NFaaS: Named Function as a Service

Michał Król
University College London
m.krol@ucl.ac.uk

Ioannis Psaras
University College London
i.psaras@ucl.ac.uk

ABSTRACT

In the past, the Information-centric networking (ICN) community has focused on issues mainly pertaining to traditional content delivery (*e.g.,* routing and forwarding scalability, congestion control and in-network caching). However, to keep up with future Internet architectural trends the wider area of future Internet paradigms, there is a pressing need to support edge/fog computing environments, where cloud functionality is available more proximate to where the data is generated and needs processing.

With this goal in mind, we propose *Named Function as a Service* (NFaaS), a framework that extends the Named Data Networking architecture to support in-network function execution. In contrast to existing works, NFaaSbuilds on very lightweight VMs and allows for dynamic execution of custom code. Functions can be downloaded and run by any node in the network. Functions can move between nodes according to user demand, making resolution of moving functions a first-class challenge. NFaaSincludes a *Kernel Store* component, which is responsible not only for storing functions, but also for making decisions on which functions to run locally. NFaaSincludes a routing protocol and a number of forwarding strategies to deploy and dynamically migrate functions within the network. We validate our design through extensive simulations, which show that delay-sensitive functions are deployed closer to the edge, while less delay-sensitive ones closer to the core.

CCS CONCEPTS

• **Networks** → **Network architectures**; *Network management*; Network simulations;

KEYWORDS

Networks, Network architectures, Information Centric Networking, Mobile Edge Computing, Function Migration

ACM Reference format:
Michał Król and Ioannis Psaras. 2017. NFaaS: Named Function as a Service. In *Proceedings of ICN '17, Berlin, Germany, September 26–28, 2017,* 11 pages.
https://doi.org/10.1145/3125719.3125727

1 INTRODUCTION

While the current Internet handles content distribution relatively well, new computing and communication requirements call for new functionality to be incorporated. Powerful end-user devices and new applications (*e.g.,* augmented reality [1]) demand minimum service delay, while the Internet of Things (IoT) [2] generates huge amounts of data that flow in the reverse direction from traditional flows (that is, from the edge towards the core for processing). As a result, computation needs to be brought closer to the edge to support minimum service latencies and to process huge volumes of IoT data.

In contrast to cloud computing, edge and fog computing promote the usage of resources located closer to the network edge to be used by multiple different applications, effectively reducing the transmission delay and the amount of traffic flowing towards the network core. While offering multiple advantages, mobile edge computing comes with many challenges, *e.g.,* dynamic placement of applications at edge nodes and resolution of requests to those nodes [3] [4]. Furthermore, the application software needed to run edge computation must be first downloaded on the edge-node, while mobile clients request for resources from different locations. Mobility as well as diversity in user demand makes it very difficult to predict which functions will be requested in the future and from where in the network.

Indeed, ICN principles can directly address some of the above challenges. Explicitly named functions can be resolved in network nodes, while network-layer requests (*i.e.,* Interests) can carry input information for edge-executable functions. Function code can be stored in node caches and migrate across the network following user demand.

With the exception of a few relatively early works [5] [6] [7], there has been no focused attempt to adjust existing proposals for Information-Centric Networks to support edge computing. Early works in the area have enhanced the ICN stack to support service discovery, *e.g.,* [5][6] [8]. However, these works do not support dynamic service instantiation or system adaptation to user demand. Functions are static and are executed at one node (each), while their resolution relies on central controllers ([6]) failing to exploit the stack's full potential. As a result, edge-nodes can get overloaded in case of increased demand for some function, while the system cannot adapt to user mobility by migrating functions to other parts of the network. We argue that any service should be able to run at any node in the network, migrate to the most optimal node based on the user's location, replicate in case of increased demand and dissolve when demand for some function declines.

Closer to our work *Named Function Networking* (NFN) builds on λ-functions [7] to allow services to be executed anywhere in the network. Being restricted to basic λ-functions included in the Interest name, NFN is constrained by the number of services it can support. In many scenarios, nodes require more sophisticated processing, custom code and libraries, which is difficult to express only through λ-functions, and acquiring additional function code presents new challenges.

