Compute First Networking: Distributed Computing meets ICN

Michał Król
University College London/UCLouvain
michal.krol@uclouvain.be

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

David Oran
Network Systems Research & Design
daveoran@orandom.net

Dirk Kutscher
University of Applied Sciences Emden/Leer
dirk.kutscher@hs-emden-leer.de

ABSTRACT

Modern distributed computing frameworks and domain-specific languages provide a convenient and robust way to structure large distributed applications and deploy them on either data center or edge computing environments. The current systems suffer however from the need for a complex underlay of services to allow them to run effectively on existing Internet protocols. These services include centralized schedulers, DNS-based name translation, stateful load balancers, and heavy-weight transport protocols. In contrast, ICN-oriented remote invocation methodologies provide an attractive match for current distributed programming languages by supporting both functional programming and stateful objects such as Actors. In this paper we design a computation graph representation for distributed programs, realize it using Conflict-free Replicated Data Types (CRDTs) as the underlying data structures, and employ RICE (Remote Method Invocation for ICN) as the execution environment. We show using NDNSim simulations that it provides attractive benefits in simplicity, performance, and failure resilience.

CCS CONCEPTS

• **Networks → Naming and addressing**; **In-network processing**; *Network architectures*; Session protocols.

KEYWORDS

Information Centric Networks, Named Data Networking, in-network processing, naming, thunks

ACM Reference Format:
Michał Król, Spyridon Mastorakis, David Oran, and Dirk Kutscher. 2019. Compute First Networking: Distributed Computing meets ICN. In *6th ACM Conference on Information-Centric Networking (ICN '19), September24–26, 2019, Macao, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3357150.3357395

ICN '19, September24–26, 2019, Macao, China

ACM ISBN 978-1-4503-6970-1/19/09…$15.00
https://doi.org/10.1145/3357150.3357395

1 INTRODUCTION

Domain-specific distributed computing languages like LASP[17] have gained popularity for their ability to simply express complex distributed applications like replicated key-value stores and consensus algorithms. Associated with these languages are execution frameworks like Sapphire[32] and Ray[18] that deal with implementation and deployment issues such as execution scheduling, layering on the network protocol stack, and auto-scaling to match changing workloads. These systems, while elegant and generally exhibiting high performance, are hampered by the daunting complexity hidden in the underlay of services that allow them to run effectively on existing Internet protocols. These services include centralized schedulers, DNS-based name translation, stateful load balancers, and heavy-weight transport protocols.

We claim that, especially for compute functions in the network, it is beneficial to design distributed computing systems in a way that allows for a *joint optimization* of computing and networking resources by aiming for a *tighter integration of computing and networking*. For example, leveraging knowledge about data location, available network paths and dynamic network performance can improve system performance and resilience significantly, especially in the presence of dynamic, unpredictable workload changes.

The above goals, we believe, can be met through an alternative approach to network and transport protocols: adopting *Information-Centric Networking* as the paradigm. ICN, conceived as a networking architecture based on the principle of accessing named data, and specific systems such as NDN[33] and CCNx[2] have accommodated distributed computation through the addition of support for remote function invocation [6, 10, 11, 29] and distributed data set synchronization schemes such as PSync [34].

Introducing Compute First Networking (CFN) We propose CFN, a distributed computing environment that provides a general-purpose programming platform with support for both stateless functions and stateful actors. CFN can lay out compute graphs over the available computing platforms in a network to perform flexible load management and performance optimizations, taking into account function/actor location *and* data location, as well as platform load and network performance.

Use Case - Health Screening System We present a simple airport health screening system as a motivating use case for our framework. Such a system can be deployed using simple microphones or commodity mobile phones to detect people with highly-infectious pulmonary diseases before they board a plane [24]. Pulmonary ailments, including tuberculosis, cystic fibrosis, lower respiratory infection and over a hundred others [9], account for four of the top ten causes for death worldwide and coughing is a symptom of

many of these ailments [20]. An efficient screening system could substantially decrease spread of those diseases, but comes with several challenges.

Constrained listening devices can be placed in various points at the airport to collect sound samples, but the sound processing requires significant CPU power. This processing includes of eliminating any speech present in the audio for privacy protection while retaining the cough sounds [12]. The audio sample is then analysed to detect cough sounds [3] and extract multiple cough features such as "wetness" or "dryness" [4]. Finally, coughs with similar features sre clustered and analysed together, as coughing frequency could be an important indicator of multiple aliments [9].

Our framework can enable the mobiles devices collecting audio samples to use external computing resources with minimal code changes and allow deployment without any specialized hardware at the network edge.

CFN makes the following contributions:

(1) Marries a state-of-the art distributed computing framework to an ICN underlay through RICE[10]. This allows the framework to exploit important properties of ICN such as name-based routing and immutable objects with strong security properties.
(2) Adopts the rigorous computation graph approach to representing distributed computations, which allows all inputs, state, and outputs (including intermediate results) to be directly visible as named objects. This enables flexible and fine-grained scheduling of computations, caching of results, and tracking state evolution of the computation for logging and debugging.
(3) Maintains the computation graph using *Conflict-free Replicated Data Types* (CRDTs) and realizes them as named ICN objects. This enables implementation of an efficient and failure-resilient fully-distributed scheduler.
(4) Through evaluations using ndnSIM[14] simulations, demonstrates that CFN is applicable to range of different distributed computing scenarios and network topologies.

To the best of our knowledge CFN is the first system to combine modern distributed system programming concepts in the application layer with ICN in order to provide completely decentralized in-network computing. While scenarios such as the one describd above could be implemented by existing TCP/IP overlay distributed computing with centralized orchestration, the key CFN idea is to provide the same functionality (from an application developer perspective) without requiring all the additional mechanisms such as DNS, discovery mechanisms, application-layer RPC, and orchestration protocols. In the following, we describe how CFN leverages mechanisms of the ICN network layer to 1) reduce complexity and to 2) provide a tighter integration of the distributed computing and the network layer.

The rest of this paper is structured as follows. In Section 2 we explain the overall architecture of CFN. We then present the programming model and its relationship to function and data naming in Section 3. Section 4 describes the various components that make up the system, followed by our evaluation of CFN in Section 5. Related work is discussed in Section 6. We conclude with some discussion and plans for future work in Section 7.

2 CFN ARCHITECTURE OVERVIEW

This section describes CFN's key design goals, technical concepts, fundamental building blocks, and introduces the terminology used throughout the paper.

2.1 Design Goals

We summarize CFN's design goals as follows:

- to provide a general-purpose distributed computing framework;
- to flexibly exploit compute platforms in diverse scenarios, including edge computing environments;
- to enable multi-tenancy (sharing of resources among multiple, mutually untrusted application contexts);
- to support dynamic invocation of stateless functions as well as working with stateful actors;
- to support dynamic resource allocation and continuous adaptation to changing resource availability and load;
- to provide improved performance and efficient resource utilization through joint optimization of computing and networking resources; and
- to leverage the rich ICN network layer facilities, such as routing, forwarding, cryptographic integrity, and data placement (caching) for better performance, security, and failure resilience.

