipped with three data structures. The first one is a Forwarding Information Base (FIB), which contains name prefixes along with one or more outgoing interfaces, and is used for Interest forwarding. The second is a Pending Interest Table (PIT), which stores recently forwarded Interests that have not brought back a data packet yet. The third one is a Content Store (CS), where routers store recently retrieved data packets to satisfy future Interests for the same data.

Note that NDN applications can define their own semantically meaningful naming schemes, called "naming conventions"; a set of principles/rules on how producers should name the data they produce and how consumers should request it. For example, applications that produce video files can define that the first component of the name of a produced video will be its title and the second component will be the index of a chunk in the video file (*e.g.,* "`/video/0`" will be the first data chunk in a file with title "video"). Following the defined conventions, consumer applications learn how to form names in order to request the data they need. For example, an application requesting "video" will send an Interest with a name "`/video/0`" to fetch the first video chunk, an Interest with a name "`/video/1`" to fetch the second video chunk, etc.

C. Edge Computing Research over NDN/ICN

The first attempt to explore an ICN-based edge computing system was Named-Function Networking (NFN) [34]. NFN uses function names to locate remote compute resources and perform in-network computation over NDN. NFaaS [18], building on top of NFN, focuses on placing functions in the network and executing them through virtual machines. Functions can be downloaded by any node in the network through uni-kernels.

NFN and NFaaS are preliminary designs for in-network function execution, thus they are inefficient for compute-intensive applications, since they require the network to keep long-lasting state during the execution of a function. To overcome this limitation, RICE [16] decouples the invocation of a function/service from the retrieval of the results. Mtibaa *et. al.* [25], Grewe *et. al.* [10], and Amadeo *et. al.* [3] summarize different challenges for the design and implementation of edge computing systems over NDN and perform some initial exploration of the design space. In our previous work, we performed a preliminary investigation of the potential benefits of NDN for the discovery of edge services and functions at the edge [23]

ICedge differs from prior work, since it enables the network to "learn" paths to all the CNs offering a service and to dynamically prioritize different paths based on the conditions of the edge ecosystem (Section VI). *ICedge* allows for seamless on-boarding of users onto the edge network (Section V) and offers network-based compute reuse abstractions to applications (Section VII).

IV. System Model & Assumptions

We define an edge network as an autonomous network consisting of end users, a set of forwarding nodes (*i.e.,* routers) and a set of CNs offering a set of pre-installed services to users without requiring any *a priori* configuration or registration. The CNs are small server-class nodes with computation and storage capabilities. We assume that CNs belonging to an edge network can be accessed though direct links, *e.g.,* LTE, 5G, or multihop links (2-4 hops).We assume that the CNs in a specific edge environment are administered by a single entity (*e.g.,* stakeholder, ISP). Figure 1 illustrates an edge network consisting of four routers, Router A through D, and three CNs, CN_1, CN_2, and CN_3.

Users can join an edge network (*e.g.,* while visiting a city or a museum), discover the offered edge services, and invoke pre-determined services offered by CNs. We assume that the edge network can scale up to several hundred CNs; we deem larger numbers unnecessary, as they are subject to diminishing returns. Further, if a service cannot be executed by the local edge network (*e.g.,* due to the CNs being fully

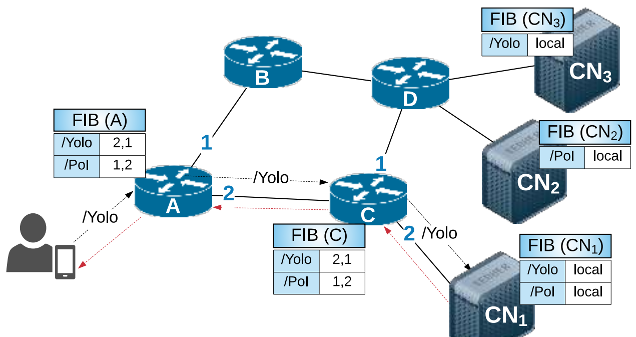


Fig. 1: *ICedge* in action: A user requests a service, such as /Yolo for image annotation or /PoI to find Points of Interests (PoI) on a map. Routers forward the request to the most suitable CN based on information in their FIBs. A FIB entry consists of a service name prefix and a list of outgoing interfaces towards a service.

utilized), the service request will be forwarded to a distant Cloud for execution. This assumption is motivated by the fact that forwarding tasks to the Cloud is generally faster than accessing a distant edge, which may have longer round-trip delays and slower processing capabilities than the Cloud [31].

System Model: We consider an edge network E consisting of n CNs $(CN_1, \cdots, CN_n)$, m users $(u_1, \cdots, u_m)$, and q routers. E offers a set of k services, $\mathcal{S} = \{s_1, \cdots, s_k\}$, where each CN offers a subset of services $\mathcal{S}_i$, such that $\mathcal{S} = \bigcup_{i=1}^{n} \mathcal{S}_i$.

Security Assumptions: Each entity (users and CNs) has a pair of public/private keys and an identity: a name bound to the entity's public key through a certificate [40]. We assume that the users and the CNs have common trust anchors established, so that they are able to verify the certificate of each other on a certificate chain with the common anchors as the root. In this way, *ICedge* takes advantage of NDN abstractions to provide mutual authentication between users and CNs and provenance of data at different stages of processing.

System Components: *ICedge* delivers its functionality through three building blocks, namely: (i) a service discovery component that allows users to discover the services offered by the edge network and how to invoke them, (ii) a service invocation framework that facilitates invoking a service, forwarding a service request to the most suitable CN, and retrieving the execution results, and (iii) a component that offers network-assisted reuse of previously executed tasks to maximize the efficiency of the utilization of edge computing resources. We discuss these blocks in detail in the following sections.

V. Distributed Service Discovery

In this section, we first discuss the need for a service discovery mechanism that helps users identify available services in the network. Then, we propose a scheme that enables seamless service and resource discovery to bootstrap the remote invocation and execution of services.

A. Problem Statement

Why Do We Need Discovery? Most existing frameworks assume that users are *a priori* aware of the service name they would like to invoke [16], [34], [17]. In reality, users connecting to an edge network may be unaware of several factors. For instance, they may not know the name of the service of interest. If users are aware of the service name, they may not know if the desired service is offered by the network they are connected to. Multiple versions of the same service or multiple services may exist, which one should be selected? Applications may also have certain naming conventions for service invocation (we discuss examples in Section VII).

What Exactly Needs to be Discovered? Applications need to discover the following information: (*i*) the availability of services, and (*ii*) the naming conventions to be used in order to invoke edge services. The first piece of information is needed so that users can discover the service(s) providing the functionality they need in the most faithful way along with service-related metadata (*e.g.,* service description, version, complexity). The second one defines how the application can invoke a service (*e.g.,* what are the service input parameters, what are the components needed to form a name for a service).

B. Service Discovery Design

We propose a distributed discovery mechanism that runs at each CN without relying on auxiliary entities deployed in the network (*e.g.,* network controllers). Users send a *discovery Interest* under the "`/discovery`" namespace, which carries a description of the service functionality they are looking for (Interest *I1* in Figure 2). This Interest reaches the closest operational CN, which sends back a response. The response contains the service (along with related metadata) that matches the user description and a naming convention, so that the user learns how to invoke the service (data packet *D1* in Figure 2).

Distributed Service Synchronization: Given that *any* operational CN in the network needs to be able to respond to the discovery Interests, every CN needs to be aware of the services offered by the edge. This can be achieved through a distributed synchronization protocol (sync for short), such as RoundSync [7] or PSync [38]. Sync operates in a peer-to-peer fashion, where CNs subscribe to a common namespace (*e.g.,* "`/CN/sync`") and exchange information about what services they offer (along with metadata and naming conventions for each service). When a change happens to the services offered by a CN (a CN adds, removes, or updates a service), the CN multicasts a signed sync Interest (publishes a message) under the common namespace. For example, in Figure 2, CN_2 sends *I2* to notify CN_1 and CN_3 that it has added a service "`/PoI`", so that users can find Points of Interest (PoI) on a map. A sync Interest reaches all the CNs in the edge network and triggers them to request the latest service change from the CN that sent the sync Interest.