ICN '17, September 26–28, 2017, Berlin, Germany

ACM ISBN 978-1-4503-5122-5/17/09…$15.00
https://doi.org/10.1145/3125719.3125727

In this work we assume very lightweight Virtual Machines (VMs) in the form of *unikernels* [9], which are explicitly named. We assume network nodes have: *i)* storage capacity to cache *unikernels* (*i.e.,* function code) and *ii)* computation capacity to execute those functions.[1] Moreover, edge nodes can store more unikernels than they can run simultaneously.

We introduce the concept of *Named Function as a Service* (NFaaS), whereby rich clients, containing most of the application logic, request for named functions through Interests [10]. The structure of NDN Interests is modified to request for function execution (as opposed to merely requesting for static content only) and include information that acts as input for the requested function. In addition to the Content Store, NFaaS nodes also have the *Kernel Store*, which is responsible not only for storing function code, but also for making decisions as to which functions to execute. Depending on user demand, functions move between network nodes and the routing fabric is updated to resolve moving functions.

According to the resulting framework, network nodes (or clusters of them) in some specific part of the network domain specialise in executing specific functions. Although these nodes then become the reference point for those functions, this does not prevent functions from executing elsewhere in the network too. Nodes send Interest packets with the name of the function to execute. Any node in the network can run the service and return results. Decisions on which function to execute are based on a *unikernel score* function, whose purpose is to identify and download the most popular functions. We base our system on the *serverless architecture* [11] [12], thus, removing state information from the hosting node. This allows us to migrate services much easier without any costly handover procedure and adapt to the current demand.

We implement NFaaSas an extension to the NDN stack [13]. Our system remains fully compliant with the original NDN implementation and does not require all nodes to support the extension. For simplicity, we assume two main types of functions: *i)* functions for delay-sensitive applications, and *ii)* functions for bandwidth-hungry applications. Our results show that the system adapts according to our expectation: delay-sensitive functions migrate and execute mostly towards the edge of the network, while bandwidth-hungry functions execute further in towards the core of the network, but always staying within the boundaries of the edge-domain. To the best of our knowledge, NFaaSis the first framework enabling this functionality without a global view of the network.

The remainder of the paper is organized as follows. Section 2 provides background information on unikernels and serverless architecture. The design details of NFaaS along with a summary of the main research challenges are presented in Section 3. Section 4 presents initial results evaluating the design with simulations and a real-world prototype. In Section 5, we summarize previous work on service invocation in information-centric networks.

[1]We use terms *unikernels*, functions and services interchangeably to refer to edge-executable applications.

2 BACKGROUND

2.1 Unikernels

Recent advances in virtualisation techniques have allowed rapid deployment of cloud services. However, if we want to achieve system isolation, it is required to set up a whole virtual machine for every hosted service. A virtual machine image contains a complete system with a full kernel, a set of drivers, system libraries and applications. Most of these components remain unused by the invoked service. This approach makes the image big in size, slow to boot and increases the surface of attacks. Container-based virtualisation alleviates the problem, but solutions such as Docker, while making the deployment process easy, require running on top of a complete operating system and contain a significant amount of additional components.

Unikernels [9] propose an alternative approach, whereby an application is analysed, to determine the required system components. Only this part is compiled, together with the application binary into a bootable, immutable image. *Unikernels* have several advantages over the classic solutions presented above. They are small in size, introduce minimal attack surface and can operate on bare metal or directly on a hypervisor. Their compilation model enables whole-system optimisation across device drivers and application logic. Multiple systems already exist that are able to turn any custom code into unikernels [14][15][16]. Thanks to the small size (few MB) and low overhead, unikernels can be downloaded and executed in milliseconds. While still at an early stage of development, unikernels present huge deployment potential as unikernel-based systems show very good performance in comparison with virtual machines and containers [17], can be easily cached and run on almost any device. These characteristics make unikernels a great solution for edge/fog computing environments where functions migrate within the network. The technology is already used in projects such as Jitsu [18], where after receiving a DNS packet, a server instantiates a unikernel that processes the request and sends back the response introducing delays of only few milliseconds.

