DICer: Distributed Coordination for In-Network Computations

Uthra Ambalavanan
uthra.ambalavanan@de.bosch.com
Robert Bosch GmbH, Bosch Research
Germany

Dennis Grewe
dennis.grewe@de.bosch.com
Robert Bosch GmbH, Bosch Research
Germany

Naresh Nayak
naresh.nayak@de.bosch.com
Bosch Research, Robert Bosch GmbH
Germany

Liming Liu
fixed-term.liming.liu@de.bosch.com
Robert Bosch GmbH, Bosch Research
Germany

Nitinder Mohan
mohan@in.tum.de
Technical University of Munich
Germany

Jörg Ott
ott@in.tum.de
Technical University of Munich
Germany

ABSTRACT

Application domains such as automotive and the Internet of Things may benefit from in-network computing to reduce the distance data travels through the network and the response time. Information Centric Networking (ICN) based compute frameworks such as Named Function Networking (NFN) are promising options due to their location independence and loosely-coupled communication model. However, unlike current operations, such solutions may benefit from orchestration across the compute nodes to use the available resources in the network better. In this paper, we adopt the State Vector Synchronization (SVS), an application dataset synchronization protocol in ICN, to enhance the neighborhood knowledge of in-network compute nodes in a distributed fashion. As such, we design distributed coordination for in-network computation (`DICer`) that assists the service deployments by improving the resolution of compute requests. We evaluate the performance of `DICer` against NFN and observe an increase in the resource utilization at the edge and a reduction in the request completion time.

CCS CONCEPTS

• **Networks → In-network processing**; **Network management**; **Network control algorithms**; **Network design and planning algorithms**;

KEYWORDS

Distributed Coordination, Named Function Networking, In-Network Compute, Synchronization Protocols

ACM Reference Format:
Uthra Ambalavanan, Dennis Grewe, Naresh Nayak, Liming Liu, Nitinder Mohan, and Jörg Ott. 2022. DICer: Distributed Coordination for In-Network Computations. In *9th ACM Conference on Information-Centric Networking (ICN '22), September 19–21, 2022, Osaka, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3517212.3558084

ICN '22, September 19–21, 2022, Osaka, Japan

ACM ISBN 978-1-4503-9257-0/22/09…$15.00
https://doi.org/10.1145/3517212.3558084

1 INTRODUCTION

Next-generation networked applications, such as autonomous vehicles, Augmented Reality and Virtual Reality (AR/VR), the Internet of Things (IoT), etc. are characterized by their demand for transmission of huge volumes of generated data that need to be processed within strict latency bounds [18]. Such demands are further challenging in mobile scenarios where the underlying network and user location can be dynamic and unpredictable. Take, for example, *Electronic Horizon* [7] – an autonomous vehicle application realized today as a cloud-based virtual sensor that aggregates information from several sources such as road topography, and traffic conditions, or weather forecasts, etc. The benefits of centralization using cloud-based infrastructures, such as elastic scalability and low resource maintenance overhead, come at the cost of significant transmission times to send sensor data and receive the computed environmental model from cloud backends [4, 6].

Edge computing is a potential solution that aims to reduce transmission times and conserve network bandwidth by reducing data transfer volume towards cloud backends [5, 18]. For example, the ETSI Multi-access Edge Computing [12] standard introduces compute resources at the edge of the communication infrastructure enabling the execution of applications/services at the edge, closer to the consumers. However, provisioning the right application at the right edge server is a non-trivial problem that challenges service deployment and network management – especially in mobile scenarios where the end-users frequently (dis-)connect to the communication infrastructure [10].

Popular orchestration frameworks, such as Kubernetes [3], already support the distributed service deployment and management of cloud-native applications. With the advent of edge computing, such existing frameworks are being increasingly adopted for orchestrating services within edge infrastructures (e.g., KubeEdge [9], KubeFed [1], etc.). However, their inherent dependency on centralized state management (e.g. in Kubernetes off-shoots – `etcd`) challenges the placement of the cluster controller at the resource-constrained network edge [13]. Furthermore, such infrastructures make strong assumptions regarding the heterogeneity, geographical distribution and availability of underlying infrastructure, which further challenges the effective utilization of resources [19, 28].

An alternative to the centralized architecture may be exploring fully distributed in-network computing solutions based on data-oriented networking, such as Information-Centric Networking (ICN). ICN is inherently beneficial for service computations

at the edge since it supports location-independent addressing, a loosely coupled communication model and in-network caching [29]. Systems such as Named Function Networking (NFN) [25], Named Function as a Service (NFaaS) [14] enhance ICN by providing direct access to computations and dynamically generated compute results in the network. The compute request queries are expressed with name-based addressing such as λ-expressions in NFN and QoE-specific application classes in NFaaS, which are resolved by the resolution engine at the forwarding nodes. While the decentralized resolution of the suitable compute node helps to dispatch requests quickly, the local decision-making at every ICN node in the network may result in sub-optimal resource utilization due to a lack of broader network knowledge [24]. We argue that the decision-making of resolution engines can be significantly improved by augmenting the local knowledge to broader scope while retaining the benefits of using ICN based compute frameworks.

In this paper, we present `DICer` - a **DI**stributed **C**oordinat-ion mechanism to assist the resolution of in-network computations using the State Vector Synchronization (SVS) [20] solution in a distributed architecture. SVS is proposed for application dataset synchronization for the ICN architecture - Named Data Networking (NDN). `DICer` strives for improving the service placement at compute nodes by identifying unresolved yet popular function requests and instantiating them on off-path nodes with available compute resources, balancing the load between compute nodes. This is achieved by the augmented neighborhood knowledge of resolution engines. NFN is chosen as a basis for evaluating `DICer`. Our implementation and evaluation of `DICer` via network simulations also shows improvements in request completion time compared to the default behavior of NFN.

2 BACKGROUND AND RELATED WORK

In this section, we present the technical background on Named Function Networking (NFN) and the building blocks of State Vector Synchronization (SVS) protocol.

2.1 Named Function Networking (NFN)

Named Function Networking [25] extends NDN by providing dynamically access to generated compute results. While consumers in NDN get access to static data using *content identifiers* such as naming schemes, NFN provides access to compute results by expressing a compute workflow using the λ-calculus. An example of an NFN workflow for fetching the word count of an input file can be as follows.

```
func/word_count(data/input_file.txt)
```

Such λ-expressions are reduced and resolved to suitable compute nodes using the NFN resolution engine in the forwarding plane of every NFN node.