2.2 Key Concepts

CFN incorporates concepts from modern application-layer frameworks, is fully dynamic with respect to function/actor instantiation and invocation and can offload important functions to the network layer (for example, forwarding strategies for finding the "best" instance of a function in the network).

It provides abstractions and corresponding APIs for programmers so that they can focus on program semantics without having to consider networking aspects. The system automatically translates code into distributed method invocations, and executes methods at the most suitable network location, while taking into account available resources and the location of both input parameters and method state.

CFN can handle dynamic stateless and stateful method invocations. The system supports multi-tenancy, *i.e.,* it can run multiple concurrent programs from different users over the same infrastructure, dynamically assigning resources to individual computations. Isolation among tenants emerges naturally from the hierarchical ICN naming employed by RICE, which can bind cryptographic keying material[1] and resource allocation to names. However, we only consider well-formed distributed programs whose components can assume a single security boundary and are internally mutually trusted.

In CFN, compute nodes that can execute functions within a given program instance are called *workers*. The allocation of functions and actors to workers happens in a distributed fashion. A CFN system knows the current utilization of available resources and the least cost paths to copies of needed input data. It can dynamically decide which worker to use, performing optimizations such as instantiating functions close to big data inputs. The bindings that control which execution platforms host which program interfaces (or individual

[1]For both signature-based integrity and confidentiality, flexible tenant-specific trust schemas can be established by employing the methods in [30].

functions/actors) is maintained through a computation graph (see section 4.2).

To realize this distributed scheduling, workers in each resource pool advertise their available resources. This information is shared among all workers in the pool. A worker execution environment can decide, without a centralized scheduler, which set of workers to prefer to invoke a function or to instantiate an actor. In order to direct function invocation requests to specific worker nodes, CFN utilizes the underlying ICN network's forwarding capabilities – the network performs late binding through name-based forwarding and workers can provide forwarding hints to steer the flow of work.

2.3 Building Blocks

CFN employs ICN protocols, Remote Method Invocation in ICN (RICE), and Compute Graphs realized with Conflict-Free Replicated Data Types (CRDTs) as building blocks.

2.3.1 *Information-Centric Networking.*

CFN interacts with the (Information-Centric) network layer directly and depends on a number of its features. In CFN, programs consist of functions and actors named via ICN name prefixes. Individual functions are named with RICE-compatible (section 2.3.2) *referentially transparent* names under the corresponding prefix. Actors (à la Ray[18]) are similarly named with RICE-compatible names under the prefix.

ICN name-based routing is used to direct function invocations to workers on execution nodes. The names of workers and of programs are completely independent of each other. It is the task of the scheduler (section 4.4) to manage the relationship in order to distribute function execution among the workers. The bindings that control which workers execute which programs (or individual functions/actors) is maintained through the *computation graph* (see section 4.2).

The selection of the locus of execution when a function is invoked is determined by a combination of the information in the computation graph and native ICN routing, augmented with forwarding hints.

2.3.2 *RICE: Remote Method Invocation in ICN.*

ICN systems such as NDN[33] or CCNx[2] provide a more powerful forwarding service than TCP/IP because they employ a more elaborate per-hop behavior with per-packet soft-state in forwarders. ICN forwarders can respond in the data-plane to dynamic information to make better forwarding decisions, enable local retransmission, and allow in-network congestion control schemes. Therefore, ICN's fundamental service model of accessing named data through its INTEREST/DATA primitives provides an attractive basis for remote function invocation, as demonstrated by, for example, Named Function Networking[29] or NFaaS [11].

RICE[10] is a remote method invocation framework for ICN providing additional mechanisms for client authentication, authorization and ICN-idiomatic parameter passing. RICE implements a secure 4-way handshake in order to achieve shared secret derivation, consumer authentication and parameter passing.

Furthermore, RICE enables long-running computations through the concept of *thunks*[1] from the programming language literature. This decouples method invocation from the return of results – thunks name function results for retrieval purposes. In CFN, we use RICE machinery extensively for implementing all of the function invocation primitives and for result fetching.

2.3.3 *Conflict-Free Replicated Data Types.*

Conflict-free Replicated Data Types[26] (CRDTs) are data structures that can be replicated across multiple computers in a network while allowing independent, coordination-free update. CRDTs provide clean eventual consistency guarantees (*i.e.,* if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value). These properties make CRDTs valuable in applications such as collaborative document editing [13, 16] where updates come from multiple parties.

In CFN, we use CRDTs to implement the shared computation graph (section 4.2).

2.4 Terminology

We adopt the following terminology:

Program - a set of computations requested by a user. A program can be treated as a single sequential code thread calling multiple functions on multiple remote nodes.
Program Instance - one currently executing instance of a program
Function - a specific computation that can be invoked as part of a program. When assigned by the scheduler, a function of a particular program instance is instantiated and executed by one worker in the network.
Data - represents function outputs and inputs. Furthermore, actors make their internal state visible as a data object.
Future - futures are objects representing the results of a computation that may not yet be computed. In CFN a future is just the name of the data to be produced (section 3.2).
Worker - the execution locus of a function or actor of a program instance
CFN node - a node with the ability to host workers. Each CFN node can act as both worker and scheduler. Workers can join one or more program instances. Joining a program instance subscribes a worker to the computation graph associated with the program name prefix and allows other workers in this program instance to schedule functions on this worker.

3 PROGRAMMING MODEL AND NAMING

CFN implements a dynamic task graph computation model, *i.e.,* it represents an executing application as a graph of dependent tasks that evolves during execution. Within this model, CFN provides both an actor and a task-parallel programming abstraction.[2] In the following, we describe the programming model, our naming scheme and give an overview of the CFN program execution process.

3.1 Programming Model

Computations in CFN can be directly expressed in a variety of distributed programming languages. For our prototype, we use Python

[2]This unification differentiates CFN from related systems like CIEL[19], which only provides a task-parallel abstraction, and from Orleans [14] or Akka [1], which primarily provide an actor abstraction.

code. Developers can turn any regular Python program into a distributed computation with almost no change to the code. CFN hides the networking aspects from the programmer and automatically handles the details of name assignment and the execution of function calls. We designed a simple API to capture relations between data and function and re-use already computed results through the future mechanism (Table 1).