CNs periodically send sync Interests to other CNs in the edge network to notify them about their latest offered services. This period is called *sync Interest interval*. A given service s_i will be removed if all the offering CNs explicitly publish a removal of s_i or all the offering CNs fail. A CN is considered to have failed, when other CNs do not hear back from it within a certain timeout interval.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2020.2966924, IEEE Internet of Things Journal

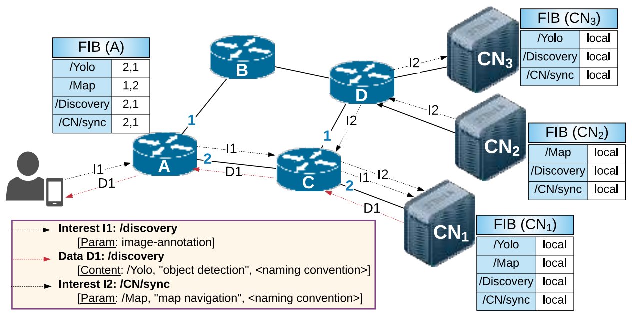


Fig. 2: Service discovery example. A user sends a discovery Interest that typically reaches the closest operational CN, which will help the user connect to the edge network.

VI. Network-Aware Service Invocation

A. Problem Statement

After a user discovers the name of the service to invoke and the appropriate naming conventions, we need a set of protocol abstractions that enable users to request the execution of the service by a CN. There are certain challenges that need to be addressed here. The network may not be aware in advance of which CNs offer each service and through which paths to reach each of the CNs for a given service. If the network is unaware, it has to learn how (*i.e.,* through which paths) to forward user requests to the CNs that offer the service. Given the highly dynamic conditions of an edge ecosystem (*e.g.,* in terms of user service invocation rates, CN load), even if the network knows the pertinent CNs, it has to adaptively forward requests to the most suitable CN based on the current edge conditions.

Overall, the *ICedge* service invocation building block consists of three sub-components: (i) *a service invocation protocol*, (ii) *a network self-learning mechanism to establish service reachability*, and (iii) *a service request forwarding component that adapts to the dynamic edge conditions*.

B. Service Invocation Protocol

In Figure 1, we illustrate a high level example of *ICedge* in action. A user sends an Interest (request) for the execution of an edge service named "`/Yolo`". The network paths (routes) to CNs are determined based on information in routers' FIBs (we explain how routers establish this information in Section VI-C). Each router ranks its next-hops for a service based on various metrics, such as network latency, CN load, and/or compute capabilities (we present details in Section VI-D). As two out of three CNs in Figure 1 (CN_1 and CN_3) offer "`/Yolo`" (marked with a FIB entry with a "local" next-hop), intermediate routers perform a FIB lookup to forward the user request to the most suitable CN (CN_1). We term the most suitable CN, a *selected CN*.

Once a user reaches a selected CN for a service, further message exchanges occur. First, the selected CN may request input parameters/data for the service (if the service requires input, *e.g.,* a stream of frames in video annotation). Then, the selected CN executes the invoked service using the required input data. Once the service execution is complete, the user retrieves the execution results. Figure 3 illustrates the exchanges between a mobile user and a selected CN for service invocation.

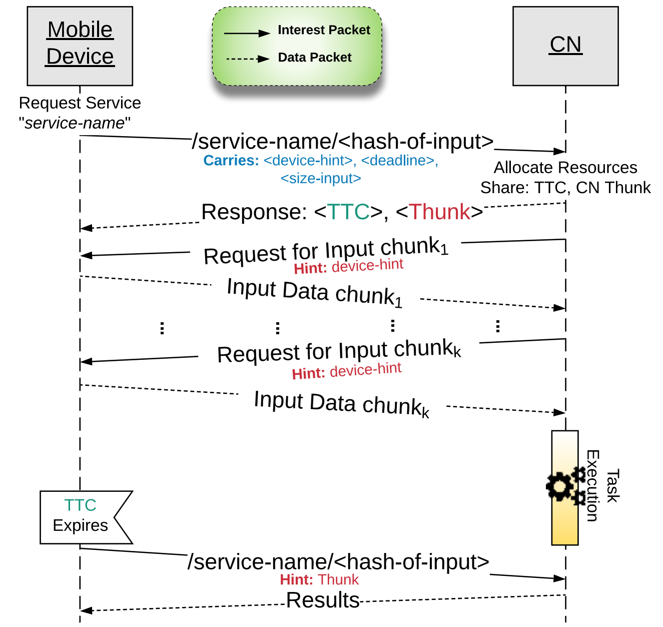


Fig. 3: Service invocation design. The Interest/data exchanges between a mobile device and a CN are illustrated. The device invokes an edge service and retrieves the results when the service execution is complete.

Service Invocation Initialization: A user initializes the service invocation by sending a signed request with a name: "`/service-name/<hash-of-input>`". The first part of the name indicates the invoked service and the second one includes the hash of the input data (*e.g.,* a hash of an input file) to leverage cached service results in the network[2]. The request also carries the size of the input data, a deadline for the service execution, and the "forwarding hint" of the user device[3]–an identifier used by the selected CN to reach the user device and retrieve input parameters for the invoked service. Thus, Interests carry both "what" (name of data) and "where" (forwarding hint) to retrieve the data from, aiding the routers with forwarding.

Upon the reception of a service request, the selected CN first authenticates the user by verifying the request signature. It then verifies that it has enough resources (*e.g.,* CPU and/or memory) to execute the requested service within the requested deadline. If the selected CN does not have adequate resources, it sends a Negative Acknowledgment (NACK) to notify the network [21] (*e.g.,* this can happen when edge conditions are changing fast). Then, the network can forward the request to another CN. For instance, in Figure 1, if CN_1 cannot execute the "`/Yolo`" service, it sends a NACK, which is received by Router C. C then forwards the request towards CN_3. In case of non-availability of resources in the entire edge network, the device sends the request to a distant Cloud.

If the selected CN has sufficient available resources, it first saves the service execution state, deadline, estimated Time To Completion (TTC), and size of data, and then allocates resources for the service execution. The selected CN sends

[2]The name of the initial invocation request may contain components specific to the service naming conventions. We present examples in Section VII.

[3]In NDN, there is no source identifier that can be used to reach a user, thus a hint helps the network route to the user that initiated the service invocation.

a response that includes the TTC and a *thunk* [15] for the task back to the device. The thunk is a name used by devices to fetch execution results from the selected CN. Without a thunk, users may not be able to reach the same CN (*e.g.,* if the network does load balancing) that executed a task in order to fetch the results (*e.g.,* requests for results for the "`/Yolo`" task in Figure 1 may be forwarded to CN_3 instead of CN_1).

Input Parameter Passing: After sending its response, the selected CN also sends Interests to fetch service input data (*e.g.,* the frame that users need to annotate). In some cases, input data may not be needed (*e.g.,* request for current time) or can be appended to the initial service invocation request (*e.g.,* provide a route to a given destination). In more general cases, input data is requested by the selected CN using the forwarding hint of the device. If the input size is large (it cannot fit into a single network layer data packet), multiple exchanges can be employed as illustrated in Figure 3.

Service Execution and Result Fetching: Once the selected CN receives all input data from the user, it executes the service. Users wait until the TTC expires to request the results from the selected CN using the task thunk as a forwarding hint. In this way, requests for results reach the selected CN, which sends the results back to the user. If service execution is not complete when results are requested, the selected CN sends a new estimated TTC to the user.

Robustness to CN Failures: The network is aware of the paths to the CNs that offer each service. For example, in Figure 1, Router C is aware that "`/Yolo`" is offered through interfaces 1 and 2. If CN_1 fails, C detects the failure, since requests for "`/Yolo`" forwarded through interface 2 will timeout. C will conclude that CN_1 has failed, thus forward future requests for "`/Yolo`" through interface 1 towards CN_3.

C. Service Reachability Through Self-Learning

In broadcast-based self-learning [27], the first packet for an unknown path (route) across the network is broadcast. When a response is received, forwarding information is created at the routers, so that future packets towards the same destination are sent through the "learned" path. In the context of NDN, self-learning has been proposed for the discovery of a single path to a single producer [32]. However, in the context of edge computing, a given service may be offered by multiple CNs, introducing the challenge of learning paths from users to all the CNs that offer a service.