The NFN resolution engine performs different decisions based on the availability of function and data objects [27]. Each node forwards the compute request upstream if neither the function to execute nor the data to apply the function over is locally available. If the function or data is available, the node fetches the missing object, executes the computation, and responds with the computed result. If both function and data are available, the execution starts immediately. Figure 1 illustrates the different steps of the NFN

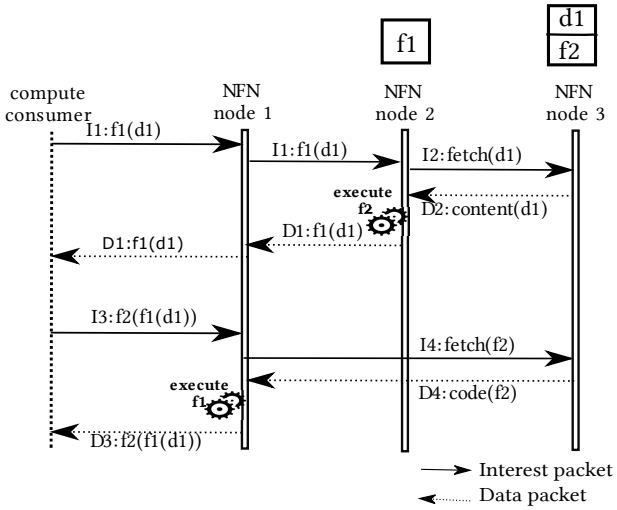


Figure 1: Named Function Networking: A consumer requests the network for a compute result, the NFN resolution engine at the compute nodes decide to either forward, fetch data/function code or execute depending on the local node knowledge [25].

resolution engine. The presence of data and function objects is mentioned above each node.

NFN `node1` forwards Interest `I1` upstream due to lack of function `f1` and Data `d1`. NFN `node2` which has the function `f1` fetches the data `d1` (known as *data drag*). Once the data `d1` is available at `node2`, it executes and responds with an NDN Data packet `R1` containing the result. `Node1`, on path to the consumer, stores a replica of `R1` for subsequent requests. In case such request (here `I3`) reaches `node1`, it has already the data and will drag the byte code for function `f2`. Finally, NFN `node1` executes the computation and responds back with the result (here: `R3`). The decisions to fetch content or functions are performed locally and independently at any forwarding NFN node within the resolution engine.

Though quick, flexible and scalable in making forwarding decisions, the resolution engine lacks the knowledge of the network at large, hence taking sub-optimal *local* decisions that favor the node instead of the network or consumers. For instance, `node 2` decides to fetch the data unaware of its size and lifetime, instantiate and execute the function. `Node 2` is also unaware of the resource availability at `node 3` (which also has an instance of the function already running), resulting in unnecessary network and compute resource utilization (refer Scherb et al. [24] for further limitations of NFN in IoT scenarios).

2.2 State Vector Synchronization in NDN

Several synchronization protocols have been proposed in ICN for synchronizing content across distributed applications (refer Moll et al. [21] for a survey on synchronization protocols in ICN). Typically, the proposed solutions consist of three building blocks: (i) *data representation model* – to represent the latest dataset, (ii) *protocol workflow* – to exchange the datasets across all relevant parties, and

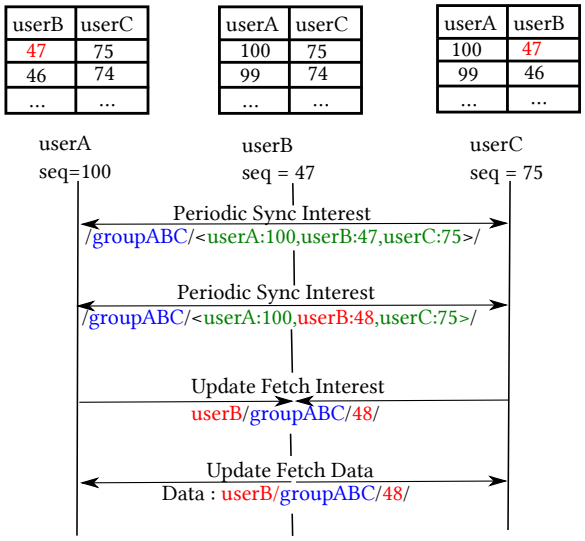


Figure 2: State Vector Synchronization [20].

(iii) *naming scheme proposals* – to provide access to datasets and to map the protocol flow.

In this work, we utilize the State Vector Synchronization (SVS) protocol at the core of our system design. SVS is a simple, lightweight protocol allowing periodic and event-triggered notifications. SVS also enables the storage of historical updates for data recovery. We now present the three building blocks corresponding to the SVS protocol: the data model, the SVS protocol flow, and naming schemes.

SVS Data Model. SVS introduces a vector-based data model inspired by Vector Clocks [8] to store states between different entities in the network. The vector contains an identifier for each subscribing entity (e.g., the application or user) and a sequence counter indicating the number mapped to the latest published state change at each entity within the vector. Such a structure allows all participants within a group to keep track of changes for every other participant.

SVS Protocol Flow and Naming Conventions. SVS combines aspects of the protocol flow and the naming schemes. To share the vector across participants, SVS describes synchronization groups comprising of all potential members using a common multicast group prefix ("`groupABC`" in Figure 2 written in blue). This prefix can be configured a-priori or negotiated between the group members. Besides the regular Interest/Data packet for data/compute queries, SVS introduces a `Sync-Interest` packet to notify group members regarding state changes. `Sync-Interest` packet does not expect any response in return.

The semantics of `Sync-Interest` include the multicast group prefix followed by a vector of entity-specific prefixes and their corresponding sequence counters denoting the latest changes. Figure 2 shows an example of `Sync-Interest` packets periodically multicasted. The second notification is initiated by `userB` to update the members of `groupABC` of the change corresponding to sequence number 48. Every group member stores the history of vectors, including the member prefix and its sequence counter in its datastore. Changes communicated in `Sync-Interests` are evaluated by each member against their locally stored state. In Figure. 2, `userB's` changes at the local state of `userA` and `userC` is only until sequence number 47. Hence the members `userA` and `userC` need to fetch and update its local state with the latest change from `userB` (i.e., sequence number 48). The node queries the latest data with the Interest/Data exchange procedure of the underlying NDN architecture (In Figure 2, `userA` and `userC` initiate an Interest to fetch the latest changes from `userB`).

Compared to other synchronization solutions in the literature, SVS offers a simplistic and lightweight approach to sharing data across network participants. Sequential naming and direct comparison of state vectors with the local state for detecting state change and retrieval is simple yet effective. SVS allows for simultaneous state change updates with minimum data dissemination delay (1.5 RTT). However, the use of state vectors in `Sync-Interests` to represent the entire dataset state challenges the scalability of SVS – as more members in a group would result in an explosion of the Interest names. State vector encoding and compression techniques can help to a limited extent [20].