To minimize code changes we use function and class decorators. Developers designate functions as referentially transparent or referentially opaque through these decorations. CFN defines wrappers around decorated classes and functions to enable additional processing. On invocation, each decorated function immediately returns a future to the results instead of the actual data. The future can be then used either in a subsequent call to retrieve the result or passed as an input parameter to other functions. CFN will always return a future even if the result is already available on the worker. It allows us to keep clean semantics coherent with ICN pull-based model and presents potential performance benefits. The computed result might not be required on the caller node, but rather on another worker that receiving the future as an input argument, can directly fetch the required data. In order to retrieve the computed data, functions call *get(future)*; this will block until the requested data has been computed and fetched.[3]

In CFN, each class is a stateful actor comprising all its methods. CFN automatically recognizes state modifications and is able to migrate actors' state among different workers. The whole process is transparent for developers. Decorating an actor's functions does not require a different approach from that used for stateless functions. We present a sample CFN code below:

```
@cfn.actor
class CoughAnalyzer:
    #class state
    coughs = []
    alert = False

    @cfn.transparent
    def addSample(self, sample_f, features_f):
        sample, features = cfn.get(sample_f, features_f)
        coughs.append([sample, features])
        if diseaseDetected(coughs):
            alert = True

@cfn.opaque
def removeSpeech(sample_f):
    sample = cfn.get(sample_f)
    # remove speech from the sample
    return anonymized_sample

@cfn.transparent
def extractFeatures(sample_f):
    sample = cfn.get(sample_f)
    # analyze the sample
    return features
############ main ############
analyzer = CoughAnalyzer()
while True:
    sample_f = recordAudio()
    anonymized_sample_f = removeSpeech(sample_f)
    features_f = extractFeatures(anonimized_sample_f)
    analyzer.addSample(anonymized_sample_f, features_f)
```

The main function runs on a recording device (*i.e.,* a mobile phone), collects audio samples and invokes processing functions for speech removal (*removeSpeech()*) and feature extraction (*extractFeatures()*). Each function is decorated with *@cfn* decorator, returns a future referencing results and can be invoked on remote devices. Finally *CoughAnalyzer* class collects annotated samples and performs cross-sample analysis. CFN automatically manages state of the class, invokes functions on the most suitable nodes and performs state/data migration between workers. As a result, the constrained mobile phone can run a CPU-expensive application by using resources of nearby workers without heavy code modifications.

3.2 Naming

In CFN, each node automatically constructs the needed ICN names, subscribes to the corresponding program prefixes and executes code as RICE remote method invocations. Figure 1 summarizes our naming scheme.

Each worker capable of computing CFN programs is assigned an ICN-routable **CFN Node Name** prefix to which it appends its local worker name. This name is used to fetch data produced by the node or to schedule computations (section 4.4). Furthermore, we define a **Framework Prefix** for group communication among all CFN nodes under a given administrative entity (*e.g.,* /LA-CFN/). This is used to disseminate information about new programs and allows CFN nodes to join computations (section 4.1). Each CFN program has a unique **Program Name** representing its code base (*e.g.,* /EHealth/). When a new program instance is instantiated, it receives a unique instance identifier.[4] The **Program Instance Name** is formed by prepending the **Progam Name** with the Framework Prefix and appending an instance identifier. This is used for communication among nodes running the same program instance (*i.e.,* /LA-CFN/EHealth/45/). For example, the resource advertisement protocol (section 4.3) uses the framework instance specific **Resource Ads Prefix** (*i.e.,* /LA-CFN/resource/) and computation graph maintenance (section 4.2) uses the program instance specific **Computation Graph Prefix** (*i.e.,* /CFN/EHealth/45/graph/) .

Furthermore, we adopt the **Function Name** scheme used in RICE[10]. Referentially transparent function names consist of a program name, name of the function and a hash of the input parameters (*e.g.,* /EHealth/extractFeatures/(#)/). This allows us to use already computed results named by the future. In referentially opaque function names, the last component is replaced by a unique identifier generated by the caller (*e.g.,* /EHealth/removeSpeech/123/), which is needed to prevent incorrect result sharing.

We extend this model to support stateful actors in the form of classes. Each **Class Method Name** consists of program name, class name, and function name followed by an input hash or unique identifier (*e.g.,* /EHealth/CoughAnalyzer/addSample/(#)/). Finally, actors' state uses class name, 'state' keyword and a state object hash (i.e., /EHealth/CoughAnalyzer/state/#/). Such an approach allows for efficient state storage if the state was not modified by function invocations.

Data produced by functions share the name of the function invocation. Furthermore, function names do not include the program instance-specific prefix; this allows sharing already computed results among multiple program instances.

[3]Since functions of a program instance can be executed in parallel, futures also serve as the synchronization primitive that prevents race conditions.

[4]If the environment is capable of fetching and installing code dynamically, workers can use the program name to fetch the components required to run the program.

Table 1: CFN API.

Name	Description
@CFN.transparent / @CFN.opaque	Decorator for a referentially transparent or opaque stateless function. Each function invocation is automatically scheduled for remote invocation. Functions can take objects or futures as inputs and return one or more futures. This method return immediately.
object=CFN.get(futures)	Return the values associated with one or more futures. This method will return when the result is computed by a remote function or otehrwise block.
@CFN.actor	Class decorator for stateful actors. Instantiate the decorated class as a remote actor, and return a handle to it. Call a method on the remote actor and return one or more futures. Both are non-blocking

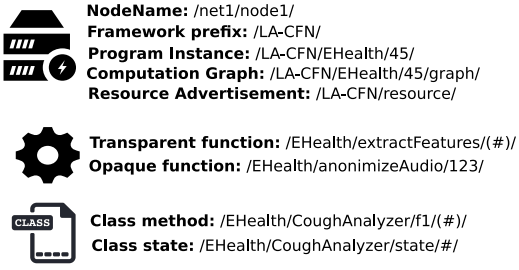

Figure 1: CFN naming scheme.

3.3 Program execution overview

Given this programming model and our naming scheme, we proceed with an overview of our system behaviour using the example shown in Figure 2. Our sample program[5] (and its main function) gets instantiated on NodeA that proceeds to create a program instance prefix and invoke the main function locally. This initial worker implicitly specifies which workers are eligible to participate in the program instance by deciding in which resource pool to execute the program instance (section 4.1). Nodes running the same program instance maintain its distributed computation graph (section 4.2).

When the code reaches a remote function invocation *removeSpeech()*, the call is sent to the local task scheduler (step 1). The scheduler immediately returns a future representing the result to be produced (step 2) and the caller code can continue its execution. In the meantime, the scheduler (section 4.4) takes into account available resources (*e.g.,* CPU, memory) on other workers in the program instance's resource pool and assigns *removeSpeech()* to NodeB by putting a forwarding hint in the computation request (step 3). When receiving the computation request, NodeB updates the computation graph which gets synced with the other workers. The whole process repeats when *extractFeatures()* is called (step 4-5). However, this time, the scheduler also takes into account the placement of the function input parameters (retrieved from the computation graph) when deciding to send the request to NodeC (step 6). NodeC starts executing *extractFeatures()* immediately and fetches all the required input parameters in the background (step 8) by sending an Interest towards NodeB (step 9). The computation blocks only when the function requests its input parameters by calling *get* and they have not yet been fetched. Once the input is fetched, NodeC can resume executing *extractFeatures()* and returns the final result.