To this end, we propose a multi-producer (multi-CN) self-learning mechanism, which enables the network to dynamically discover multiple network paths to all the CNs that offer a given service. This mechanism may be enabled when a service is invoked for the first time, since the network will not be aware of which CNs offer the requested service and how to reach them. Through this mechanism, edge routers are able to create FIB entries and associate each entry with one or more outgoing interfaces.

Given that paths to different CNs may be of a different length or delay, the self-learning process may last arbitrarily long. To avoid delays during service invocation, our mechanism first discovers one of the CNs that offers the requested service. This CN allocates resources and follows the protocol of Figure 3 for seamless service invocation. In the background, the network discovers paths to all the CNs that offer the requested service, so that future requests for this service can be forwarded to all the available CNs as needed.

Multi-Producer Self-Learning Design: A router might receive a service request (*e.g.,* Router A receives a request for "`/Yolo`" in Figure 4) that it does not know how/where to forward (*i.e.,* it does not have any forwarding information for this service in its FIB). In such cases, the router initiates the self-learning process by adding a *self-learning* tag to this request. It also creates a duplicate request with a *self-learning-duplicate* tag. The request with the *self-learning* tag will be used for the discovery of a route to a CN offering the service to make sure that users can invoke edge services without any delay. The duplicate request, created by the router, initiates a delay tolerant discovery of all the CNs that offer the service which may last arbitrarily long (*e.g.,* paths to different CNs may be of a different length or delay).

While both Interests are broadcast towards all the CNs offering "`/Yolo`", only a single CN allocates resources and replies with a data packet to a *self-learning* Interest[4]. All CNs offering "`/Yolo`" reply to the *self-learning-duplicate* Interest without allocating resources (CNs not offering the requested service reply with a NACK). Each of these two Interests creates a separate PIT entry in the network as shown in Figure 4, where Router A created the duplicate Interest ("`/Yolo/self_learning_dup`") to gather information on all the CNs running "`/Yolo`". Responses to the *self-learning* Interest ("`/Yolo/self_learning`") are sent back to the user via interface 1 of Router A, however, responses to the duplicate Interest are consumed by A and are not forwarded to the user.

Routers wait for a response to the *self-learning-duplicate Interest* from each interface they forwarded the Interest through before consuming the PIT entry. Once routers receive responses through all interfaces (or after a pre-defined timeout period), they aggregate the responses into a single data packet, sign this packet, and forward it to their downstream. This "aggregated" data packet consumes the PIT entry for the *self-learning-duplicate* Interest. In Figure 4, Router D waits for a response from CN_2 and CN_3. Once both responses are received (CN_2 will send a NACK, since it does not offer "`Yolo`"), D forwards a single response that contains only the positive response of CN_3 back to Routers C and B; B forwards the response back to A. C might have already received a response from CN_1. Once C receives the response from D, it aggregates the two responses and forwards them back to A. Given that A is the router that initiated the self-learning process, once it receives responses from B and C, it satisfies its PIT entry for the *self-learning-duplicate* Interest, without forwarding the responses back to the user.

Routers keep track of the interface, through which a response to a self-learning and a self-learning-duplicate Interests was received. They use the response to the former to create a new FIB entry for the requested service ("`/Yolo`" in our

[4]The selection of this CN can be determined through the service synchronization process (Section V-B), or be pre-configured by the network administrator.

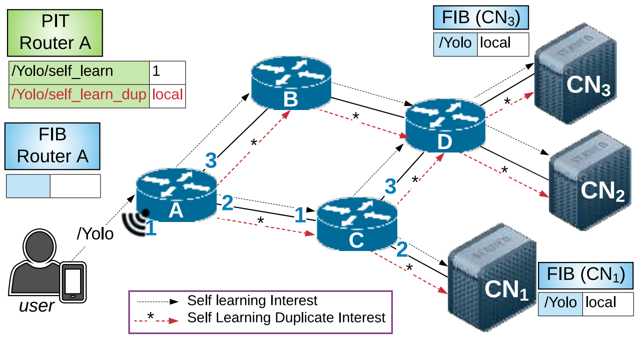


Fig. 4: Multi-producer self-learning mechanism; Router A creates and sends a duplicate self-learning Interest to find routes to all CNs that offer service *"/Yolo"*.

example) through the interface this response was received (interface 2 for Routers C and A). This enables routers to identify a network path from the user to the CN that responded (CN_1 in our example). Routers use the responses to the self-learning-duplicate Interest to: (i) add more next-hops to an existing FIB entry (*e.g.,* Router C adds a next-hop through interface 3 that can reach D), or (ii) create a new FIB entry if the routers were not on the path between the user and the CN that responded to the self-learning Interest (*e.g.,* Router D creates a new FIB entry towards CN_3). Once this process is done, routers have multiple paths to the multiple CNs offering the service and can forward future service requests to all the available CNs.

D. Adaptive Resource-Aware Forwarding

Operating conditions at the edge are highly dynamic (*e.g.,* the user invocation rates may rapidly change, CNs may reach their processing capacity). Routers need to be aware of the current conditions of the edge resources, so that they can forward (by selecting one among the many available outgoing interfaces) service requests to one among the many CNs offering a service based on one or more objectives. For example, if the objective is to equally distribute load among CNs, the network needs to be aware of how loaded each CN is. If the objective is to minimize energy consumption, the network needs to know the energy consumption of each CN.

The administrator of an edge network can define a number of objectives (*e.g.,* CN energy consumption, computation power of CNs, CN load, CN storage) that routers should take into account when deciding how/where to forward a service request. This can be formulated as a multi-objective optimization problem, which routers solve to rank their interfaces (next-hops) towards the offered services. In this way, routers can forward requests to the most suitable CN for each service as determined by their solution to this optimization problem. For instance, in Figure 1, this solution helps Routers A and C to rank interface 2 as the best forwarding option for "`/Yolo`". Note that this *ICedge* component is general-purpose and the optimization mechanism of choice is plug-and-play.

Making the Network Resource-Aware: To enable resource-aware request forwarding, the network needs to know the state of CNs (*e.g.,* load, energy consumption, storage resources). We call this information *utilization information*, which is used by routers as input to their multi-objective optimization. The frequency at which the network updates the utilization information is important and may vary based on the rate of requests, popularity of services, etc. We call a message that contains the utilization information of one or more CNs for one or more offered services *a utilization update*. We propose two mechanisms, a *proactive* and a *reactive* one, for the network to acquire up-to-date utilization information from CNs.

Proactive Utilization Updates: CNs periodically (every Update Period–UP) broadcast a utilization update to the network. For example, CN_2 in Figure 1, broadcasts an Interest with name "`/util/`CN_2`/s1/s2/../sn/util-info`". The prefix "`/util`" allows the network to interpret the message as a utilization update. The name contains the CN sending the update (*e.g.,* CN_2), a list of services, $\{s_1, s_2, \cdots, s_n\}$, installed and running at this CN, and the CN's current utilization information. When routers receive an update, they update the utilization information of the services executed by the CN sending the update. To limit the overhead caused by updates, CNs perform scoped flooding. Each update is propagated for a certain number of hops in the network based on a Time-To-Live (TTL) value set by the CN sending the update.

Reactive Utilization Updates: Routers update the utilization information on a per service basis according to the rate that users request each service. As a result, the network updates utilization information more frequently for popular services. Routers keep track of the number of invocations for any given service. For every IPU (Invocations Per Update) forwarded invocations for a service, routers generate an additional Interest that they tag as a "utilization" request to update the service's utilization information. This Interest is multicast to all the CNs offering the service. In Figure 1, for example, for every 30 invocations of "`/Yolo`" that Router A receives from users, A creates an additional utilization Interest that is multicast to CN_1 and CN_3.

The CNs send a utilization update in response to a utilization Interest. Utilization update responses are aggregated by routers into a single response in the same way as responses for self-learning-duplicate Interests (Section VI-C). Once routers receive a utilization response, they update the utilization information for the service.

Services invoked infrequently might be associated with outdated utilization information. To mitigate the negative impact in such cases, we attach to each update a *Freshness Period (FP)*, which is set by the CN sending the update. In addition to the utilization information, routers store the freshness period of an update. If a service is invoked after the expiration of the freshness period of its last update, routers send a utilization Interest for the service.