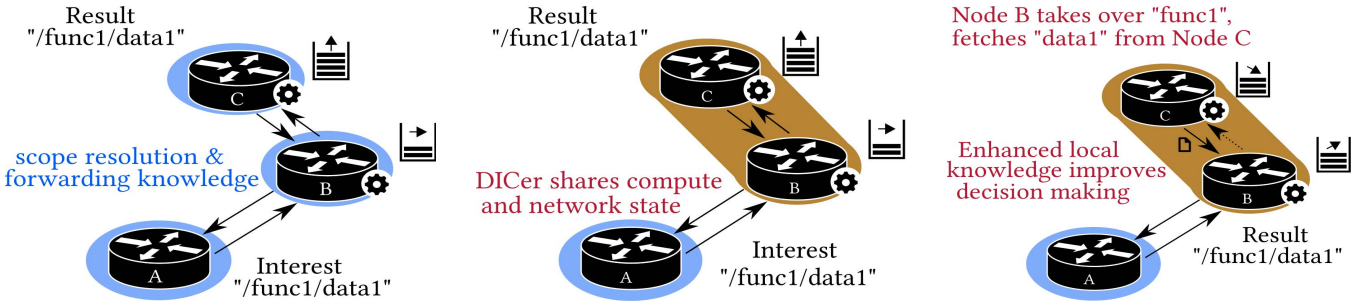
3 SYSTEM MODEL

We model the network topology as an undirected graph $G \equiv (V, E)$. While V represents the nodes (end systems along with the network elements like NFN forwarders) in the topology, $E \subseteq V \times V$ represents the set of edges between these nodes. The set of functions available in the system is denoted by the set $F = (f_1, f_2, \ldots f_n)$. Depending on their compute capability, the nodes in the topology, V, host the functions, F, to resolve and respond to the NFN compute requests. The orchestration map, O, represents functions hosted at nodes and determines function deployment in NFN systems, i.e., $O : V \times F \rightarrow \{0, 1\}$. $O(v, f) = 1$, if function f is hosted on node v, else it is 0. We intend to extend this problem to incorporate the placement of data objects in future work.

3.1 Problem Formulation

The NFN forwarders determine and adapt the orchestration map, O, based on the function requests passing through them, i.e., local knowledge. This knowledge scope for NFN resolution engine is denoted by the blue circles around nodes `A`, `B` and `C` in Figure 3a. The queue next to the nodes represents its compute resource consumption. In NFN, a compute request `"/func1/data1"` is resolved at node `C`. Since the node `B` is unaware of the resource availability at node `C`, it forwards the request to `C` when most of its resources are already occupied with other existing computations.

With `DICer`, we augment NFN nodes with additional metrics on current system properties such as resource utilization, function availability, etc. (represented as P) across the nodes V. We utilize these metrics to create an improved orchestration map using an off-the-shelf SVS synchronization protocol. Deployment of synchronization protocols for distributing control plane information requires the creation of node groups. The information synchronized between the members varies based on the group's scope definition. We model a synchronization group, g_i, as a tuple (V_i, t_i, P_i). Here, $V_i \subseteq V$ is the set of nodes in the group, t_i is the synchronization



(a) Plain NFN compute request processing. The local resolution engine evaluates the request and decides to execute or forward it.

(b) DICer allows to form synchronization groups to enhance local knowledge for resolution by sharing compute and network state.

(c) The enhanced knowledge allows different actions at compute nodes, e.g., taking over the execution of a function.

Figure 3: The system model of the DICer - distributed coordination concept.

frequency, and $P_i \subseteq P$ is the set of properties being synchronized across V_i. We represent the collection of all the synchronization groups in the network as SG.

Participating in bigger synchronization groups enables an NFN node to obtain a broader perspective of the system for better management of the orchestration map, O. However, the overhead of such synchronization traffic is directly proportional to the size of the group and the amount of data being synchronized. Alternately, restricting the scope of synchronization could result in a lack of relevant network knowledge for better placement decisions. Hence, using multiple synchronization groups is beneficial for conserving network overhead while flexibly gaining neighborhood knowledge.

In DICer, we integrate SVS and deal with the problem of discovering nodes V – determining the creation and scope of synchronization groups (V_i), the metrics that the corresponding group members exchange (P_i), and the synchronization frequency (t_i). With the synchronized information, DICer presents a coordination algorithm to achieve improved system performance (e.g., resource utilization, completion times, etc.).

3.2 DICer in Action

In Figure 3b, nodes B and C form a synchronization group. The nodes within the synchronization group exchange their node characteristics such as resource utilization, capability to host functions, functions instantiated, and function-specific information like the function's compute requirements, function to data dependency, etc. This is done with the aid of the SVS protocol as described in Section 2.

The shared knowledge obtained from synchronization groups is consolidated for taking certain types of actions to execute computations or share data applied to computations between the synchronized nodes. In Figure 3c, node B is aware of the compute load at node C after synchronization. During coordination phase, node B alleviates the compute load on node C by instantiating func1 and fetching data data1 from node C. Hence, node B equips itself to resolve the future requests of func1 over data1 with the help of DICer.

In the following subsections, we describe in detail the different stages of DICer namely *node discovery (V), synchronization group formation (SG), information sharing (P) and Coordination* (altering orchestration map O). The node discovery and synchronization group formation phases function asynchronously to find nodes entering and exiting the network scope dynamically. The information sharing happens periodically using the SVS synchronization, followed by the coordination phase.

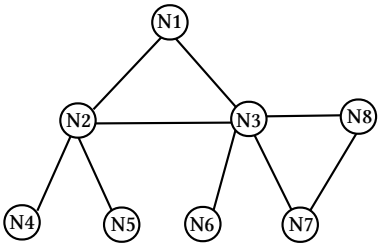
3.3 Discovery of Neighbors

To discover other compute nodes in the system and learn their computational state, DICer defines an active mechanism inspired by the NDN Named Link State Routing (NLSR) [11] protocol for discovering forwarders. Compute nodes frequently broadcast discovery messages using lightweight NDN Interest packets with a DICer specific namespace and provide a node-specific identifier (e.g. 'DICer/region/site/ nodeID/hello/'). The hierarchical naming scheme (based on NLSR), for node identifiers enables and restricts the discovery of nodes to a specific region/site. To avoid flooding the network, the discovery messages can also be restricted to a desired hop limit using the underlying NDN beyond which the Interest is no longer forwarded. The NDN forwarders prevent Interest loops by detecting duplicate discovery Interests from multiple paths. Additionally, each node maintains the state of discovered nodes to avoid redundant responses for already discovered nodes. We also adopt the mechanisms for detecting failures of remote nodes/links and their recovery from NLSR to DICer.

Each node receiving a discovery Interest from a new compute node responds back with the Data packet. The Data payload comprises the node-specific identifier along with the static properties of the node such as its compute configuration like CPU, GPU, memory, storage specifications and network configuration such as the hop distance and link status, etc. that needs to be exchanged just once. The nodes receiving the response can also infer the round trip time taken between the two nodes to estimate the path latency. Such static information shared during the discovery phase can be extended flexibly.

3.4 Formation of Synchronization Groups

As the nodes become aware of their neighbors from the discovery phase, they initiate synchronization group formation with discovered nodes. In DICer, we detect and group nodes based on hop



One Hop Groups	~~(N4,N2)(N5,N2)~~(N2,N1,N3,N4,N5) ~~(N1,N2,N3)~~(N3,N1,N8,N7,N6,N2) ~~(N6,N3)(N7,N3,N8)(N8,N7,N3)~~
Two Hop Groups	~~(N4,N1,N3,N5)(N5,N1,N3,N4)~~ ~~(N2,N6,N7,N8)~~(N1,N4,N5,N6,N7,N8) ~~(N3,N4,N5)(N6,N2,N1,N7,N8)~~ ~~(N7,N6,N1,N2)(N8,N1,N2,N6)~~
Three Hop Groups	~~(N4,N6,N7,N8)~~(N5,N6,N7,N8) ~~(N6,N4,N5)(N7,N4,N5)(N8,N4,N5)~~

Figure 4: The knowledge scope at nodes obtained from one, two and three hop groups for an exemplary mesh network is listed in the table. Subset groups are striked out in red.

distance; for example, we group a node with its one-hop network neighbors configured a priori. However, the group formation criteria can be abstract and network operator defined, as several other ways for determining the synchronization scope exist. For example, nodes can form groups based on their compute configuration (hardware support) or their centrality degree. While a highly capable node can benefit from the information synchronized and opt into multiple synchronization groups covering a broader network area, a less capable node can operate in the network while only synchronizing with its immediate neighbors to react quickly to local changes. This structure is not unlike Internet as we see today as transit Autonomous Systems(AS) providers in the core of the Internet peer with many other ASes and therefore receive BGP updates from them. On the other hand, serving ISPs in different countries (like M-Net in Germany) have a large customer cone but only peer with 1-2 transit ASes.