4 SYSTEM COMPONENTS

We provide a detailed description of the CFN components.

[5]We omit the program prefix for better readability.

4.1 Program Membership Management

Each program starts on a worker that receives the initial request and is responsible for running the *main* function. The initial worker creates and advertises a program instance prefix and a prefix for maintaining the computation graph (section 4.2). If the worker decides not to handle the program computation by itself, it can schedule functions on additional workers that currently are not part of the program instance, but advertise available resources (section 4.3). Upon receipt of a scheduled request, a CFN node can decide to join the computation by subscribing to the program instance prefix.

A worker thereby receives computation graph updates, can accept computations assigned by other nodes and can request execution of program functions on other nodes executing the same program instance (section 4.4). Each worker can be executing portions of as many programs as its resources permit. If the current number of workers is insufficient to meet the performance metrics of the program, any CFN node participating in the program instance's execution can try to involve additional nodes using the same mechanism. Overall resource management is the responsibility of the task scheduler (section 4.4) which can perform scale up or scale down to respond to changes in demand for a given program, tenant, or globally for the CFN framework instance as a whole.

4.2 Computation Graph

In CFN, workers maintain the computation graph for each program instance being executed. Each graph is maintained only by workers participating in the execution of a given program instance. The graph captures the information needed to track invoked functions and produced data (Figure 3). There are 3 types of graph nodes[6]:

Stateless Function - represents a stateless function invocation for either a referentially transparent or referentially opaque computation. A stateless function graph node contains a function name, a list of input names, a list of produced data and the name of the parent function that called it.

Actor Function - represents a stateful function invocation for either a referentially transparent or referentially opaque computation. As with functions, an actor graph node contains a function name, a list of input names, a list of produced data and the name of the parent function it was called by. Actor graph nodes also include names of objects representing an input state (before the method was invoked) and an output state (after the method was invoked).

Data - represents either data produced by function or state objects capturing the persistent state of the function. A data graph node

[6]For simplicity and to limit the number of independent objects needed to maintain the graph, graph arcs are represented by the parent function element of each graph node.

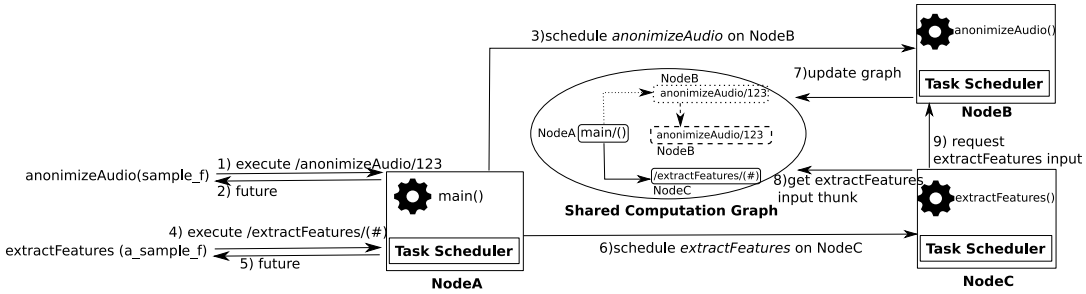

Figure 2: CFN Overview.

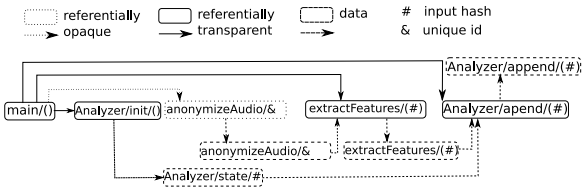

Figure 3: Computation graph created from sample Python code.

contains a data name, a name of the function it was produced by, and a *size* representing the resources required to store the corresponding data object. The size is an estimate while the data is still being produced and is updated to the actual value when the producing function finishes. Data nodes may represent Actor state objects.

Furthermore, each graph node type contains a list of thunks. At first, the thunk list contains the name of a worker that invoked the function or produced data described in the graph node. However, the list can be extended if multiple workers invoke the same referentially transparent function (*e.g.,* due to network partitioning) or if data gets replicated.

Each graph node is uniquely identified by its name as described in Section 3.2. Graph nodes for referentially opaque computations receive a unique name component value from their callers, while multiple referentially transparent function instances sharing one name (and thus input parameters) can be merged into a single graph node as below. Data names are derived from the function that produced them and inherit their features.

When an actor function is invoked, it takes a state object as input (except actor creation function) and after execution creates a new state object.

The old state object is kept on the invoking worker.[7] Such an approach allows us to unify the handling of stateful and stateless computations, and by considering state objects to be input/output parameters, migrate any type of computation among worker nodes. This also simplifies the design of the scheduler described in section 4.4.

The computation graph must be kept up to date among the workers involved in executing a program instance. However, in contrast to a centralized scheduler, in a distributed environment updates can be lost, duplicated or reordered so that maintaining a shared, synchronized data structure is difficult. When a function is invoked or data is produced, the responsible worker creates a corresponding graph node. Each node keeps information about related objects in their input and /output/called_by field; this enables graph reconstruction entirely from updates and allows participating workers to explicitly request missing information.

Such an approach translates maintaining the graph into maintaining a set of updates. To represent the graph as a CRDT, we implement an efficient merge operation among graph nodes with the same name, but different content. Our naming scheme (section 3.2) confines merging to cases where the same referentially transparent function gets invoked by different workers or data is replicated among workers. The graph nodes can differ if data has been migrated among computation nodes as the computation evolves. In such cases, the merge simply updates the computation node list in the graph node.[8]

For simplicity and interoperability, our CFN prototype represents graph updates in JSON. Updates can be distributed through NDN synchronization protocols, such as Psync [34], which provide the necessary primitives to achieve the eventual consistency semantics of CRDTs. For our current prototype (section 5), we have implemented a simplified synchronization scheme through a multi-cast namespace among CFN nodes.

4.3 Resource Pool Management

Each administrative domain of a CFN network can be divided into several resource pools, partitioned by tenant. Each CFN node may be assigned to one or multiple resource pools of one or more tenants.[9] By subscribing to a resource pool, a worker can accept computations assigned by other nodes and can request assignment of program functions to other nodes executing the same program (section 4.4). This reduces the amount of information handled by each worker and exploits the available parallelism to reduce completion times. We investigate the impact of resource pool size in section 5.

For resource advertisements, we implement a scoped flooding protocol. An instance the resource advertisement protocol runs among members of each resource pool. This straightforward ICN implementation approach provides reasonably low convergence delay but no optimality guarantees for resource placement.