VII. Compute Reuse

A. Problem Statement

Data gathering and execution of tasks at the edge can be complex and compute-intensive. Often, users share needs, thus request "similar" tasks, especially when they share a given context or environment [29]. For instance, users attending the same game in a stadium would often request similar information, such as a player's best videos/stats, or visitors in a museum would request annotation of paintings or scenes already requested by other visitors. The reuse of previously executed computations at the edge could reduce the utilization of edge resources, resulting in an increase of the edge network *capacity* (*e.g.,* in terms of the number of user requests that the edge can serve). In this section, we explore the benefits and challenges of designing network-based mechanisms to facilitate the reuse of edge computing tasks.

B. Network-Based Compute Reuse

To take advantage of compute reuse, we argue that application-defined metadata must be visible to the network. This allows the network to make "informed decisions" and forward tasks to CNs that may be able to reuse (fully or partially) previously executed task output for current requests. In IP-based frameworks, the required application-layer metadata is invisible to the network. In *ICedge*, which runs on top of NDN, this metadata can be encoded in application-defined names, which are visible to the network and are used as the identifier for communication purposes between users and CNs.

The high level idea of *ICedge*'s compute reuse design is to *enable the network to seamlessly dispatch "similar" tasks towards the same CNs by simply looking at the service requests names*. This involves a trade-off between the need of the network operations to be general enough to cover the needs of *any* application and the need of the operations to be specific enough to forward service requests based on application-specific metadata in order to facilitate reuse. To tackle this tradeoff, we propose a set of general-purpose compute-aware naming conventions (Section VII-B1), which can be used by a variety of edge applications. Each convention provides instructions to the application on how to encode the necessary metadata in the names of its service requests, so that *ICedge* can facilitate reuse. This is achieved through a compute-aware forwarding scheme per convention, which is deployed on network routers[5] (Section VII-B2).

1) Compute-Aware Naming Conventions: We define a set of compute-aware naming conventions, which indicate how edge service providers or network administrators have clustered the computation of tasks at CNs. Such clustering can be pre-configured or dynamically determined based on traffic, resources, etc. To facilitate the use of compute-aware conventions, we extend the design presented in Section VI-B, so that users initialize the service invocation process by sending a request with a name "`/<service-name>/<compute-aware-convention>/<forwarding-scheme>/<input-hash>`". The "`forwarding-scheme`" component indicates the *ICedge* forwarding scheme for this convention as we discuss in Section VII-B2. The hash of the input enables reuse of results for the same task with the same input from in-network caches. Users discover the conventions during the discovery phase (Section V).

We define a preliminary set of compute-aware convention examples, which do not represent an exhaustive list. We

[5]Forwarding schemes can be installed on routers by the network administrator or can be dynamically retrieved from CNs.

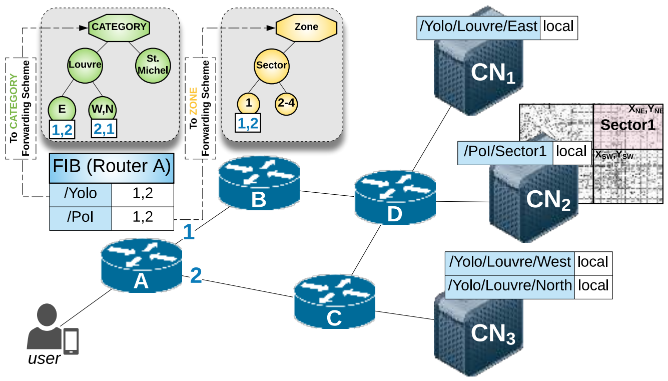


Fig. 5: Example of *ICedge*'s reuse mechanism. Interests are forwarded based on their names by compute-aware forwarding schemes, which aim to maximize the reuse of parts of previously executed tasks.

envision that new conventions will be created as new edge applications with different reuse needs emerge.

- "`/<building_X/floor_y/Room_z>`" for clustering of tasks based on buildings, floors, and/or rooms. This convention can be used by image annotation applications, so that annotation requests for pictures taken in the same building, floor, and room are forwarded by *ICedge* to the same CNs.
- "`/<X_coordinate/Y_coordinate>`" for clustering based on a location on a map. This convention can be used by location-based applications. For example, applications that request information about points of interests on a map, so that requests within a certain sector of the map are forwarded by *ICedge* to the same CNs.
- "`/<config_X/Param_Y>`" for clustering based on specific configuration and/or parameters. For example, applications that require matrix multiplication (*e.g.,* augmented/virtual reality, video games) can indicate the matrix dimensions along with a multiplier (*e.g.,* "`/20x20/3times`" for the multiplication of a matrix of size 20x20 to itself thrice). Requests are forwarded to CNs based on the dimensions of the matrices to be multiplied (*e.g.,* requests for matrices with dimensions from 1×1 to 30×30 are forwarded to one CN).

2) Compute-Aware Forwarding Schemes: Each naming convention is associated with a compute-aware forwarding scheme, which applications include in the name of their requests. This scheme enables the network to forward similar tasks to the same CNs by making routers aware of the logic that task computation has been clustered at CNs. In the rest of this section, we present three sample schemes to illustrate service request forwarding under different clustering use-cases.

CATEGORY: This scheme allows for task clustering based on disjoint categories, such as annotation of scenes in different rooms of a museum. Routers receiving an annotation request, *e.g.,* "`/Yolo/Louvre/East/R123/CATEGORY/<input-hash>`", utilize this scheme to dispatch tasks based on a given building ("`Louvre`"), area ("`East`" aisle), and room ("`R123`"). Tasks belonging to the same category reach the same CN, maximizing the reuse potential. Routers implement longest prefix match for the highest granularity–*i.e.,* building, area, or room. Routers learn which CN is responsible for which building, area, or room during self-learning (Section VI-C).

In the example of Figure 5, when requests for service "`/Yolo`" for pictures taken at the Louvre arrive at routers, such requests are dispatched to the category forwarding scheme. This scheme selects the most suitable next-hop based on a longest-prefix match of the request name ("`/Yolo/Louvre/East`" in our example, which reaches CN_1) on the information maintained by the scheme. Any additional processing required for reuse purposes happens in the forwarding scheme module of routers. As a result, the size of FIB stays the same with and without reuse, allowing routers to provide fast path forwarding to traffic that may not require reuse.

ZONE: Zone forwarding enables clustering of tasks based on locations or regions, such as ZIP codes, grids, or sectors on a map. For instance, a user looking for Points of Interest (PoI) around the Eiffel Tower through its coordinates $(X = 2.29, Y = 48.85)$, can use the naming convention "`/PoI/X/Y/ZONE/<input-hash>`". This request triggers the zone scheme at routers, which guides the request towards the CN that executes tasks for a map sector, where coordinates $(X = 2.29, Y = 48.85)$ fall into. In the self-learning phase, routers receive the coordinates of the map area that each CN is responsible for. In the example of Figure 5, the network forwards the request for "`/PoI`" around the Eiffel Tower to CN_2, since we assume that the coordinates in the request name fall into "`Sector1`"; a sector is defined as a square map area represented by (X_{NE}, Y_{NE}) and (X_{SW}, Y_{SW}), its north east and south west corner coordinates respectively.

CONFIG: In the configuration forwarding scheme, a CN can be responsible for a range of parameters for a given service. Let us consider a matrix multiplication example (omitted from Figure 5), where CN_1 performs multiplications of matrices with dimensions from 1×1 to 30×30. As a result, a request with a name "`/multiply/20x20/3times/CONFIG/<input-hash>`" for the multiplication of a matrix (with hash "`<input-hash>`") of size 20×20 to itself three times will be forwarded to CN_1.

3) Estimation of Task Completion Time with Compute Reuse: When compute reuse is utilized, the estimation of task execution times might not be accurate prior to CNs receiving the input data of the task. To tackle this problem, two mechanisms may be applied: (i) the CNs can explicitly notify users when a task is completed, allowing users to request the execution results as soon as they become available, or (ii) the CNs estimate the completion time of tasks based on the service popularity. This sometimes results in CNs underestimating the completion time. As a result, users may request the results before the task execution is completed. The CNs, at this point, can let users know about the updated completion time. We have implemented and experimented with both mechanisms, concluding that the choice of one or the other does not have a noteworthy impact on the *ICedge* performance.