While synchronization may improve resolution decisions by the compute nodes, it comes with additional overhead. `DICer` uses multiple synchronization groups that operate at different frequencies and may share information at different granularity to balance the network overhead while minimizing information loss. In the following section, we illustrate the optimization of such groups to avoid redundant synchronization between the same nodes belonging to groups with different synchronization scopes.

3.4.1 Group Optimization. Figure 4 illustrates the different synchronization groups of `DICer` in an example network. Each node forms groups with neighboring nodes one, two or three hops away. The shorter the range of the synchronization group (e.g., one hop groups), the limited the number of nodes within such synchronization groups sharing fine-grained information at higher frequencies. Hence, the nodes within one hop group react to the needs of their peers promptly since they are well informed of any state updates.

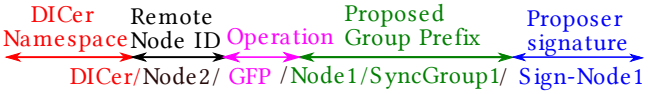


Figure 5: Group Formation Proposal sent by Node1 for forming a group "Node1/SyncGroup1" with Node2

Figure 6: `Sync-Interest` for update notification at Node1 for Data name N1-SG1 within Node1/ SyncGroup.

As the scope for synchronization group widens (two and three hop groups), the number of nodes that become part of such synchronization groups increases – restricting the synchronization frequency and reducing the granularity of shared information. This limits the network overhead rooting from large group synchronization while minimizing information loss.

Each group listed in the table in Figure 4 is formed from the perspective of the node that is in the first position in the group members list. For example, nodes {N2, N1, N3, N4, N5} are in one-hop group for node N2. Every node group that is a subset of a larger group is excluded as a synchronization group ((N4, N2) is a subset of (N2, N1, N3, N4, N5)). Additionally, every group sharing coarse information is also excluded if the members already form a group sharing fine-grained details (for instance, a two-hop group (N3, N4, N5) is excluded as it is already a part of one-hop group (N2, N1, N3, N4, N5)). Such group optimizations ensure no redundant synchronizations between nodes – limiting the number of group prefixes and the network overhead. Note that the synchronization groups are not restricted to those based only on distance or exclusive topologies. For instance, we could employ groups comprising safety-certified compute nodes for synchronizing on safety-critical function deployments. We envision researchers finding other group creation rules that best allow them to categorize nodes for optimizing their metrics of interest.

3.4.2 Group Formation. The node intending to form a group with a neighbor initiates a Group Formation Proposal (GFP) Interest. Figure 5 shows the Interest packet structure for GFP for forming a synchronization group between nodes `Node1` and `Node2`. `Node1` initiates an Interest with the `DICer` namespace, followed by the node it is willing to address (here `Node2`) using its identifier obtained during *Node Discovery*. The Interest is identified as GFP from its prefix. It is followed by the synchronization group identifier (here `Node1/ SyncGroup1`) and is signed by the node proposing the group formation, here `Node1`. This prevents unauthorized nodes from forming groups and fetching node-specific information.

The remote `Node2` uses this identifier to look for any existing groups under a different prefix with the same node and scope. `Node2` as a response to this Interest returns an acknowledgment and the name of the data that *will* be synchronized within the group after a random wait time on the receipt of GFP. If two nodes propose a group simultaneously, the node receiving the acknowledgment during this wait time suppresses its acknowledgment. The group is

successfully formed when the acknowledgment from the remote node is received. Therefore, the nodes in a group are aware of the group prefix and information synchronized within the group. The group is frequently updated as new nodes are discovered or lost within the scope.

3.5 Information Sharing

After group formation, the status changes at a node are updated using SVS periodic synchronizations (see Figure 6). A `Sync-Interest` is sent periodically with incremental sequence numbers in the event of an update. For instance, if a node becomes busy due to a sudden request peak, the other nodes of the synchronization group are notified of this change during the periodic `Sync-Interest` with increasing sequence numbers as seen from Section 2.2. The nodes receiving this `Sync-Interest` may respond with another Interest to fetch the changes mapped to the notification. The busy node responds to this interest with its node status such as, functions instantiated, functions requested, resource availability, etc. Thus, every node is aware of the other group members' status, and this information is used for performing service placement to assist the resolution decisions of NFN.

The synchronization overhead in `DICer` specifically arises from the information sharing phase, which can be reduced by controlling the synchronization periodicity and synchronized information. Additionally, the nodes can refrain from fetching every update change notification received from other group members. For instance, information exchange within a synchronization group can be many-to-one, i.e. every node in the group except the competent nodes refrains from sending the fetch Interest. At the end of this phase, every node acquires the necessary state information of its neighborhood.

3.6 Coordination & Decision Making

Different strategies can be employed during coordination based on synchronized optimization objectives and information. The coordination phase is invoked at each NFN node and local decisions are made using the enhanced knowledge. In `DICer`, we design a coordination algorithm that aims at balancing the compute resource utilization across different nodes in the network. A node may be a member of multiple synchronization groups ($sg \subseteq SG$), and the synchronized information is aggregated for decision making.

Algorithm 1 presents the pseudocode of the placement algorithm. We denote the set of functions instantiated at each node as a list F_{inst} and the compute resource availability as $ResAvail_{node}$. Each compute node can host a limited number of unique functions depending on its memory capacity, denoted by $HostCap_{node}$. The algorithm identifies functions whose requests are unresolved from each node (FUR_{node} - FunctionUnresolvedRequest at node) and was forwarded upstream. An NFN node decides to forward a function request upstream due to either lack of compute resources while processing the request, or a lack of the relevant function code and data at the compute node. The algorithm attempts to discover a node with the required resources and information (here, the function code) for resolving such requests within the synchronization group.