Note that it is up to the local resource manager on each node to decide whether resources get shared or partitioned among their local resource pools, and whether to do optimistic or pessimistic

[7] Older states may be used by a debugger - this can be very useful when debugging a distributed program as it captures an important part of the state evolution of the program.

[8] In CFN, once downloaded and advertised, data cannot be removed until the program ends. Supporting deletion would require a more complex computation graph representation and complicate the CRDT design.

[9] Tenants can be isolated from each other and their resource pools structured hierarchically by exploiting native ICN naming capabilities.

advertisements. If they are shared among tenants, again it is a local decision on how to manage the isolation to minimize/eliminate inter-tenant competition for resources.

Every time interval t each worker in a pool advertises its resources in terms of available CPU and the number of tasks in its task queue. This information is then used by the task scheduler when assigning functions to specific workers (Section 4.4).

4.4 Task scheduler

A major component of our CFN system is a task scheduler. In any system with a large number of tasks, the scheduler must be extremely efficient and have a sufficiently global and up-to-date view of the system state to perform load distribution considering both performance and failure resilience. Centralized schedulers can become a bottleneck of the whole system[18]. Therefore, we instead chose to develop a decentralized scheduler that shares the load of scheduling tasks among all the workers participating in each program instance. Our scheduler design achieves the following desirable characteristics:

- Functions are invoked close to the data they rely on. We apply the same criteria for stateless functions and Actors by treating the encapsulated state of an Actor the same as an input parameter. The scheduler's optimization function tries to keep stateful computation local to one physical node.
- Each graph node has an assigned thunk name by which data or functions can be accessed/invoked. If the thunk is accessed before the data is available, the requestor is informed via an error return to try again later.
- By keeping track of thunks, the scheduler can ascertain if someone makes a call to already existing stateful instance, and hence know where to inform the caller to send it.

Function requests originate on the worker hosting the calling (*i.e.,* parent) function. This worker uses information from the computation graph and the available resources in the resource pool executing the program instance to choose the most suitable worker to run each function.[10] Once the choice has been made, the scheduling worker invokes the function through RICE, using the function name as the Interest Name. The chosen worker is selected by placing its worker name in a forwarding hint for the RICE Interest, which allows any ICN node (not just CFN nodes) to forward the interest toward the correct worker. Once the scheduled worker starts function execution, it puts the thunk for its result into the computation graph so callers can fetch it or pass it to other functions.

When processing a request, the scheduler follows the following algorithm:

(1) Extract the names of function inputs from the requests.
(2) Query the computation graph for the size of each input.
(3) Retrieve any missing parts of the graph if necessary.
(4) Select the input of the largest size.
(5) Query the computation graph for the thunk of the closest node[11] having the largest portion of input data.
(6) Check with the resource advertisement protocol if the selected node is not overloaded (this is only possible when being within the range of the target node).
(7) If the target node has enough resources, send out the Interest to invoke the function.
(8) If the target node is overwhelmed, send the packet towards the closest node with enough resources within the same program instance.
(9) If all neighbour nodes involved in the program instance are overwhelmed, schedule the function on any node within the same resource pool with available resources.

This algorithm is a good match for the underlying RICE and ICN protocol substrate. However, we do not make any claims concerning its optimality as a scheduler algorithm *per se*. Its dynamic performance depends on a number of factors, including the overhead versus responsiveness tradeoffs of the resource advertisement protocol and the convergence delay of the CRDTs representing the computation graph.

4.5 Exceptions and Failure Recovery

Any distributed computing framework has to deal with failures, for example, unavailable nodes due to crashes or network partitions. Because CFN is leveraging ICN it can directly benefit from ICN's fault tolerance and problem mitigation mechanisms for network failures (link failures, congested paths etc.) such as location-independence (enabling late-binding of Interest names to producers through forwarding), ICN routing, and local repair through caching.

In addition, CFN can tolerate node failure and corresponding loss of computation results. There are two alternative strategies (we compare their efficiency in Section 5):

(1) Upon failure detection, CFN can *proactively* recompute everything that has been lost (i.e., CFN can directly invoke the corresponding function again.).
(2) CFN can also defer re-computation until another function actually needs the results (*reactive* approach).

5 EVALUATION

We have implemented a prototype of CFN[12] in ndnSIM [15]. We evaluated this prototype in a variety of network topologies, including line, tree, and mesh topologies, as well as a rocketfuel topology [27] of 175 nodes. For the line, tree, and mesh topologies, we experiment with a variable number of CFN nodes. For the rocketfuel topology, we randomly select 50 CFN nodes to evaluate our prototype in non-deterministic conditions, where CFN nodes are connected through different number of hops and network delays.

While several cloud-based datasets are available for public use, they usually do not contain any information about input/output data sizes or their initial placement. For our experiments, we therefore use a set of synthetic task traces generated in Python containing 1000 functions. Each program is initiated in the main function spawning from 1 to 8 processes. Each new process randomly calls from 0 to 8 new functions until reaching 1000 functions. We consider traces containing 1000 functions large enough for an initial investigation of the tradeoffs of our CFN prototype, while we plan to evaluate CFN in the context of larger traces in the future. Each function in a trace has a variable number of inputs, sizes of produced data and input/output dependencies, as we aim to understand

[10] A worker assigning a function can of course choose itself.
[11] Based on the ICN forwarder's metrics.
[12] https://github.com/spirosmastorakis/CFN

the specifics of the CFN design while varying these parameters. For each experiment, we did ten simulation runs and we report on the 90th percentile of the results.

5.1 Small topologies

We start our evaluation by investigating the scalability of our approach for three different topology types (line, mesh and tree). We submit a 1000-function program with no input parameters, and compare the completion time against an optimal case, without communication delay between nodes (Figure 4). CFN imposes only minimum performance penalty resulting from the delay of the links between the scheduling nodes and the assigned workers. Completion time is reduced with every added worker.

We continue by evaluating the mechanisms used for scheduling tasks. Figure 5 presents the completion time of a 1000-function program while taking into account data placement and resources (locality) and when relying uniquely on available resources (no-locality). When a topology is small, the data locality mechanism does not play an important role. However, with increasing distances between workers, pulling large input parameters may heavily delay the program completion time. This effect is more visible in less connected line and tree topologies, causing 88% and 48% performance decrease respectively with 30 nodes.

Furthermore, we evaluate the same mechanisms with different average size of data parameters (Figure 6). As expected, increasing the size of input parameters increases the completion time for all the cases, since functions may wait longer to fetch the required data. However, sending requests toward the large input parameters allows us to decrease the running time by 50% in the line topology and by 40% in the tree topology of 30 nodes. Both tests below show how important it is to take into account data placement in an edge environment when moving data causes significant delays.