VIII. Evaluation

In this section, we present the evaluation of *ICedge* via an extensive simulation study. Our evaluation is two fold. First, we disable reuse and evaluate the performance of our service invocation design, which includes a comparison of our proactive and reactive utilization update approaches (Section VIII-C1). Then, we enable reuse to evaluate its feasibility and benefits as well as compare *ICedge* to an IP-based centralized approach, which emulates a software defined network (Section VIII-C2).

A. ICedge *Implementation*

We implement *ICedge* using the ndn-cxx library to ensure compatibility with the NDN router prototype [2]. The main components of our prototype are the following: (i) a *user daemon* interacting with mobile applications, (ii) a network component, which can be dynamically deployed and configured at NDN routers, and (iii) a *CN daemon* that interacts with the CN framework. The user daemon invokes the edge services requested by applications. The *ICedge* network component contains plug-n-play forwarding schemes that are responsible for self-learning operations, handling of utilization updates, forwarding of service requests to CNs, and compute reuse. The CN daemon is responsible for initializing the execution of edge services, synchronizing with other CNs, participating in the service and naming convention discovery process, and responding to self-learning and utilization requests.

B. Experimental Setup

For our simulations, we run NDN directly on top of MAC layer (IEEE 802.11n for wireless and IEEE 802.3 for wired connections). We present the 90^{th} percentile of the results collected after 10 trials. We use the NetworkX library [14] to generate 50 random edge network topologies of size 20 to 30 interconnected nodes.

For each topology, we randomly select 10 nodes to act as CNs and we experiment with two traffic profiles; *light* and *heavy* traffic, consisting of 10 and 100 randomly distributed users respectively. Each user has a wireless connection to a topology node through an Access Point (AP). All users are mobile, following the random walk mobility model within square-shaped cells, while they can also move from the cell of one AP to another. We set the number of cores per CN to 12, while varying the processing speed of each CN. Our complete simulation setup and topology characteristics are presented in Table I. We assume that the execution of each task occupies one core for a given time period. If all the CNs for a service

TABLE I: Simulation setup and topology characteristics.

Parameter	**Value**
Link delay	10ms
Wired link bandwidth	10Gbps
Wired link loss probability	1%
Wireless link bandwidth	10Mbps
Wireless link loss probability	5%
Hops between users and CNs	[3,6]
Average node degree	3.2
Number of CNs	10
Number of services	100
Services per CN	$[30, 60]$
User service request rate	15 request/min (Zipf)
Service execution deadlines	$[500, 5000]ms$
Service execution time	$[300 - 4500]ms$ (Zipf)
Cores per CN	12

are occupied, users invoke services on the Cloud. We assume that the round-trip delay to/from Cloud is 200ms and that the Cloud executes tasks $3\times$ faster than edge CNs.

NDN-based Simulations: We ported our *ICedge* prototype into the ndnSIM simulator [22], which is based on the ns-3 network simulator to experiment with *ICedge* at scale. ndnSIM features software integration with the real-world NDN prototypes offering high fidelity simulation results. At the beginning of each experiment, there is no service reachability and utilization information in the network. We apply the multi-producer self-learning mechanism to establish service reachability (Section VI-C) and the mechanisms (proactive and reactive) for the propagation of service utilization information in the network (Section VI-D).

IP-based Simulations: We developed an IP-based centralized prototype system, which we refer to as *dispatcher-based approach*, and we ported it to ns-3. This system consists of a centralized application-layer entity called a task dispatcher. The dispatcher periodically receives information about the utilization and previously executed tasks (for compute reuse purposes) by CNs. User requests are forwarded to the dispatcher, which distributes them to CNs based on utilization and/or compute reuse potential. We place the dispatcher at most two hops away from CNs. The centralized scheme resembles to Software-Defined Networking (SDN), where the central entity represents a network controller at the edge.

Instrumenting the centralized task dispatcher: Our goal is perform a fair performance comparison between the SDN-like centralized (task dispatcher) and a fully distributed (*ICedge*) solution. To achieve the *ICedge* features (*e.g.,* adaptive task forwarding, compute reuse), the centralized solution needs to be augmented with awareness of the communication endpoints (*e.g.,* IP addresses of the CNs offering each service) and the communication context (*e.g.,* semantics of the computational tasks) to enable computation reuse. We extended the typical SDN tuple semantics, so that the dispatcher understands the semantics of the adaptive forwarding of tasks. However, to prevent the dispatcher from being a single point of failure all the functions mentioned above need to be replicated across redundant dispatchers. We present results for a single task dispatcher and replicated dispatchers in Section VIII-C.

C. Results

Metrics: We consider two main metrics for the evaluation of *ICedge*: (i) *task completion time*, and (ii) *normalized overhead*. The task completion time is measured as the time elapsed between users sending their request for task execution and receiving the execution results from CNs. The normalized overhead is measured as the ratio between the volume (in bytes) of overhead messages and the volume (in bytes) of service invocation traffic. The volume of overhead messages includes: (1) the volume of network traffic transmitted for the propagation of service reachability and utilization information (Sections VI-C and VI-D), (2) the volume of traffic transmitted during the sync process among CNs for service discovery (Section V), and (3) the volume of traffic generated due to NACKs sent by CNs when a service request is forwarded to a fully loaded CN (Section VI-B). Service requests are forwarded to CNs based on two factors: (i) CN load, and (ii) CN processing power. In other words, the network forwards requests to the CN that can execute them as fast as possible, while it also avoids overloading CNs that are fully utilized.

TABLE II: Evaluation parameters for proactive and reactive mechanisms (numbers in bold are nominal values used when the value of a parameter is fixed).

Parameter	Variable	Value
Update Period (Proactive)	UP	$\{15, \mathbf{30}, 60\}s$
Scope of flooding (Proactive)	TTL	$\{2, \mathbf{3}, 4\}hops$
Invocations Per Update (Reactive)	IPU	$\{15, \mathbf{30}, 60\}$
Freshness Period (Reactive)	FP	$\{15, \mathbf{30}, 60\}s$

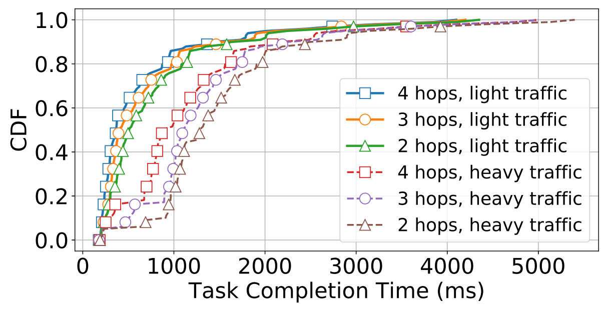
1) Evaluation of ICedge *with Different Service Utilization Mechanisms:* In this subsection, we disable reuse and evaluate our two proposed service utilization mechanisms (*i.e., proactive* and *reactive*) in isolation. Then, we compare their performance and discuss the trade-offs in different network and traffic settings. Table II presents a list of evaluation parameters and the values used for each parameter in our experiments. We evaluate the impact of one parameter at a time; when we vary a given parameter, the remaining parameters are equal to their nominal values (numbers in bold in Table II).

Proactive Service Utilization Mechanism: We evaluate the impact of the flooding diameter (scope of flooding in terms of number of hops) and the frequency of updates on the performance of the proactive approach. In Figure 6a, we present the distribution of task completion times for varying scoped flooding Time To Live (TTL) values of service utilization updates. For light traffic, increasing the scope of flooding from two to four hops results in 4-8% lower completion times. In such cases, the CNs are rarely fully loaded; as a result, any CN offering a service has the resources to execute it as well. For heavy traffic, CNs are typically under full load, therefore, increasing the scope of flooding helps the network forward a request to the available CN with the highest processing power, resulting in 13-28% lower task completion times.