Algorithm 1 Distributed Coordination Algorithm

Require:
$SG, F_{inst}, ResAvail_{node}, HostCap_{node}, FUR$
$Constants : NodeBusy_{Thresh}, NodeFree_{Thresh}$

```
1:  function CoordinationPhaseHandler
2:      f_lru = getLeastRecentlyUsed(F_inst)
3:      if ResAvail_node ≤ NodeBusy_Thresh then
4:          F_inst.pop(f_lru)
5:      else if ResAvail_node ≥ NodeFree_Thresh then
6:          for ∀ sg ∈ SG do
7:              for ∀ gm ∈ sg do
8:                  if ResAvail_gm ≤ NodeBusy_Thresh then
9:                      FUR_node.push(FUR_gm)
10:                 end if
11:             end for
12:         end for
13:         sortDescending(FUR_node)
14:         for ∀ f in FUR_node do
15:             if HostCap_node! = 0 then
16:                 F_inst.push(f)
17:             else
18:                 F_inst.pop(f_lru)
19:                 F_inst.push(f)
20:             end if
21:         end for
22:     end if
23: end function
```

The algorithm uses thresholds for node resource availability ($ResAvail_{node}$) to identify the compute nodes as busy or free. A node is busy if the available resources are less than $NodeBusy_{Thresh}$ (line 3). On the other hand, the node is free if the available resources exceed $NodeFree_{Thresh}$ (line 5). Busy nodes look for idling functions (f_{lru}, the least recently used function) that can be evicted to save the compute capacity for other more frequently requested functions. These nodes forward the Interests of evicted functions upstream instead of resolving them (lines 3-4).

Free nodes identify unresolved function requests locally and at the busy group member nodes (gm is the group member of sg). They further aggregate it to FUR_{node} list (see line 6 - 12). This list is sorted in descending order (line 13) to act on functions that were frequently requested but rarely executed. If there is an available capacity to host new functions, the node instantiates them. If not, the node tries to evict any idling hosted function using the least recently used (f_{lru}) clearing strategy (lines 14-21). This is similar to the NDN's content store cache clearing strategy. On eviction of a popular function at a node by `DICer`, the FUR_{node} increases in the upcoming coordination cycle driving `DICer` to then place it on that node or its neighborhood. `DICer` thus maintains the resource consumption at every node between the $NodeBusy_{Thresh}$ and $NodeFree_{Thresh}$ levels.

The enforcement of DICer coordination decisions that involve enabling or disabling functions is reflected in routing with the help of solutions like automatic prefix propagation in NDN [15] which enables registering and deregistering the prefixes at the forwarders.

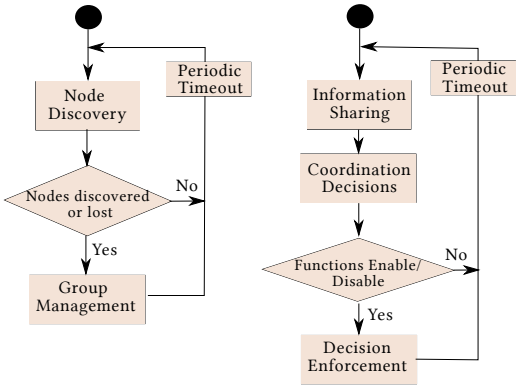


Figure 7: DICer flow of different stages. The flow on left is the periodic invokation of Node discovery and management. This is asynchronous to the flow described on the right which involves the periodic synchronization, coordination and decision making stages.

While replicating functions, the forwarders perform load balancing by with the help of forwarding strategies that forwards the requests to alternate faces advertising the prefix.

The flow of the above described DICer phases is illustrated in Figure 7. The flow on the right is the periodic node discovery phase that identifies nodes entering or leaving the network scope, followed by group management phase that reorganizes the nodes to their respective groups. This flow is asynchronous to the DICer information sharing and coordination process shown on the right. If the DICer coordination phase results in decisions to enable or disable a function, they are enforced before the periodic synchronization is invoked again.

4 IMPLEMENTATION

In order to evaluate DICer against NFN, we implement the algorithm in ndnSIM [17]. ndnSIM is an NDN protocol stack compatible with the network simulator ns-3 [23]. incSIM is an extension of ndnSIM supporting the NFN default resolution strategy FoX (Find-or-Execute) in the forwarding plane [2]. We choose the best route forwarding strategy, where the compute nodes determine the "best" execution node for the compute Interests. To implement DICer, we integrated the open-source SVS module [16] with incSIM. We also implement the stages of neighbour discovery, synchronization group formation and termination workflows to support multiple group synchronization updates between neighbors.

We implement two types of synchronization groups, long-range and close-range. The long-range synchronization group is created at each node with every other node present at three-hop distance. We chose this three-hop distance based on the hierarchical network topology used in the model. The close-range group is created with every node and its one-hop neighbours. Since the nodes two hops away are subsets of one-hop groups, we remove them during group optimization. The information synchronized during the close-range one hop synchronization is the node's resource availability, functions instantiated, function hosting capacity, the list of functions requested and executed in the previous synchronization cycle and the list of function requests forwarded upstream. The information exchanged during the long-range three hop synchronization is every node's aggregate information obtained during its close range synchronization, i.e., we share the average resource availability, the most popular and least popular functions at a node and its one-hop neighbours. This helps in sharing the crucial state of a zone (node and its one hop neighbors) across the larger network such that the neighbour zones are aware and prepared to handle a sudden influx of load for specific functions.

The close-range synchronizatons are invoked at twice the frequency of long-range synchronizations to handle the network overhead caused by the latter. At every node, the DICer coordination phase starts right after the completion of the close-range synchronization process. Based on the information obtained during synchronization, the periodic coordination performs a decision that alters the choice of execution node by NFN resolution engine.

5 EVALUATION

In this section, we present the evaluation setup used in the incSIM environment to test the performance of DICer system against NFN. We further present our results comparing DICer with NFN in the following subsection.

5.1 Evaluation Setup

We evaluate DICer on a 3-tier hierarchical topology network. It comprises consumers that generate data as well as request computed results. The compute nodes, equipped with the NFN resolution engine, resolve and execute these requests. We use 10-35 compute nodes to handle the load from 100-700 unique functions invoked by consumers. The nodes are interconnected by network links supporting a bandwidth range of 250Mbps-10Gbps and a propagation delay of 3ms-200ms (refer [26] for our motivation on network setup). The popularity of functions at the consumers follow a Zipfian distribution and are fetched from a central repository at the cloud. These functions are stateless and monolithic. The execution duration of functions range from 2 to 6 seconds in order to simulate both short running and long running functions while simulating sufficient load in the network. In order to prevent consumers from sending redundant compute requests, they are requested at 8 seconds interval to account for function execution duration as well as network delays before a data packet can reach a consumer including long running functions. However, the consumers request different functions at different time instances to avoid idle time in the network without request load. The number of functions that the compute nodes can execute and host in parallel depends on their compute configuration. Beyond this limit, new functions can be instantiated only after an existing function terminates. We choose the function to be terminated based on the Least Recently Used (LRU) clearing strategy among the idling functions. If all functions have an active instance running, the request for computing a new function is forwarded upstream. A compilation of the network setup and simulation parameters is presented in Table 1.

The compute request load is distributed amongst the consumers connected to the two edge sub-networks. The proportion of load generated by the two subnetworks is determined by the Imbalance Factor (IBF) in the range of 0 to 1. When the IBF is 0, the compute

Parameter	Values
Topology	3-tier hierarchical topology [26]
Network links	250Mbps – 10Gbps
no. compute nodes	10–35
no. consumer nodes	100
no. functions	100–700 unique functions
function execution duration	2s–6s
no. data objects	100
cache size	100000 $\frac{object}{node}$
Request interval R	8s
Close Range sync. interval	2s–50s
Long Range sync. interval	4s–100s
Coordination Interval	2s–50s
Sim Duration	750 seconds
Total no. runs	2700

Table 1: Table presenting the network and function characteristics of the evaluation set up.