We repeat the test by keeping a fixed average size of input parameters, but increasing their number (Figure 7). Surprisingly, more inputs does not necessarily increase the program completion time. Functions can fetch input parameters in parallel, and the total download time is determined by the furthest and largest of them. Increasing the average number of input parameters increases the chance for downloading large objects located far away. This effect is clearly visible in the tree and line topologies, while it has little impact on well-connected nodes in the mesh topology. The results suggest that placing functions at the center of gravity of all its inputs can significantly decrease the completion time.

Figure 8 presents the impact of node failures[13] on the program completion time. We randomly stop the specified percentage of nodes between 50s and 100s of each simulation. We investigate the proactive and reactive recovery strategies presented in Section 4.5. For all the investigated topologies, the proactive strategy results in increased completion time. As observed previously, the line topology results in the highest performance decrease due to longer paths between nodes.

We then follow by investigating the total number of tasks computed in each scenario (Figure 9). As expected, the reactive strategy reduces the total load by recomputing only tasks that are necessary to complete the program.

[13]We assume that the CFN instance of a topology node has failed, however, the ICN forwarder installed on this topology node is still functional, maintaining the connectivity of the overall topology.

5.2 Rocketfuel topology

We test the scalability of our framework by running a 1000-function program choosing different nodes as initial workers in the AS6461 rocketfuel topology (Figure 10). We observe minor (<7%) differences between central and peripheral nodes. The CFN distributed design schedules functions using multiple workers allowing it to work efficiently regardless of the initial worker's location.

Figure 11 presents the impact of data size and the number of input parameters on program completion time. The tests confirm our previous results for smaller topologies (Figure 7). The completion time is mainly influenced by the size and location of the input parameters (especially of the largest and furthest ones) rather than their number.

Finally, we reinvestigate the impact of node failures and performance of our two recovery strategies (Section 4.5). In the previous setup (Figure 8) we concluded that the reactive strategy performed better for all the topologies. However, this time we run 3 different programs with increasing number of input/output dependencies between functions. For low number of dependencies (program1), the reactive strategy still performs better, but for high number of dependencies (program3) the proactive strategy results in a decreased completion time. Analyzing and predicting the degree of interaction among functions could help to choose the optimal recovery strategy and decrease the overall completion time.

5.3 Discussion

Our evaluation suggests that CFN is able to efficiently distribute sequential programs among multiple workers, significantly speeding up its execution. Utilizing ICN protocols for these tasks brings several important benefits. Our framework automatically assigns names to program files (*i.e.,* source code/binaries) as well as function results. We can then easily migrate functions and data among nodes without using costly, DNS-like discovery (objects can be obtained from the closest node in the network).

CFN uses RICE as its external tool. It allows to map all interactions to one RMI interaction type providing a simple, yet powerful interface. RICE was evaluated in its respective paper [10] and a more detailed evaluation is out of the scope of this work.

As currently designed, CFN requires all worker nodes to process all computation graph updates for the program instances they run (which arrive as JSON messages). This overhead increases linearly with the number of functions/data produced. Dividing a program into smaller functions enable fine-grained load balancing, but incurs increased overhead.

Furthermore, when routing requests, computation nodes choose a worker with the largest input parameters. In CFN, since all the dependencies are explicitly listed in the request, they can be efficiently extracted from the computation graph (*i.e.,* using a hash map with $O(1)$ access time), and comparing sizes of input parameters involves only simple arithmetic operations. Finally, each node collects information about resources available on neighbouring nodes. However, usage of scope flooding guarantees fixed overhead of this process regardless of the total number of nodes in the resource pool.

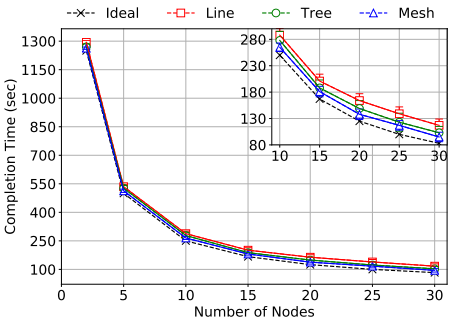

Figure 4: CFN scalability.

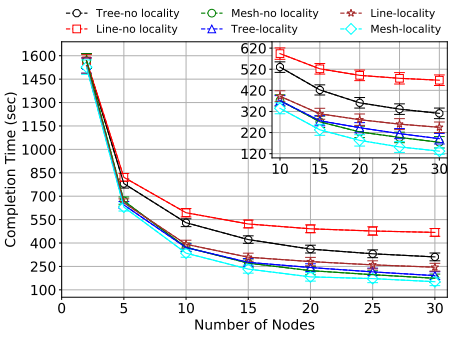

Figure 5: Impact of data locality mechanisms for different number of workers.

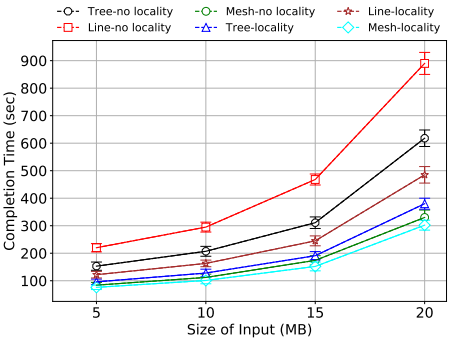

Figure 6: Impact of data locality mechanisms for different data sizes.

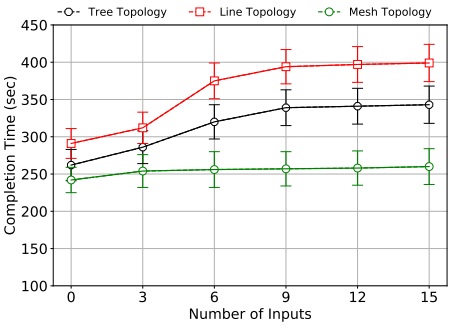

Figure 7: Impact of the number of function inputs on the completion time.

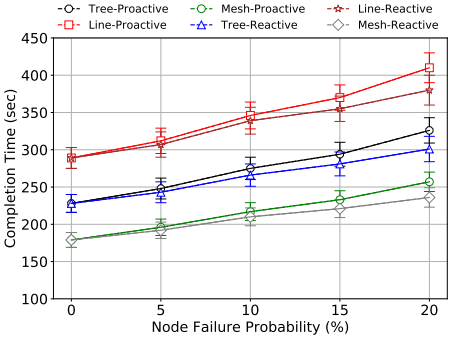

Figure 8: Impact of node failures on the completion time.