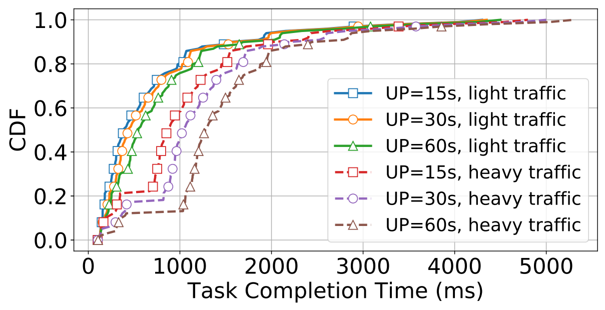
In Figure 6b, we present the task completion time for a varying Update Period (UP). The results show that frequent updates help the network maintain up-to-date service utilization information, resulting in 3-7% and 13-36% lower completion times as we increase the update frequency for light and heavy traffic respectively.

In Figure 7a, we present the overhead for varying flooding scopes and periods of utilization updates. The results show that the performance benefits of increasing the scope of update flooding and/or the frequency of updates comes at the cost of increased overheads especially in cases of light traffic loads. The normalized overhead reaches 0.5 (*i.e.,* the volume of overhead traffic is 50% of the volume of the user service invocation traffic) for frequent updates (with 3 hops of scoped flooding) and 0.54 when updates propagate 4 hops into the network (with an update period UP=30s).

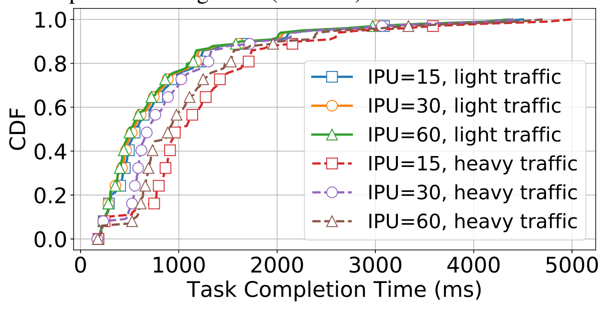
Reactive Service Utilization Mechanism: In Figure 6c, we present the task completion times when a service utilization update is performed for every 15, 30, and 60 invocations (denoted as Invocations Per Update or IPU for short). For light traffic loads, completion times are almost independent of IPU



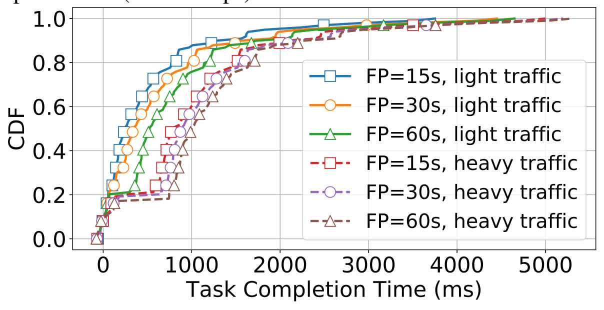
(a) Proactive approach, varying the scope of service utilization update flooding TTL (UP=30s)



(b) Proactive approach, varying the service utilization update period UP (TTL=3hops)

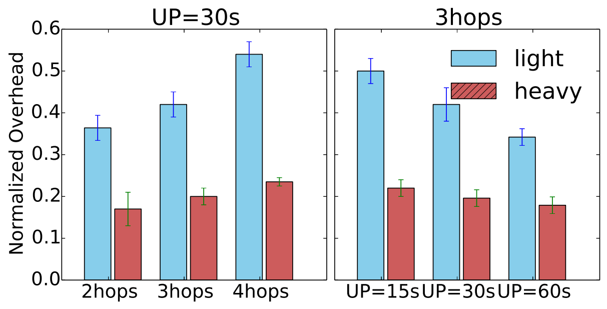


(c) Reactive approach, varying the number of service IPU (FP=30s)

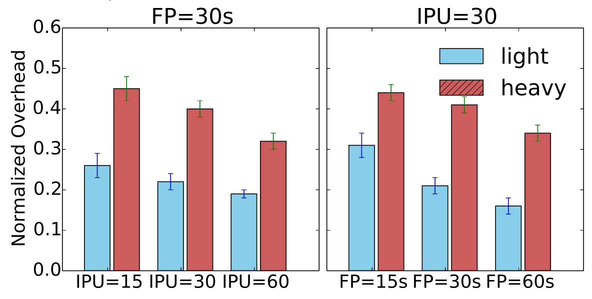


(d) Reactive approach, varying the FP of service utilization updates (IPU=30)

Fig. 6: Task completion time results of *ICedge*'s service invocation design (markers in CDF figures do not represent actual data points, but are only used for better readability).



(a) Proactive approach, using various utilization update parameters (TTL and UP)



(b) Reactive approach, using various utilization update parameters (IPU and FP)

Fig. 7: Overhead results of *ICedge*'s service invocation design: (a) proactive, and (b) reactive approaches.

(less than 4% impact); most utilization updates are triggered due to the expiration of their Freshness Period (FP), since not enough traffic is observed by the network to trigger updates. For heavy traffic loads, frequent utilization updates (*i.e.,* low IPUs) can improve completion times by 9-34%.

In Figure 6d, we present the task completion time results for a varying freshness period of utilization updates. As we described above, in cases of light traffic, utilization updates are mainly triggered due to expiration of their FPs. Therefore, the freshness period can heavily impact performance (6-41% based on our results). In cases of heavy traffic, updates are mainly triggered due to the large traffic volume observed by the network, therefore, the impact of FP on the completion time is less than 12%.

In Figure 7b, we present the overhead while varying the number of invocations per update (IPU) and the freshness period of each update (FP). The overhead ranges from 0.16 to 0.34 for light traffic and from 0.32 to 0.45 for heavy traffic.

Trade-offs: Our results indicate that each approach has its own merit and trade-offs. In terms of task completion time, the proactive approach achieves 6-17% lower times in most setups, since CNs proactively inform the network about their utilization and the services they can execute. On the other hand, in the reactive approach, routers request the utilization of a service only when it is actually used, achieving 14-238% lower overheads than proactive for light traffic. For heavy traffic, the reactive approach results in 26-48% higher overhead than proactive, since services are heavily used.

2) Evaluation of ICedge *With Network-based Compute Reuse Enabled:* We consider two evaluation parameters of compute reuse to model different task and application profiles: (i) the *reuse ratio*, and (ii) the *probability of reuse*. The *reuse ratio* (R_u) for a given task is defined as the ratio of computation resources that can be reused compared to those that need to be utilized for the execution of a task. For instance, $R_u = 40\%$ indicates that the task can save up to 40% of resources (thus time of execution) due to reuse. The *probability of reuse*, $P(reuse)$, indicates the probability for a given task

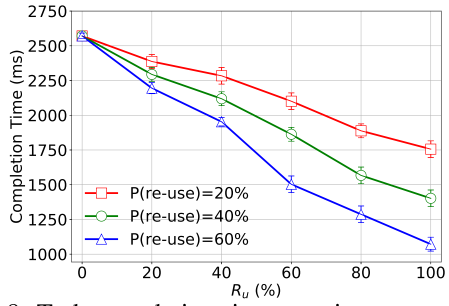


Fig. 8: Task completion time, varying reuse ratios.

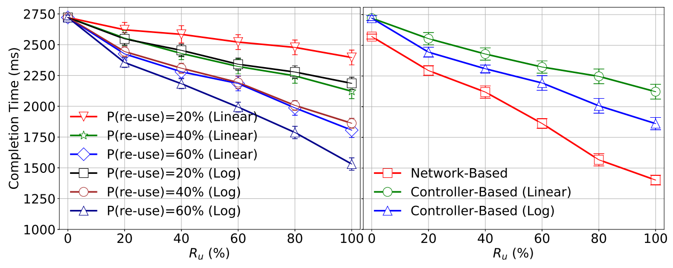


Fig. 9: Task completion time for dispatcher-based reuse and comparison with network-based reuse. Completion time for dispatcher-based reuse, varying reuse ratios (left). Comparison between dispatcher-based and network-based reuse (P(reuse)=40%) (right).

to find previously executed results of similar tasks at a CN. For instance, if $P(reuse) = 20\%$, then out of 100 tasks, only 20 can benefit from reuse (reuse ratio applies).

We vary these parameters and present the task completion time results in Figure 8. The results show that *ICedge* can reduce completion times by 1.14-$2.51\times$ due to compute reuse either at the CNs or in the network (through cached results). Note that the testbed deployment of *ICedge* (Section IX) showed that completion times can be further reduced (up to $50\times$) due to compute reuse.