2.2 Serverless architecture

Serverless architecture or *Function as a Service* is a recent development in cloud client-server systems. In the traditional approach, a thin-client request invokes some services on a server. The server represents a monolithic unit implementing all the logic for authentication, page navigation, searching and transactions. In a *Serverless architecture*, we create a much richer client and decompose the server into a set of small, event-triggered and (ideally) stateless functions (Fig. 1). Those functions are fully managed by a 3rd party. The state is recorded on the client or stored in a database and can be transmitted using tokens. The client is also responsible for invoking services in order and manages the logic of the application. Such an approach presents several advantages. Firstly, there is no need for dedicated hardware to support the system. Instead, all functions are uploaded in the cloud and invoked if necessary. Secondly, decomposition allows to handle traffic peaks better and use *pay as you go* pricing systems. Thirdly, the system is more resilient to DDOS attacks, as it is easier to attack a standalone server than a distributed cloud of smaller functions. The *serverless architecture* is

already implemented on existing platforms such as AWS Lambda[2] or Google Cloud[3]. We argue that the serverless approach to executing *functions as a service* is a perfect fit for edge/fog computing environments, as it increases flexibility in managing edge-clouds.

2.3 Serverless Unikernels for NDN

We argue that the Serverless Architecture is a required component that needs to be integrated into the NDN stack to deliver edge network functions. The requesting node requests functions by name, which can then be invoked on any node while the results follow the path established by the Interest packet. Because function state is managed by clients, consecutive calls for the same function can be served by different nodes without any handover process. Architectures in which services can be executed only on a given set of nodes (Sec. 5) must find a service and then communicating de facto with the hosting node, which is contradictory to ICN principles. Even if service handover is supported, in traditional architectures the process requires locating the node previously executing the function to synchronise the state. Instead, in the serverless architecture, the initial state is attached to the Interest packet (Sec. 3.2), while updated state (after the function execution) is sent to the client in the resulting data packet. If maintaining state requires large amounts of memory, the state itself can be stored as a named data chunk in the network and be requested by the node executing the function.

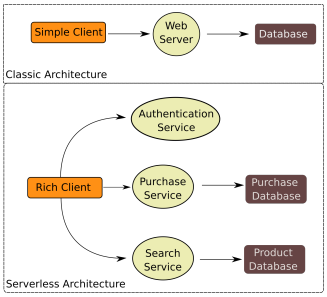


Figure 1: Serverless architecture.

3 SYSTEM DESIGN

3.1 Overview

In NFaaS, a node requesting a job, sends an interest with the kernel name requesting its execution (we discuss function naming in Sec. 3.2). A router receiving the interest, checks if it has the unikernel stored locally. If it does and enough CPU resources are available, it is instantiated and receives the Interest as its input (we discuss function storage and execution in Sec. 3.3). The Interest packet includes all the required input parameters, *e.g.,* a state token, or additional named-data required by the function (Sec. 3.2).

The unikernel performs the requested action and sends back the result as a data packet. The response follows the downstream path constructed by the input Interest and reaches the requesting node. If more input data is required (*e.g.,* in applications requiring image processing), the instantiated unikernel sends interest packets towards the requesting node that can provide it with the necessary input. If the unikernel is not present or the router does not have enough resources to run the function, the initial interest is forwarded following rules described in Sec. 3.4.1. Fig. 2 presents an overview of the system. An Interest to execute function `/foo/bar` issued by node A is forwarded to node C, where the corresponding kernel is instantiated. The resulting data is sent back following the same path.

The main entity that manages storage and execution of named functions is the *Kernel Store* (discussed in Sec. 3.3). The *Kernel Store* keeps historical statistics to make decisions on which functions to download locally and which ones to execute. The goal is to proactively place delay-sensitive functions as close to the edge as possible and push functions with relaxed delay sensitivity requirements further towards the core.

Finally, functions are resolved based on a routing protocol and two forwarding strategies (discussed in Sec. 3.4). The purpose of the forwarding strategies is to find the requested functions in the neighbourhood.

The combination of the above components: *i) function naming* (Sec. 3.2), *ii) function storage and execution* (Sec. 3.3), and *iii) function resolution* (Sec. 3.4), results in a *decentralised system of executable, mobile named-functions*. Each component is described in detail in the following sections. Under steady-state, functions are placed according to their requirements, *e.g.,* for delay-sensitivity, the system load-balances computation among nodes, it adapts quickly to changing network conditions and incures minimum control overhead.

Fig. 3 provides a high-level overview of the operation. The node in the middle receives an interest for a function it does not have stored locally and hence, forwards the Interest towards the cloud (Fig. 3b). After seeing enough requests for the same function, it decides to download the unikernel and run the service locally (Fig. 3b). If the demand for this service exceeds the node's capacities, the node will start forwarding part of the Interests to the next node, which might also decide to run the unikernel itself or forward to its own neighbours.