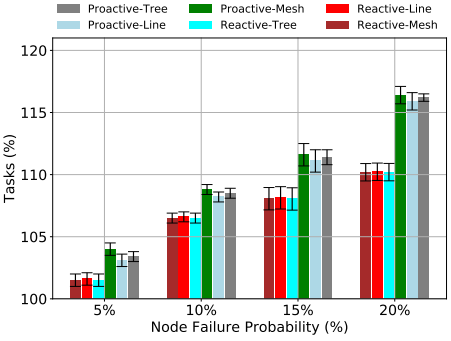

Figure 9: Impact of node failures on the number of executed tasks.

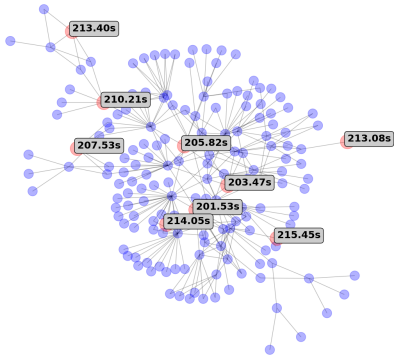

Figure 10: Impact of initial worker placement.

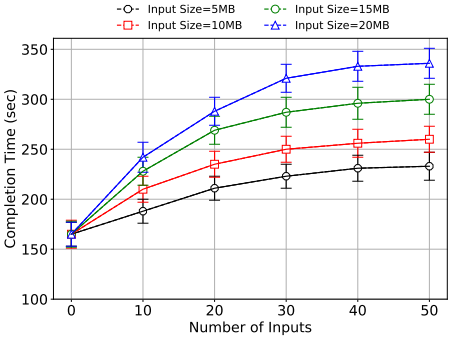

Figure 11: Impact of input number and size on performance.

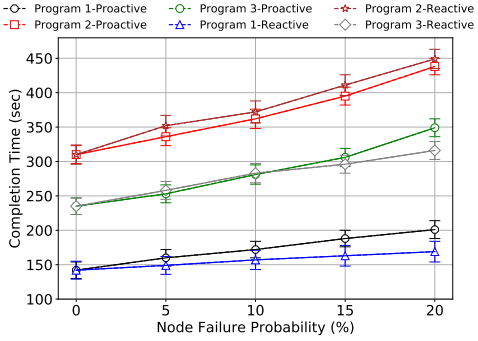

Figure 12: Impact of node failure on the overall performance.

Utilizing ICN protocols can significantly reduce the bandwidth required to propagate computation graph updates. This overhead could be further decreased using more space-efficient techniques such as Protocol Buffers[14]. Additionally, we observe that not all the workers in the resource pool need to be notified about every update. However, those patterns are difficult to formalize due to the dynamic and unpredictable nature of each program. It may be feasible to apply program-specific machine learning techniques to further divide resource pools into more specialized clusters and drastically reduce the overhead of our approach.

[14]https://developers.google.com/protocol-buffers

6 RELATED WORK

In the following, we explain how CFN relates to prior and related work on in-network and edge computing, distributed computing frameworks, and ICN.

In-network and edge computing In-network computing has mainly been perceived in four variants so far: 1) Active Networking [28], adapting the per-hop-behavior of network elements with respect to packets in flows, 2) Edge Computing as an extension of virtual-machine (VM) based platform-as-a-service, 3) programming the data plane of SDN switches (through powerful programmable CPUs and programming abstractions, such as P4 [25]), and 4) application-layer data processing frameworks.

Active Networking has not found much deployment due to its problematic security properties and complexity. Programmable data planes can be used in data centers with uniform infrastructure, good control over the infrastructure, and the feasibility of centralized control over function placement and scheduling. Due to the still limited, packet-based programmability model, most applications today are point solutions that can demonstrate benefits for particular optimizations, however often without addressing transport protocol services or data security that would be required for most applications running in shared infrastructure today.

Edge Computing (as traditional cloud computing) has a fairly coarse-grained (VM-based) computation-model and is hence typically deploying centralized positioning/scheduling though virtual infrastructure management (VIM) systems.

Distributed computation frameworks Application-layer data processing such as Apache Flink [8] provide attractive dataflow programming models for event-based stream processing and lightweight fault-tolerance mechanisms – however systems such as Flink are not designed for dynamic scheduling of compute functions.

Modern distributed applications frameworks such as Ray [18], Sparrow [22] or Canary [23] are more flexible in this regard – but since they are conceived as application-layer frameworks, their scheduling logic can only operate with coarse-grained cost information. For example, application-layer frameworks in general can only infer network performance, anomalies, and optimization potential indirectly through observed performance or failure, so most scheduling decisions are based on aggregate metrics such as platform load.

Centralized schedulers such as Dryad [7], Ciel [19] or Spark [31] can implement a optimal task placement taking into account data location. However, the centralized design incurs increased scheduling latency and exhibits scalability problems when the number of tasks increases beyond the capacity of a single CPU to schedule.

In data centers where resource pools are rich and failure domains large, a number of distributed shared memory approaches to distributed computing have been notably successful. These include, among others, Memcached [5] and RAMCloud [21]. Such systems however do not generalize to heterogeneous or edge computing environments, which makes them not a direct alternative to CFN.

Information-Centric Networking We explained how CFN leverages ICN and RICE in section 2. Named Function Networking [29] is the seminal proposal for combining ICN's access to named data with dynamic computation invocation over ICN. Whereas in NFN, the compute-graph is known in advance (as a nested set of expressions that get evaluated, leading to invocations of function instances in an NFN network), CFN is more dynamic, as the program execution can lead to arbitrary instantiation of new actors that can in turn invoke additional functions, instantiate new actors etc. CFN can thus be characterized as a distributed application platform, whereas NFN is intended for deterministically evaluating function expressions.

NFaaS [11] allows efficient, opportunistic task scheduling, but takes into account only available resources (i.e., CPU) ignoring data placement. Furthermore, the framework focuses on stateless computations and does not provide explicit support for stateful actors.

7 CONCLUSION

In-network computing is an important and consequently popular research topic with a fairly wide design space. Assessing the state-of-the-art with a systems perspective has led us to the design of CFN– a distributed computing framework with a fresh approach to decentralized resource allocation, employing the concept of *joint optimization of computing and network resources*.

We have demonstrated that there is a sweet spot in the design space in combining 1) feature-complete distributed programming with support for stateless and stateful computations (as provided by Ray [18] and other frameworks) with 2) a rigorous computation graph approach for representing distributed computations, directly supported and tightly integrated with a suitable ICN network layer, and 3) a distributed, replicated representation of the computation graph using CRDTs, also directly supported by the ICN network layer.

This allows CFN to overcome the inefficiencies of state-of-the-art overlay-based approaches and to provide better availability and reaction to failures as we demonstrated in our evaluation. These features are enabled by leveraging the ICN network layer and the RICE framework for performing functions such as request forwarding, caching, and load management without the need for additional overlay mechanisms. We believe that the potential of systems like CFN is huge. They could enable the development of general-purpose distributed computing platforms that are applicable to a wide range of scenarios, from (mobile) edge computing, to distributed computing and network programmability in data centers.