Comparison to the Dispatcher-Based Solution: CNs periodically notify the dispatcher about their utilization and previously executed tasks. The dispatcher stores the reuse information onto a local database. Users send their requests to the dispatcher and the dispatcher performs a lookup operation on its database to identify which CN is the one that can offer compute reuse. The dispatcher ensures that the CN identified through this lookup is not fully loaded and forwards the service request to it. The size of the dispatcher reuse database is the aggregate of the size of the reuse database of each CN.

In Figure 9 (left), we present the task completion time for the dispatcher-based design for varying compute reuse ratios. We experiment with different probabilities of reuse (20%, 40%, and 60%). As expected, the completion time is lower as we increase the reuse ratio and the probability of reuse. In Figure 9 (right), we compare the completion time of the dispatcher-based reuse design to *ICedge*'s reuse design (reactive service utilization mechanism) for probability of reuse equal to 40%. The network-based approach results in 6-33% lower completion times than the dispatcher-based approach. This is due to the fact that *ICedge* takes advantage of application-defined naming conventions to identify which CN can offer reuse of previous results. On the other hand,

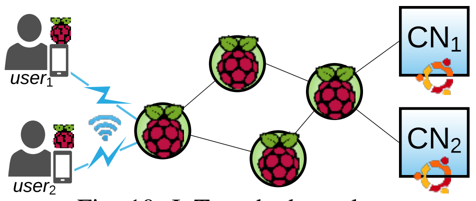


Fig. 10: IoT testbed topology.

CNs periodically notify the dispatcher about the tasks that they have previously executed and their utilization information. Even when these updates happen frequently, the dispatcher still makes sub-optimal decisions in the time between the updates being sent by CNs and being received by the dispatcher.

In Table III, we compare the overhead of the dispatcher-based and the network-based reuse approaches. The overhead of the dispatcher-based approach consists of: (i) the traffic volume of reuse notifications sent by each CN to the dispatcher, and (ii) the traffic volume of notifications for the utilization of each CN sent to the dispatcher. The results indicate that the overhead of the network-based design is about $1.27\times$ lower than the design based on a single dispatcher. However, having a single dispatcher per edge network introduces a single point of failure. To provide fault tolerance, the dispatcher needs to be replicated–for example, by introducing a second (backup) dispatcher. In this case, the overhead significantly increases, since each update eventually needs to received by both dispatchers. The results demonstrate that having two dispatchers results in $2.32\times$ higher overhead than *ICedge*.

IX. Proof-of-Concept Deployment

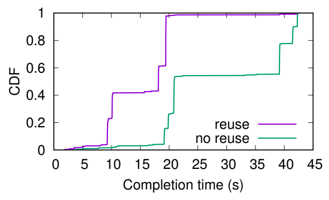
We consider a video surveillance application, where rotating security cameras in a city periodically send a video snapshot to CNs, which provide face detection services. Each "camera" instance produces snapshots of a fixed width w_s at a constant rate while moving from left to right for a full capture width of w_c (*i.e.,* the range of the camera's view). Once a camera takes a snapshot, it transmits it to a CN, which receives the snapshot and runs it through a Histogram of Oriented Gradients (HOG) face detection algorithm. The CN returns an array sorted by the coordinates of the faces it has detected. Assuming a reasonable camera rotation speed, there will be an overlap between two successive snapshots (quantified by an overlap percentage o). This overlap can be leveraged for reuse to avoid detecting faces on a fraction of the capture, where faces have already been detected in a previous capture.

A. Experimental Setup

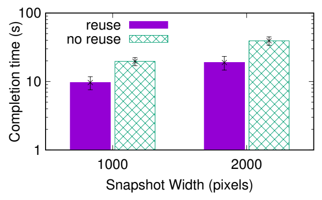
We deployed a *ICedge* prototype on a small-scale testbed illustrated in Figure 10, which consists of four Raspberry Pi devices acting as NDN routers, two 4-core 2.5GHz desktop machines acting as CNs, and two Raspberry Pi devices acting

TABLE III: Overhead, dispatcher and network-based reuse

Solution	Variant	Normalized Overhead
Network-based	w/o reuse	0.40
	reuse	0.41
Dispatcher-based	1 dispatcher	0.52
	2 dispatchers	0.95



(a) CDF



(b) Avg. completion time

Fig. 11: Testbed results for a *face detection* application illustrating (a) the CDF of task completion times, and (b) the task completion times for varying snapshot widths ($o = 0.7$)

as cameras. Our goal is to evaluate *ICedge*'s compute reuse mechanism in terms of the task completion time speedup.

For our experiments, we implemented the configuration naming and forwarding scheme presented in Sections VII-B1 and VII-B2. The two cameras are provided with full camera captures with dimensions $D_c = w_c \times h_c = \{14070 \times 1100, 13978 \times 1180\}$ pixels^2. For each camera, we vary $w_s = \{1000, 2000\}$ pixels and test $o = \{0.7, 0.8, 0.9\}$ for each w_s. We run five trials of sending the entire capture, snapshot-by-snapshot, for each $\langle D_c, w_s, o \rangle$ camera capture.

B. Experimental Results

We evaluate the performance of *ICedge* with the compute reuse mechanism enabled (*reuse*) and disabled (*no reuse*). We plot the CDF of the task completion time (*i.e.,* time elapsed between users sending their camera capture tasks and receiving the corresponding results) in Figure 11a for $o = 0.7$.

The results demonstrate that without reuse, 10% of tasks barely finish in 10s. However, with reuse more than 61% of tasks finish within this timeframe. Furthermore, 98% of tasks finish within 20s using *ICedge*'s reuse mechanisms. Without reuse, it takes more than twice the amount of time (42s) to execute the same percentage of tasks.

In Figure 11b, we present the average task completion time for $o = 0.7$ and varying snapshot widths. Note that the snapshot width is proportional to the computation complexity. While we confirm that with reuse *ICedge* outperforms the no reuse case for all tasks sizes and complexities, we show that the performance gain increases as the complexity increases achieving *more than 51% time reduction* compared to no reuse.

We have also performed a set of experiments with a fixed snapshot width $w_s = 2000$ and varying overlap percentages o. These experiments confirm that as o increases, the gain in time reduction increases by $2\times$ from $o = 0.7$ to $o = 0.9$ (results are omitted due to space limitations). Other application use-cases (*e.g.,* graphics applications such as virtual reality) that utilize matrix multiplication tasks were implemented and evaluated, demonstrating a reduction of completion times up-to $50\times$. We note that compute reuse may not be suitable for all computation profiles. Indeed, a task reuse assessment is needed to determine if the cost of reuse (*e.g.,* storage, lookup, energy, etc.) may outweigh the benefits.

X. Conclusion & Future Work

In this paper, we presented *ICedge*, an edge networking framework, designed with the objective to serve the application needs. *ICedge* features a fully distributed design consisting of building blocks that: (i) allow users to seamlessly on-board an edge network, (ii) dispatch requests for computation to edge resources in a timely manner, and (iii) provide network-based mechanisms for the reuse of previous computations.

While *ICedge* is off to a promising start, several open issues need to be addressed in the future. Currently, *ICedge* is designed to operate in a single edge network. This design can be augmented for large scale deployments, such as smart cities consisting of multiple edge networks interconnected via *gateway nodes*. Challenges for such deployments include handling user mobility and handovers, user data replication and protection, and economics. We envision placement of multiple gateway nodes per edge network for resilient handovers leveraging mechanisms to handle mobility [39].

As an ICN-based framework, *ICedge* also inherits privacy considerations related to ICN. Specifically, requests for edge services expose the requested services through their names. Several approaches to anonymize the names of Interests have been proposed [35], which can be utilized in *ICedge*. Offloaded tasks also expose the input data to CNs. We plan to explore cryptographic schemes, such as homomorphic encryption [9], for the task execution based on encrypted input data.

Finally, we plan to develop a set of *ICedge* libraries and programming abstractions, so that edge applications can leverage the *ICedge* functions. Our ultimate goal is to deploy *ICedge* in large-scale real-world settings, where it could interact with real-world applications and CN frameworks. To this end, we will also need to extend the currently available naming conventions and forwarding schemes for compute reuse to cover a wider spectrum of application requirements and investigate mechanisms for verifiable and secure computing.