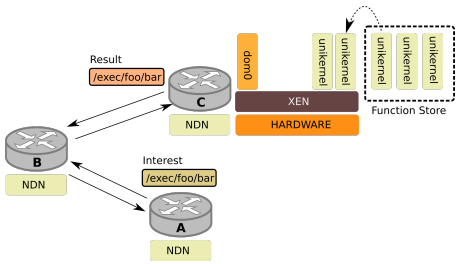
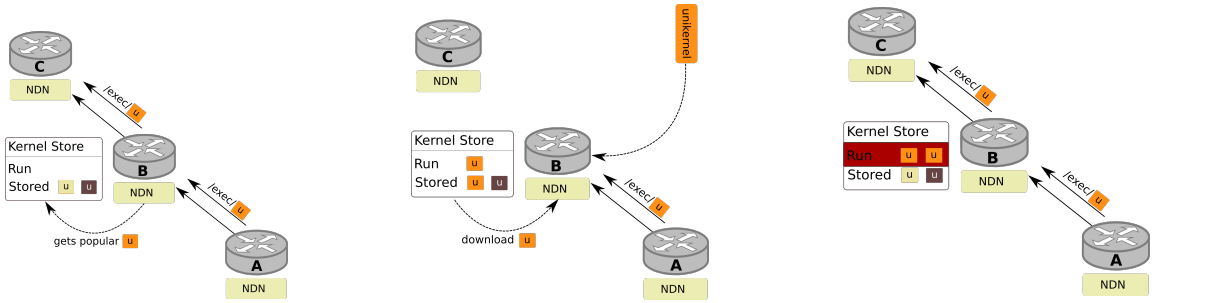


Figure 2: System architecture.

3.2 Naming Moving Functions

In NFaaS, Interest packets can request either the function (*i.e.,* unikernel) itself, or the execution of the function. While requesting the function itself is a straightforward content request and follows the naming structure (and consequently the routing and forwarding

[2] https://aws.amazon.com/
[3] https://cloud.google.com/



(a) A kernel becomes popular.

(b) Kenel Store requests to download the kernel.

(c) When overloaded, Node B forwards a part of requests towards the cloud.

Figure 3: NFaaSHigh-Level Overview

principles) of the NDN architecture, when requesting execution of some function it is not the case. To request function execution, we insert the `/exec/` prefix in the Interest packet.

We expect that edge-network moving functions will have different application requirements, *e.g.,* with regard to delay-sensitivity. Accordingly, the edge network should be able to deal with different types of tasks having different requirements and priorities, *e.g.,* for each class of tasks, the system needs to make different forwarding decisions, in order to maximise users' Quality of Experience (QoE). We therefore, enhance the `/exec/` prefix with an extra prefix field to indicate application requirements. For the sake of simplicity, in this study we assume two application classes: *i) delay-sensitive*, and *ii) bandwidth-hungry*. Delay-sensitive applications, such as augmented reality or autonomous vehicles, have very strict delay constraints and therefore need to be processed as close to the edge as possible. Bandwidth-hungry applications, on the other hand, generate large amounts of IoT data that needs processing (e.g., to reduce bandwidth usage). To avoid shipping vast amounts of data throughout the network to the distant cloud, edge operators can offer to process this data within the access domain, but not necessarily on the first few hops from the end device. For these two types of applications the `/exec/` prefix is complemented by an extra `delay/` or `bandwidth/` component - Fig. 4. Although more application classes can be used to manipulate function mobility, in this paper we make use of these basic classes in order to benchmark the main system components. Given that each execution request can be accommodated in any node, each client appends user-specific input information in the form of a hash (last part of name in Fig. 4). This is a necessary step to differentiate between consecutive requests for the same function that come from different clients. With a different suffix, each request with new input parameters creates a separate PIT entry.