This paper demonstrates the qualitative benefits (reduction of complexity without losing functionality). We have discovered additional opportunities to enhance CFN with respect to performance and resource utilization under different conditions: Our future plans include extending CFN's integration with the network layer by delegating more of the binding of functions to execution loci to the network. Currently CFN employs an explicit binding created by an offloading worker and explicit source routing through ICN forwarding hints. We envision employing the ICN routing system by enabling workers to advertize functions they can execute as name-prefix routes. The routing metrics would be adjusted by the CFN resource management that takes load information (from the named execution platforms), decides how to instantiate the graph, and tells the platforms what metric they should use in their advertisements. Furthermore, we plan to improve function placement in regard the their input parameters and automatic selection of failure recovery strategy. Finally, we plan to further investigate real-world applications, deploy a pilot system and perform an extensive comparison with TCP/IP based application.

8 ACKNOWLEDGMENTS

The authors are grateful to the ACM ICN'19 anonymous reviewers and our shepherd John Wroclawski for their constructive comments and suggestions. This work was supported by the Fonds de la Recherche Scientifique - FNRS under Grant #F452819F, EC H2020 ICN2020 project under grant agreement number 723014, EPSRC INSP Early Career Fellowship under grant agreement number EP/M003787/1 and H2020 DECODE project under grant agreement number 553066.

REFERENCES

[1] [n. d.]. Thunk. https://en.wikipedia.org/wiki/Thunk.

[2] 2018. Project CCNx. http://www.ccnx.org/.

[3] Samantha J Barry, Adrie D Dane, Alyn H Morice, and Anthony D Walmsley. 2006. The automatic recognition and counting of cough. *Cough* 2, 1 (2006), 8.

[4] GHR Botha, G Theron, RM Warren, M Klopper, K Dheda, PD Van Helden, and TR Niesler. 2018. Detection of tuberculosis by automatic cough sound analysis. *Physiological measurement* 39, 4 (2018), 045005.

[5] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5–. http://dl.acm.org/citation.cfm?id=1012889.1012894

[6] Dennis Grewe, Marco Wagner, Mayutan Arumaithurai, Ioannis Psaras, and Dirk Kutscher. 2017. Information-Centric Mobile Edge Computing for Connected Vehicle Environments: Challenges and Research Directions. In *Proceedings of the Workshop on Mobile Edge Communications (MECOMM '17)*. ACM, New York, NY, USA, 7–12. https://doi.org/10.1145/3098208.3098210

[7] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 59–72.

[8] A. Katsifodimos and S. Schelter. 2016. Apache Flink: Stream Analytics at Scale. In *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*. 193–193. https://doi.org/10.1109/IC2EW.2016.56

[9] J Korpáš, J Sadloňová, and M Vrabec. 1996. Analysis of the cough sound: an overview. *Pulmonary pharmacology* 9, 5-6 (1996), 261–268.

[10] Michał Król, Karim Habak, David Oran, Dirk Kutscher, and Ioannis Psaras. 2018. RICE: Remote Method Invocation in ICN. In *Proceedings of the 5th ACM Conference on Information-Centric Networking*. ACM.

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

[12] Eric C Larson, TienJui Lee, Sean Liu, Margaret Rosenfeld, and Shwetak N Patel. 2011. Accurate and privacy preserving cough sensing using a low-cost microphone. In *Proceedings of the 13th international conference on Ubiquitous computing*. ACM, 375–384.

[13] Xiao Lv, Fazhi He, Weiwei Cai, and Yuan Cheng. 2017. A string-wise CRDT algorithm for smart and large-scale collaborative editing systems. *Advanced Engineering Informatics* 33 (2017), 397–409.

[14] Spyridon Mastorakis, Alexander Afanasyev, Ilya Moiseenko, and Lixia Zhang. 2015. ndnSIM 2.0: A new version of the NDN simulator for NS-3. *NDN, Technical Report NDN-0028* (2015).

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

[16] Ahmed-Nacer Mehdi, Pascal Urso, Valter Balegas, and Nuno Perguiça. 2014. Merging OT and CRDT algorithms. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*. ACM, 9.

[17] Chris Meiklejohn. 2016. Lasp. *Applicative 2016 on - Applicative 2016* (2016). https://doi.org/10.1145/2959689.2960077

[18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 561–577. http://dl.acm.org/citation.cfm?id=3291168.3291210

[19] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 113–126.

[20] World Health Organization et al. 2017. The top 10 causes of death. January 2017.

[21] John Ousterhout, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, Ryan Stutsman, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, and et al. 2011. The case for RAMCloud. *Commun. ACM* 54, 7 (Jul 2011), 121. https://doi.org/10.1145/1965724.1965751

[22] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 69–84.

[23] Hang Qu, Omid Mashayekhi, David Terei, and Philip Levis. 2016. Canary: A scheduling architecture for high performance cloud computing. *arXiv preprint arXiv:1602.01412* (2016).

[24] Elliot Saba. 2018. *Techniques for Cough Sound Analysis*. Ph.D. Dissertation.

[25] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*. ACM, New York, NY, USA, 150–156. https://doi.org/10.1145/3152434.3152461

[26] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[27] Neil Spring, Ratul Mahajan, and David Wetherall. 2002. Measuring ISP topologies with Rocketfuel. *ACM SIGCOMM Computer Communication Review* 32, 4 (2002), 133–145.

[28] David L. Tennenhouse and David J. Wetherall. 1996. Towards an Active Network Architecture. *SIGCOMM Comput. Commun. Rev.* 26, 2 (April 1996), 5–17. https://doi.org/10.1145/231699.231701

[29] Christian Tschudin and Manolis Sifalakis. 2014. Named functions and cached computations. In *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*. IEEE, 851–857.

[30] Yingdi Yu, Alexander Afanasyev, David Clark, kc claffy, Van Jacobson, and Lixia Zhang. 2015. Schematizing Trust in Named Data Networking. *Proceedings of the 2nd International Conference on Information-Centric Networking - ICN '15* (2015). https://doi.org/10.1145/2810156.2810170

[31] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[32] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. 2014. Customizable and Extensible Deployment for Mobile/Cloud Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 97–112. http://dl.acm.org/citation.cfm?id=2685048.2685057

[33] Lixia Zhang, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D Thornton, Diana K Smetters, Beichuan Zhang, Gene Tsudik, Dan Massey, Christos Papadopoulos, et al. 2010. Named data networking (ndn) project. *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC* (2010).

[34] Minsheng Zhang, Vince Lehman, and Lan Wang. 2017. Scalable Name-based Data Synchronization for Named Data Networking. In *IEEE Infocom (Infocom)*. IEEE, IEEE Computer Society.