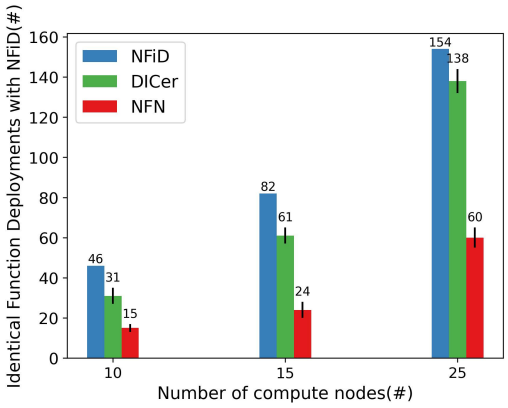


Figure 8: Comparison of the Orchestration Map generated by NFiD, NFN and DICer . The Y-Axis shows the number of functions deployed in DICer and NFN matching that obtained using NFiD

load is generated equally from the two subnetworks (no imbalance), while IBF = 1.0 implies that 100% of the compute load is generated from one of the two sub-networks, resulting in an imbalance. Therefore, with IBF-1.0, some of the compute nodes are extremely loaded while the other nodes are free. An example of an imbalanced compute load would be a university campus network that experiences a high compute request load during the day while the dormitory networks are busy in the evenings.

We incorporate the close-range synchronization between nodes separated by one-hop distance – synchronized at short intervals (in the range of 2s-50s) for testing the impact of scaleability. On the other hand, we establish long-range synchronization between nodes that are 3-hops away, synchronized at intervals twice of close-range synchronizations. The coordination decision making is periodically invoked after every close-range synchronization to ensure the network can quickly adapt to the changes. Our results in the next section are obtained from ≈ 2700 unique runs.

5.2 Results

We compare the network behaviour with and without DICer. We evaluate on the metrics - the generated orchestration map, completion time, DICer's stability on function placement changes and scalability of DICer with increasing network topologies and synchronization frequencies in order to understand the merits and demerits of DICer.

5.2.1 Orchestration Map. As discussed in the system model, orchestration map O, shows the functions deployed at each node. To evaluate the correctness of DICer's orchestration map, we compare it against a heuristical placement solution namely the Next Fit Decreasing (NFiD) algorithm. This algorithm sorts the functions based on their popularity in descending order as the function requests follow a zipfian distribution. The sorted functions are deployed iteratively onto compute nodes starting from the nodes closest to consumers, in the hierarchical network as long as the resource constraints at the nodes to host functions are met. The functions are placed only once and are not replicated.

The orchestration map generated by NFiD is compared against that generated by DICer and NFN. In an imbalanced network, NFN is unaware of 50% of the compute nodes leaving them un(der)utilized. The remaining nodes instantiate functions resulting in a map which is approximately one third similar to that of NFiD's function deployements. However, DICer is capable of detecting unused compute resources as well as unresolved functions. Thus the orchestration map generated by DICer is approximately two thirds similar to that of NFiD. This is shown in Figure 8 for different network scale. Additionally, DICer also replicates popular functions if the node at which it is currently deployed is unable to handle all the requests for the specific function.

5.2.2 Average completion time. The average completion time is defined as the round trip time since the consumer initiated a compute request until the receipt of its response. Completion time is a crucial metric in several safety critical applications that require lower completion times. The average completion time with and without DICer can be seen in Figure 9. DICerreduces in the average completion time compared to plain NFN. This is due to the increased utilization of resources at nodes closer to consumers that are idle in the plain NFN scenario due to its lack of knowledge about the closer, available, off-path nodes in the network. This is evident in a completely imbalanced network (ibf = 1.0), where one of the two edge subnetworks is idle while the other is completely loaded. With a balanced load, there is hardly any deviation in the behavior of DICer from that of NFN. Additionally, it can be observed that DICer is more beneficial in reduction of average completion time when the compute demand is higher with more unique functions in the networks raising the request load. DICer, with the added neighborhood knowledge, equips the nodes to take informed instantiation or termination decisions.

5.2.3 Stability of DICer decisions. In order to visualize the reaction time of DICer and to ensure DICer does not prevent the

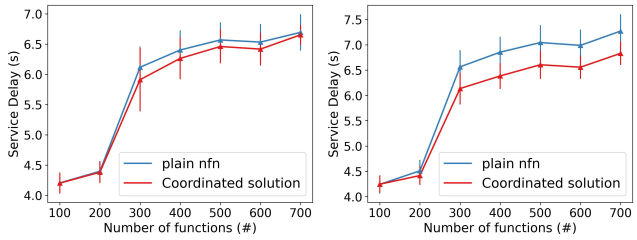


Figure 9: Average completion time with NFN and DICer for imbalance factors (IBF = 0, 1.0). DICer outperforms at higher loads in an imbalanced fashion (i.e. IBF=1.0).

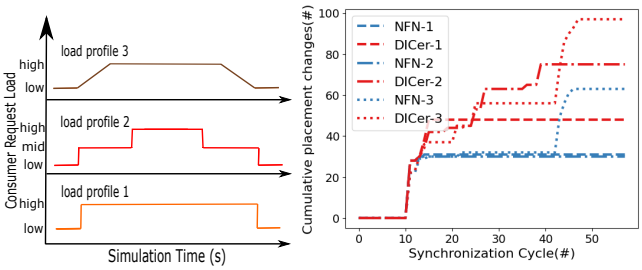


Figure 10: Function placement changes for different load profiles to evaluate the stability of DICer vs NFN.

system from reaching a stable placement state, we compare DICer and NFN (Figure 10) for three different load profiles for a completely imbalanced network. Load profile 1 is a static load where the load is constant for the entire duration of simulation. In Figure 10, NFN and DICer perform placement decisions during the start of the simulation. The initial number of placement changes taken by DICer, although stable, is greater than that of NFN. This is due to better awareness of the network infrastructure leading to increased decisions to instantiate or terminate services.

Load profile 2 shows a step increase and decrease in load at the middle of the simulation. NFN nodes have exhausted their host capacity after the first step increment and the further changes in the network load doesn't change the placement unless there are any idling hosted functions. Hence, even with increase in load, NFN placement does not change after gaining stability. However, with DICer, we notice a lot more placement changes taking effect during the middle of the simulation. The reason for more placement changes from DICer is due to the knowledge of the requested load and its efforts to spread them among the busy and free compute nodes. DICer takes longer to stabilize due to the delay in detection of busy/free nodes. Once the load is balanced across all nodes, the stable placement state is obtained.

Load profile 3 shows a gradually increasing load, followed by stable load and a gradually decreasing load. The behavior between DICer and NFN are quite similar except for the scale of placement changes. The increase in placement decisions at both NFN and DICer is due to increase in termination of idle functions as the load gradually decreases.