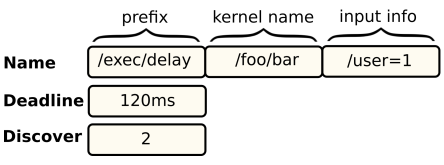


Figure 4: Interest packet structure

Together with the function execution request, the rich client in a serverless architecture appends extra information in the Interest packet in the form of optional Type-Length-Value (TLV) fields. Clients include a *task deadline*, or specify a *Discovery* field to prevent parallel service execution (details in Sec. 3.4). The *task deadline* field is also used to set up a custom value for Pending Interest Table (PIT) entries. Larger tasks set up higher expiry values, allowing data to be returned for a longer period of time. If the produced data consists of more than one chunk or takes a long time to be generated, the executing nodes return just a unique name of the content that will be produced. The client can then fetch it using a separate Interest packet.

The resulting Interest (Fig. 4) includes all the required information in order for the system to make informed forwarding decisions in order to resolve edge-network functions (details on routing, forwarding and resolution are given in Sec. 3.4. The data produced by the function execution follow the same structure. It thus can be cached and sent to subsequent users requesting function execution with exactly the same input parameters (where possible).

3.3 Storing and Executing Moving Functions

Somewhat similar to the Content Store in the NDN architecture, the *Kernel Store* (KS) is responsible for storing unikernels. The KS has to make decisions as to which unikernels to store, given that the unikernel population is much larger than the memory available at each node. In addition to storing unikernels, the Kernel Store is also responsible for deciding which unikernels to actually execute. As mentioned earlier, we consider that a node has more storage capacity than computation capability, *i.e.,* a node can store more functions than it can execute simultaneously. The Kernel Store can mark some of the downloaded kernels as inactive. Only requests for active services are executed.

In order to make decisions on which unikernels to store, which to remove from storage and which ones to execute, the KS keeps statistics from previously observed function execution requests. The statistics are kept in the *Measurements Table*, a structure already existing in the NDN reference implementation [19] and is used among other reasons to keep statistics about data prefixes in

Table 1: Measurement Table entry

Function Name	/delay/foo/bar/			
Deadline	120ms			
Popularity	2/10	7/10	4/10	3/10
Hop Count	2.43			
Faces	netdev1		netdev2	
Delay	94ms		86ms	

CCN forwarding strategy. Entries are kept per *unikernel*, are automatically removed when not refreshed. Tab. 1 presents a sample entry with the following fields:

- *Task deadline:* time (in *ms*) to finish the task and return the data to the requesting user. This value is recorded from the related field in the Interest packet (Fig. 4) and is used to sort functions based on delay-sensitivity.
- *Function popularity:* the percentage of requests for this function during the last i Interests. Each node keeps historical data for m groups of i Interests each. Each node keeps historical data for m groups of n Interests each.
- *Average hop count:* the average number of hops, h_i from the requesting client. Each Interest packet contains a hop count increased at every forwarding node. We use this information to determine how far are nodes sending interest packets.
- *Preferred faces:* a list of faces on which we forward interests for this unikernel. Based on this information, forwarding strategies (Sec. 3.4) decide on which face should each Interest be sent. For previously unseen unikernel requests, there is no entry in the preferred faces field. A forwarding strategy then adds entries based on algorithms presented in Sec. 3.4.2. Sending interests on the same faces allows a service to become popular in a given part of the network. The Kernel Store then makes sure that the corresponding unikernel is present in nodes in that neighbourhood.
- *average service delay:* for each preferred face, each node records the delay between previously forwarded Interests and the corresponding data after the function execution. Based on this information the router determines the average service delay for each unikernel. Service delay variance determines whether or not the neighbour node (indicated in the preferred faces entry) is overloaded.

3.4 Resolving Moving Functions

Based on these measurements, the *Kernel Store* calculates a *unikernel score*, Eq. 1, for every observed unikernel (*i.e.,* `/exec/` Interest for some unikernel). The purpose of the *unikernel score* is to identify the unikernels that are worth downloading locally into the node's memory.

$$unikernel\ score = \sum_{i=0}^{m} \frac{p_i}{n} * (m - i) + (R - h_i) * t_m \qquad (1)$$

In Eq. 1, p_i represents the *Function Popularity* discussed above, albeit in raw figures. That is, p_i is the number of Interests that have crossed this node for this unikernel in the last n requests. Nodes keep record of last m epochs of n packets. h_i represents the average hop count of Interests, as given above and t_m is a binary tuneable parameter to distinguish between delay-sensitive and bandwidth-hungry unikernels (positive for delay sensitive unikernels and negative for bandwidth hungry ones). Finally, R is a system parameter that represents the *radius* (in terms of hops) around a node, effectively splitting the area around a node in two. The area inside R should be kept for delay sensitive functions, while the area outside R for bandwidth-hungry functions. Together with parameter t_m, the product $(R - h_i) * t_m$ is bigger for delay-sensitive functions in the first R hops along the path from the client towards the core of the network, while it gets bigger for bandwidth-hungry functions as the request moves further away.