References

[1] 10.2 million visitors to the louvre in 2018. https://presse.louvre.fr/10-2-million-visitors-to-the-louvre-in-2018/, May 2019.
[2] Alexander Afanasyev, Junxiao Shi, et al. NFD developer's Guide. Technical Report NDN-0021, NDN, 2015.
[3] Marica Amadeo, Claudia Campolo, and Antonella Molinaro. Ndne: Enhancing named data networking to support cloudification at the edge. *IEEE Communications Letters*, 20(11):2264–2267, 2016.
[4] Rajesh Balan, Jason Flinn, M. Satyanarayanan, Shafeeq Sinnamohideen, and Hen-I Yang. The case for cyber foraging. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, 2002.
[5] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.
[6] Eduardo Cuervo, Aruna Balasubramanian, Dae ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *MobiSys'10*, pages 49–62, 2010.
[7] Pedro de-las Heras-Quirós, Eva M Castro, Wentao Shang, Yingdi Yu, Spyridon Mastorakis, Alexander Afanasyev, and Lixia Zhang. The design of roundsync protocol. Technical report, Technical Report NDN-0048, NDN, 2017.
[8] Jason Flinn. Cyber foraging: Bridging mobile and cloud computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 7(2):1–103, 2012.
[9] Craig Gentry and Dan Boneh. *A fully homomorphic encryption scheme*, volume 20. Stanford University Stanford, 2009.
[10] Dennis Grewe et al. Information-centric mobile edge computing for connected vehicle environments: Challenges and research directions. In *Proceedings of the Workshop on Mobile Edge Communications*, pages 7–12. ACM, 2017.

[11] Peizhen Guo, Bo Hu, Rui Li, and Wenjun Hu. Foggycache: Cross-device approximate computation reuse. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 19–34. ACM, 2018.

[12] Peizhen Guo and Wenjun Hu. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. In *ACM SIGPLAN Notices*, volume 53, pages 271–284. ACM, 2018.

[13] Karim Habak, Ellen W Zegura, Mostafa Ammar, and Khaled A Harras. Workload management for dynamic mobile device clusters in edge femtoclouds. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, page 6. ACM, 2017.

[14] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[15] Peter Zilahy Ingerman and ET IRONS. Thunks. a way of compiling procedure statements with some comments on procedure declarations. Technical report, PENNSYLVANIA UNIV PHILADELPHIA, 1960.

[16] Michał Król et al. Rice: Remote method invocation in icn. *Proc. ACM ICN'18*, 2018.

[17] Michał Król, Spyridon Mastorakis, David Oran, and Dirk Kutscher. Compute first networking: Distributed computing meets icn. In *Proceedings of the 6th ACM Conference on Information-Centric Networking*, pages 67–77, 2019.

[18] Michał Król and Ioannis Psaras. NFaaS: named function as a service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, pages 134–144. ACM, 2017.

[19] Yuyi Mao et al. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358, 2017.

[20] Market Research Future. Edge Computing Market Research Report - Global Forecast 2023 . https://www.marketresearchfuture.com/reports/edge-computing-market-3239.

[21] Spyridon Mastorakis, Alexander Afanasyev, Yingdi Yu, and Lixia Zhang. nTorrent: Peer-to-Peer File Sharing in Named Data Networking. In *26th International Conference on Computer Communications and Networks (ICCCN)*, 2017.

[22] Spyridon Mastorakis, Alexander Afanasyev, and Lixia Zhang. On the evolution of ndnsim: An open-source simulator for ndn experimentation. *ACM SIGCOMM Computer Communication Review*, 47(3):19–33, 2017.

[23] Spyridon Mastorakis and Abderrahmen Mtibaa. Towards service discovery and invocation in data-centric edge networks. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2019.

[24] Abderrahmen Mtibaa et al. Towards resource sharing in mobile device clouds: Power balancing across mobile devices. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, pages 51–56, 2013.

[25] Abderrahmen Mtibaa et al. Towards edge computing over named data networking. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 117–120. IEEE, 2018.

[26] Abderrahmen Mtibaa, Khaled A Harras, Karim Habak, Mostafa Ammar, and Ellen W Zegura. Towards mobile opportunistic computing. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 1111–1114. IEEE, 2015.

[27] Radia Perlman. An algorithm for distributed computation of a spanningtree in an extended lan. In *ACM SIGCOMM Computer Communication Review*, volume 15, pages 44–53. ACM, 1985.

[28] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, NCM '09, pages 44–51, Washington, DC, USA, 2009. IEEE Computer Society.

[29] Mahadev Satyanarayanan. Edge computing a new disruptive force. The Second ACM/IEEE Symposium on Edge Computing, 2017.

[30] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[31] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, 2009.

[32] Junxiao Shi, Eric Newberry, and Beichuan Zhang. On broadcast-based self-learning in named data networking. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9. IEEE, 2017.

[33] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[34] Manolis Sifalakis, Basil Kohler, Christopher Scherb, and Christian Tschudin. An information centric network for computing the distribution of computations. In *Proceedings of the 1st international conference on Information-centric networking*, pages 137–146. ACM, 2014.

[35] Reza Tourani et al. Security, privacy, and access control in information-centric networking: A survey. *IEEE communications surveys & tutorials*, 20(1):566–600, 2017.

[36] George Xylomenos et al. A survey of information-centric networking research. *IEEE Communications Surveys & Tutorials*, 16(2):1024–1049, 2014.

[37] L. Zhang, , et al. Named data networking. *ACM SIGCOMM Computer Communication Review*, 44(3):66–73, 2014.

[38] Minsheng Zhang et al. Scalable name-based data synchronization for named data networking. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[39] Yu Zhang, Zhongda Xia, Spyridon Mastorakis, and Lixia Zhang. KITE: Producer Mobility Support in Named Data Networking. In *Proceedings of the 5th ACM Conference on Information-Centric Networking*. ACM, 2018.

[40] Zhiyi Zhang et al. An overview of security support in named data networking. *IEEE Communications Magazine*, 56(11):62–68, 2018.

Spyridon Mastorakis, Ph.D., joined the University of Nebraska, Omaha in August 2019 as an Assistant Professor in Computer Science. He received his Ph.D. in Computer Science from the University of California, Los Angeles (UCLA) in June 2019. He also received an MS in Computer Science from UCLA in 2017 and a 5-year diploma (equivalent to M.Eng.) in Electrical and Computer Engineering from the National Technical University of Athens (NTUA) in 2014. His research interests include network systems and protocols, Internet architectures (such as Information-Centric Networking and Named-Data Networking), and edge computing.

Abderrahmen Mtibaa, Ph.D., is currently an Assistant Professor at Department of Computer Science in the University of Missouri–St. Louis. Prior to that he has occupied several research positions including a visiting assistant professor at the Computer Science department in New Mexico State University; a Research Scientist at Texas A&M University; and a Postdoc in the School of Computer Science at Carnegie Mellon University. He is an author in more than 70 journal and conference papers with more than 1300 citations. His current research interests include Information-Centric Networking, Networked Systems, Social Computing, Personal Data, Privacy, IoT, mobile computing, pervasive systems, mobile security, and mobile opportunistic networks/DTN.

Jonathan Lee is a first-year student at Duke University studying Computer Science and Statistics. He will graduate with a BS in Computer Science and Statistics in May 2023, where he is planning to continue on to graduate school and attain his Ph.D. in Computer Science. His research interests include the design and optimization of computer networks, specifically with regard to edge computing, and various aspects of computer security, in which he will continue to explore in the coming years.

Satyajayant Misra, Ph.D., is an associate professor in Computer Science at New Mexico State University. He completed his M. Sc. in Physics and Information Systems from BITS, Pilani, India in 2003 and his Ph.D. in Computer Science from Arizona State University, Tempe, USA in 2009. His research interests include security, privacy, and resilience in wireless networks, the Internet, IoT/CPS, and supercomputing. He has served on several IEEE journal editorial boards and IEEE/ACM conference executive committees. He has authored more than 80 peer-reviewed IEEE/ACM journal articles and conference proceedings, which have received over 4500 citations. More information can be obtained at www.cs.nmsu.edu/∼misra.