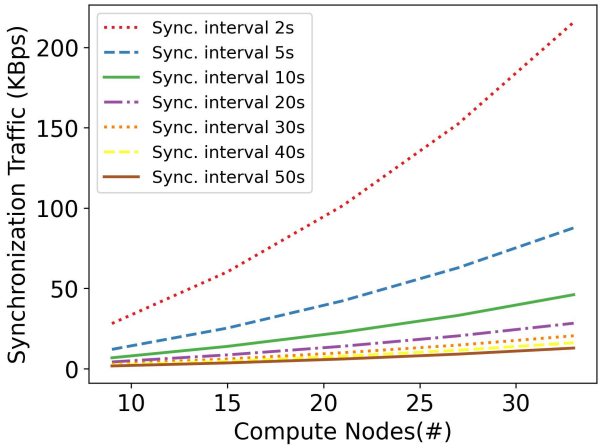


Figure 11: Scaleability of DICer with increasing network scale and synchronization intervals.

5.2.4 Scaleability. We also analysed the impact of synchronization frequency on the overhead of DICer on the network most of which comes from Sync-Interest and Sync-Data packets. The number of Sync-Interests and Sync-Data and the size of each are related to the number of nodes in each synchronization group as well as the synchronization frequency. Every cycle of update notification and retrieval involves transmitting two Interest packets and one Data packet. Currently, with periodic synchronization, excluding the discovery and group management overhead, six interest-data exchanges occur for every link connecting nodes for each synchronization update. The shorter the synchronization interval, more quick DICer is to react to changes in network load at the cost of increased network overhead. The decreasing synchronization interval shows a linear trend on the network traffic while the increasing number of compute nodes resulting in bigger synchronization groups shows a quadratic relation to the network overhead as seen from Figure 11.

5.2.5 Influence of synchronization interval. In order to evaluate the influence of synchronization interval on the completion time, we evaluated by varying the intervals from 2s to 50s on a network with the number of compute nodes ranging from 10 to 35 nodes. The lowest interval of 2s is chosen based on the lowest execution duration of the functions requested by consumers. It is observed from Figure 12 that with increasing compute nodes in the topology, the average completion time reduces for both NFN and DICer (from 6.1s with 9 nodes to 5.4s with 33 nodes) as more resources are available for computing. It is also seen that at higher synchronization interval (50s), the performance of DICer degrades and the completion time approaches closer to that of NFN, although never worse than NFN. At lower sychronization intervals, the completion time reduces and drops at its lowest at an interval of 5s. Reducing the synchronization interval below that has a negative impact on the completion time. At such low intervals, the synchronization overhead on the network is significant impacting the compute traffic. With more number of nodes in the topology, gradient of increase in completion time at 2s is steeper.

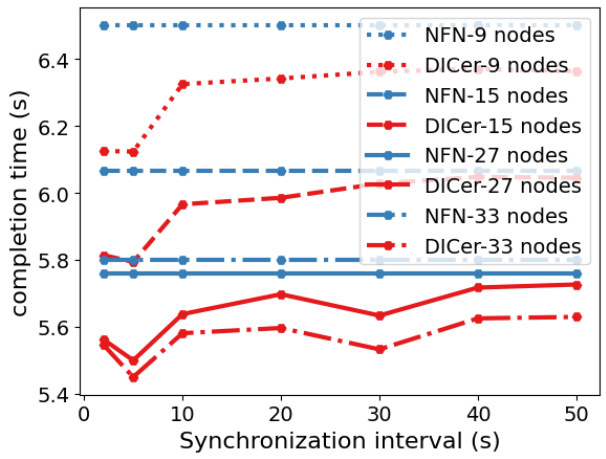


Figure 12: Performance of DICer with different synchronization intervals and network scale.

6 DISCUSSION AND FUTURE WORK

Coordination Algorithm : The current DICer algorithm only alters the placement of functions in the network. In the future iteration, the algorithm could be extended to gain knowledge of data distribution in the network and perform decisions as per the size and popularity of both data and functions. This would allow the nodes to decide whether to push/pull function or data for more efficient network usage.

Other ICN based in-network compute solutions : The current DICer system is currently implemented and evaluated against NFN. With regards to other ICN based in-network compute solutions like NFaaS, the focus on coordination algorithm as well as the information synchronized can be extended or modified accordingly. For instance, NFaaS already enables detecting popular functions and instantiating them with the help of kernel store. However, NFaaS is still unaware of its neighbourhood information of nodes which have not stored or advertised the prefix but have the compute capacity to do so. Thus, using DICer to form synchronization groups and coordinate among each other in the group could be beneficial for systems like NFaaS as well. As future work, we intend to extend DICer for other in-network compute solutions as well as evaluate DICer using real-world set ups.

Synchronization Overhead : Further extensions to DICer could be efficient synchronization overhead management using an appropriate synchronization dataset along with other methods such as opting for event-triggered instead of periodic synchronizations, ensuring disjoint group formations to avoid redundant synchronization communication between the same set of nodes over different scopes, use of compression techniques for the information shared, etc. Gaining neighborhood knowledge can be beneficial for detecting long-running functions instantiated at neighboring nodes and assisting the node by decomposing such functions into parallel subfunctions, speeding up the execution. However, such extensions will likely come at the cost of higher synchronization overhead within DICer.

DICer in Network Layer : As mentioned, the neighbor discovery in DICer is vastly inspired from NLSR, which also uses synchronization to gain the adjacency link-state information. As a result, our design choice raises the question of introducing DICer directly as a part of the NLSR protocol. However, we argue against such a merger for following reasons. While NLSR is a network layer protocol, DICer is widely application-specific and the groups formed or the information shared between coordinating nodes is configured on the application layer offering the required flexibility. More importantly, DICer is not restricted to only routing and forwarding decisions.

Security and Privacy : Security and privacy is a crucial research problem in ICN network architectures [22]. Authentication and authorization are significant techniques for imparting security in a system. Although DICer does not provide a full fledged solution on these, we present some initial directions for DICer. In DICer, we could take inspiration from the underlying NDN where producers transmit signed data packets and extend it by enabling group members to send signed synchronization interests and data to authenticate the change update notifications, if the operator of DICer demands for authenticity (eg: DICer employed in WAN instead of LAN).

Regarding the aspects such as authorization, the Named Access Control (NAC) [30] is promising for adoption in DICer, as it enables encryption and decryption of data at different granularities based on the access privileges of the nodes in synchronization group. The distribution of decryption keys can be incorporated when nodes are discovered and added to specific groups. Such methods add to the network overhead with additional inter-node messages and processing delays from encryption, decryption, etc at each node. DICer currently functions on the assumption that there is/will be a protocol in place to ensure that security aspects arising from multi-operator infrastructure are handled.

7 CONCLUSION

In-network compute frameworks based on ICN networks such as NFN are promising for distributed computing. In this paper, we present the limitations and the scope for improvements with in-network compute frameworks such as NFN's resolution strategies. For this, we adapt the SVS synchronization protocol for enhancing the node's knowledge to a broader network scope.The coordination algorithm at each node then manages the functions deployed locally by instantiating or terminating functions. DICer enables balancing the compute load between the coordinating nodes and helps reducing the completion time. We implement DICer and evaluate it against NFN with Find-or-Execute (FoX) resolution strategy using simulations and present the results and discuss on the crucial aspects of the system.