If a node becomes overloaded with incoming Interests (has a unikernel, but cannot execute it because of the number of other requests) it can reduce the number of active kernels. The KS deactivates the least popular kernels one by one based on Eq. 1. If a kernel becomes inactive, its image is still kept in the store, but it is not instantiated and corresponding interests are forwarded to neighbour nodes. The process is stopped and progressively reversed when the node is able handle the incoming traffic for its active kernels. If a node becomes overloaded, it can also be detected by other nodes through variation in service delay.

The calculated score allows a quick reaction to unikernels becoming popular (the most recent n requests have the highest weights), while avoiding too sudden changes in the KS by keeping historical data on past requests. When increasing m, nodes keep more historical data. It prevents downloading new images too frequently, but slows down reaction to new, popular kernels. Increasing n assures more fluent score evolution, with the cost of increased memory consumption. The ultimate purpose of the *unikernel score* is to *encourage nodes closer to the edge to download and activate delay-sensitive functions, indirectly leaving storage and computation capacity for bandwidth-hungry applications towards the core of the domain.* In doing so, the domain is vaguely split in unikernel-specific areas, where groups of nodes focus on specific function requirements. This effect is further magnified by reducing the amount of active kernels.

Central to the design of a distributed edge computing system is *resolution of moving functions.* We realise function resolution through a combination of: *i)* a signalling-based routing protocol, and *ii)* two separate forwarding strategies, one for each type of service supported in this work (*i.e.,* delay-sensitive and bandwidth-hungry).

3.4.1 Routing Protocol. We use the standard NDN routing protocol, NLSR [20] as implemented in the NDN reference implementation, which is based on prefix advertisement. In particular, the first node in a domain that decides to download a unikernel (based on the *unikernel score* in KS) becomes the reference node for this specific unikernel and advertises the corresponding prefix *e.g.,* `/exec/foo/bar/`. Eventually, prefix advertisements propagate throughout the network at the intra-domain level. Given that the *Kernel Store* in a unikernel-based system is less dynamic than the *Content Store* in terms of item replacement, a node becomes the default execution location for the prefixes it advertises within the domain. Note that this does not prevent other nodes from downloading and running an already advertised unikernel at a different

location. Instead, it is the job of the *Forwarding Strategy* to resolve unikernels at different (than advertised) locations.

Fig. 5 presents a topology where Node C stores and can run two unikernels. Once Node C becomes overloaded, Node A detects the face towards C as overloaded (Sec. 3.3) and consults its Forwarding Information Base (FIB) table for alternative routes. However, as C advertises both unikernels, its FIB table returns the same path, again towards Node C. To reduce the load towards itself, Node C marks kernel 1 as "inactive" and stops advertising its name. Node F has this function in store, spots the lack of advertisement from Node C and can thus start advertising. Node F will now become a new default route for this kernel in Node A's FIB table. When Node C stops being overloaded it can again activate the first service and execute it locally. Node A, having now two discovered preferred faces for the same function in its Kernel Store can perform load balancing between them.

3.4.2 Forwarding Strategies. In NFaaS we implement two forwarding strategies. One to forward interests for "delay sensitive" services (with the prefix `/exec/delay/` preceding the unikernel name) and another one for "bandwidth hungry" services (with the prefix `/exec/bandwidth/` - Fig. 4). Depending on the prefix, the corresponding forwarding strategy is invoked, if the node cannot execute the unikernel locally. In our current implementation, Interest packets are not queued waiting for the image to be downloaded, or for computational capacity to become available. Instead, if the node does not have the unikernel or does not have available CPU, it immediately forwards the packet and depending on the *unikernel score* (Eq. 1) the KS decides whether to download the function for future use. Although alternative queue-based designs are possible, diversity of execution times for different functions achieves full utilisation of the system's computation capacity.