ACKNOWLEDGMENTS

This work was done within the EU CELTIC-NEXT project PICCOLO (Contract No. C2019/2-2) supported in parts by the German Federal Ministry of Economics and Climate (BMWK) and managed by the project agency of the German Aerospace Center (DLR) (Contract No. 01MT20005A).

REFERENCES

[1] 2022. *KubeFed.* https://github.com/kubernetes-sigs/kubefed Last accessed on 10/04/2021.

[2] Uthra Ambalavanan, Dennis Grewe, Naresh Nayak, and Joerg Ott. 2021. HYDRO: Hybrid Orchestration of In-Network Computations for the Internet of Things. In *11th International Conference on the Internet of Things* (St.Gallen, Switzerland) *(IoT '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3494322.3494331

[3] The Kubernetes Authors. 2021. *Kubernetes Components.* https://kubernetes.io/docs/concepts/overview/components/ Last accessed on 20/10/2021.

[4] Lorenzo Corneo, Maximilian Eder, Nitinder Mohan, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, Per Gunningberg, Jussi Kangasharju, and Jörg Ott. 2021. Surrounded by the Clouds: A Comprehensive Cloud Reachability Study. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 295âĂŞ304. https://doi.org/10.1145/3442381.3449854

[5] Lorenzo Corneo, Nitinder Mohan, Aleksandr Zavodovski, Walter Wong, Christian Rohner, Per Gunningberg, and Jussi Kangasharju. 2021. (How Much) Can Edge Computing Change Network Latency?. In *2021 IFIP Networking Conference (IFIP Networking)*. 1–9. https://doi.org/10.23919/IFIPNetworking52078.2021.9472847

[6] The Khang Dang, Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Jörg Ott, and Jussi Kangasharju. 2021. Cloudy with a Chance of Short RTTs: Analyzing Cloud Connectivity in the Internet. In *Proceedings of the 21st ACM Internet Measurement Conference* (Virtual Event) *(IMC '21)*. Association for Computing Machinery, New York, NY, USA, 62âĂŞ79. https://doi.org/10.1145/3487552.3487854

[7] dSPACE GmbH. 2010. *Developments on the Electronic Horizon.* https://www.dspace.com/shared/data/pdf/dspace_magazine/2010-2/english/dSPACE-Magazine_ADA_2010-02_en.pdf

[8] Colin J Fidge. 1988. Timestamps in Message-Passing Systems That Preserve the Partial Ordering,âĂİ. In *Proc. 11th Austral. Comput. Sci. Conf. (ACSC '88)*. 56–66.

[9] Cloud Native Computing Foundation. 2021. *KubeEdge project website.* https://kubeedge.io/en/ Last accessed on 06/08/2021.

[10] Fabio Giust, Vincenzo Sciancalepore, Dario Sabella, Miltiades C. Filippou, Simone Mangiante, Walter Featherstone, and Daniele Munaretto. 2018. Multi-Access Edge Computing: The Driver Behind the Wheel of 5G-Connected Cars. *IEEE Communications Standards Magazine* 2, 3 (2018), 66–73. https://doi.org/10.1109/MCOMSTD.2018.1800013

[11] AKM Mahmudul Hoque, Syed Obaid Amin, Adam Alyyan, Beichuan Zhang, Lixia Zhang, and Lan Wang. 2013. NLSR: Named-data link state routing protocol. In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*. 15–20.

[12] European Telecommunications Standards Institute. 2021. *ETSI Multi-access Edge Computing (MEC).* https://www.etsi.org/technologies/multi-access-edge-computing 2021-3-16.

[13] Andrew Jeffery, Heidi Howard, and Richard Mortier. 2021. Rearchitecting Kubernetes for the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '21)*. 7âĂŞ12. https://doi.org/10.1145/3434770.3459730

[14] Michał Król and Ioannis Psaras. 2017. NFaaS: Named Function as a Service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking* (Berlin, Germany) *(ICN '17)*. 134âĂŞ144. https://doi.org/10.1145/3125719.3125727

[15] Yanbiao Li, Alexander Afanasyev, Junxiao Shi, Haitao Zhang, Zhiyi Zhang, Tianxiang Li, Edward Lu, Beichuan Zhang, Lan Wang, and Lixia Zhang. 2018. NDN Automatic Prefix Propagation.

[16] StateVectorSync maintained by named data. 2021. *ndn-svs: State Vector Sync library for distributed realtime applications for NDN.* https://github.com/named-data/ndn-svs Last accessed on: 21/10/2021.

[17] S. Mastorakis, A. Afanasyev, and L. Zhang. 2017. On the Evolution of NdnSIM: An Open-Source Simulator for NDN Experimentation. *SIGCOMM Comput. Commun. Rev.* 47, 3 (2017), 19âĂŞ33. https://doi.org/10.1145/3138808.3138812

[18] Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. 2020. Pruning Edge Research with Latency Shears. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20)*. 182âĂŞ189. https://doi.org/10.1145/3422604.3425943

[19] Nitinder Mohan, Aleksandr Zavodovski, Pengyuan Zhou, and Jussi Kangasharju. 2018. Anveshak: Placing Edge Servers In The Wild. In *Proceedings of the 2018 Workshop on Mobile Edge Communications* (Budapest, Hungary) *(MECOMM'18)*. Association for Computing Machinery, New York, NY, USA, 7âĂŞ12. https://doi.org/10.1145/3229556.3229560

[20] Philipp Moll, Varun Patil, Nishant Sabharwal, and Lixia Zhang. 2021. A Brief Introduction to State Vector Sync. *Technical Report NDN-0073, Revision 2, NDN, Tech. Rep.* (2021).

[21] Philipp Moll, Wentao Shang, Yingdi Yu, Alexander Afanasyev, and Lixia Zhang. 2021. *A Survey of Distributed Dataset Synchronization in Named Data Networking.* Technical Report. Tech. Rep. NDN-0053, Revision 2, Named Data Networking.

[22] Kostas Pentikousis, BÃűrje Ohlman, Elwyn B. Davies, Spiros Spirou, and Gennaro Boggia. 2016. Information-Centric Networking: Evaluation and Security Considerations. RFC 7945. https://doi.org/10.17487/RFC7945

[23] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.

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

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

[26] Mohit Taneja and Alan Davy. 2017. Resource aware placement of IoT application modules in Fog-Cloud Computing Paradigm. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 1222–1228. https://doi.org/10.23919/INM.2017.7987464

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

[28] Aleksandr Zavodovski, Nitinder Mohan, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. 2019. ExEC: Elastic Extensible Edge Cloud. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking* (Dresden, Germany) *(EdgeSys '19)*. Association for Computing Machinery, New York, NY, USA, 24âĂŞ29. https://doi.org/10.1145/3301418.3313941

[29] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, KC Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named data networking. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 66–73.

[30] Zhiyi Zhang, Yingdi Yu, Alexander Afanasyev, Jeff Burke, and Lixia Zhang. 2017. NAC: Name-Based Access Control in Named Data Networking. In *Proceedings of the 4th ACM Conference on Information-Centric Networking* (Berlin, Germany) *(ICN '17)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3125719.3132102