Delay-sensitive forwarding strategy: By default, and if nodes do not have "preferred faces" for the requested unikernel (*i.e.,* faces that point to a domain node), the Interest is forwarded towards the cloud according to the FIB entry. At the same time, in order to discover whether the unikernel exists in the immediate neighbourhood and meets its strict deadline, this strategy performs *scoped flooding of a discovery message.*[4] The flooded interest contains a special field indicating that it is a *discovery Interest* in order to avoid actual execution of the function. Upon reception of a *discovery Interest*, if a node has and can execute the corresponding service, it will respond with a dummy data packet. This information is then kept at the Measurement Table of the KS for future use.

In case multiple "preferred faces" exist in a node's Measurement Table the node can perform load-balancing between them .Although specific load-balancing algorithms can be deployed, they are outside the scope of the present study. Here we simply use "round-robin". As mentioned earlier, a face is marked as *overloaded* if the delay from the most recent data packet is higher than the average previously experienced for this service. If an interest sent on a face times out or the data is sent with a delay higher than a threshold, the face will be removed from the preferred faces set.

Algorithm 1 presents the pseudocode for the *delay-sensitive forwarding strategy.*

[4] According to [21], a scope equal to two achieves the right tradeoff between efficiency and signalling overhead, hence, we use this value in our evaluation.

```
Data: Interest packet
Result: Output face
outFace = null;
while face = prefferedFaces.hasNext() do
    if !face.isOverloaded() then
        score = loadBalancer.calculateScore(face) if
         score > maxScore then
            maxScore = score;
            outFace = face;
        end
    end
end
if outFace == null then
    sendDiscoveryPackets(interest);
    outFace = FIB.getCloudFace();
end
return outFace
```

Algorithm 1: Delay constrained forwarding strategy

Bandwidth-hungry forwarding strategy This forwarding strategy deals with services with high bandwidth usage and softer delay constraints. In this case, it is important to keep the task within the edge-domain, but not necessarily close to the requesting node. If the strategy does not find a preferred face to forward the interest, this forwarding strategy does not use the discovery mechanism. Instead, it directly forwards the Interest to the next hop towards the advertising node or the cloud, as indicated by the FIB table.

Algorithm 2 presents the pseudocode for the *bandwidth-hungry forwarding strategy.*

```
Data: Interest packet
Result: Output face
outFace = null;
while face = prefferedFaces.hasNext() do
    if !face.isOverloaded() then
        score = loadBalancer.calculateScore(face) if
         score > maxScore then
            maxScore = score;
            outFace = face;
        end
    end
end
if outFace == null then
    FIB.getPrefixFace();
end
if outFace == null then
    FIB.getCloudFace();
end
return outFace;
```

Algorithm 2: Bandwidth hungry forwarding strategy

3.5 Security Considerations

The addition of function execution in ICN opens many security issues. Each unikernel should be signed by its publisher and only

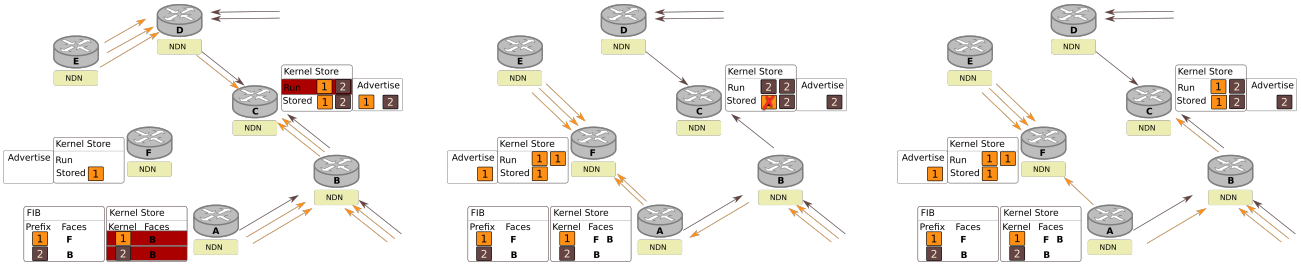


Figure 5: Prefix advertisement mechanism.

trusted images should be considered for execution. Recent hypervisors provide a solid isolation layer protecting the hosting OS and its filesystem, unless misconfigured. To protect against DoS (Denial of Service) attacks and allow charging for execution, the Interests should also be signed. However, including the signature in the name (as it is currently done in the NDN stack) breaks the caching. On the other hand, keeping it out allows a malicious user to block requests from a valid one, because of the Interest integration in the PIT. In many scenarios, the input parameters as well as the resulting data should be keep private. An efficient system for key distribution, integrating private data into the caching system and truly secure communication between clients and executing nodes are topics of our future work.

4 EVALUATION

To evaluate the performance of NFaaSwe ran extensive simulations and created a real-world prototype. We also make our implementation available to the research community. [5]

4.1 Simulations

All simulations were performed using ndnSIMv2.3 [22] - an NDN simulator based on ns-3 and using the most recent releases of the NFD daemon and the ndn-cxx library[6]. All presented values are an average of at least 5 consecutive runs.

Fist, we illustrate the behaviour of the Kernel Store function presented in Eq. 1. Fig. 7 presents the evolution of the score on two different nodes. Node 1, located close to the Interest origin ($h_c = 0$) and Node 2, located further away ($h_c = 3$). We consider *Range* of 2 hops ($R = 2$) and gather popularity statistics for 3 groups ($m = 3$) of 3 packets. We send Interests for two functions of different types: "delay-sensitive" (D) "and bandwidth-hungry" (B). We start with no statistics for any kernel. X-axis shows the times of arrival for packets of each type. When first 3 packets are received, the score for kernel D starts to rise on both nodes. However, since Node 1 is located closer to the source, the score rises faster up to the value of 13. With consecutive Interests, the score remains stable. When Interests for kernel B are received, its score on Node 2 quickly rises up to the value of 8. Because Node 1 is close to the source, it keeps the score for B low. When the 7th Interest arrives, the weight of already collected data is decreased and the score for two kernels drops on both nodes. The *unikernel score* can differentiate between "delay sensitive" and "bandwidth hungry" tasks and quickly react to new Interests while taking into account historical data.

4.1.1 Small Topology. For our initial tests, we create a small topology containing 16 nodes. Nodes 1, 2, 4, 5, 7, 8 become sources where interests are generated and Node 16 has a direct connection to the cloud (Fig. 6). Each link introduces a 10ms delay, while the connection to the cloud is 50ms. After being invoked, tasks run for 100ms before returning the data. All nodes have the same CPU power and execute up to 2 tasks in parallel. We consider a domain of 20 services, half of which are "delay sensitive" and the other half are "bandwidth hungry". Each node can store up to 10 kernels unless specified differently.

With our first simulations, we investigate the specialization process. We want the "delay sensitive" tasks to be executed closer to the sources and minimize their delay, while keeping the "bandwidth hungry" further away. At the same time, if the size of the KS is lower than the kernel domain, we want nodes to execute a fixed group of services. Otherwise, they constantly have to download new images, producing additional overhead.

Fig. 6 presents the ratio of executed tasks of two types for each node. The source nodes execute almost exclusively "delay sensitive" tasks. The Kernel Store calculates a high score for these services because of the low hop count (Eq. 1). For nodes located further away from the sources, this impact is reduced and "bandwidth hungry" tasks are executed more frequently. Finally, nodes located close to the gateway, host mainly this type of services.

We then investigate the number of executions for each function on different nodes. Fig. 9 presents the results when each node can store all requested functions in the network. In this case, nodes perform a similar number of executions for a vast palette of kernels. When the size of the KS is reduced so that each node can download images for only 25% of services, we observe a much higher specialization (Fig. 10). Nodes located close to the sources execute 5 types of tasks with the highest score. Their interests are thus not forwarded deeper in the network, making them less popular on the other nodes. The process continues so that all the functions are being executed and each node continues to run the same types of tasks. According to our design goals, this means that the content of the KSs remains stable and new images are not frantically downloaded.

We continue by observing the success rate, the average delay and the amount of tasks executed locally. A "delay sensitive" task is considered satisfied if its RTT is lower than 50ms. A "bandwidth hungry" task is satisfied if its executed anywhere within the domain.

[5] https://gitlab.com/mharnen/edge_computing/
[6] https://github.com/named-data/

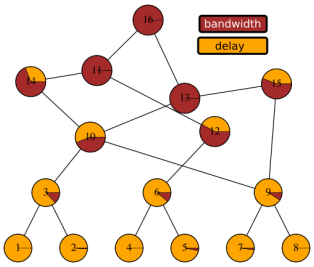


Figure 6: "Delay sensitive" and "bandwidth hungry" functions placement.

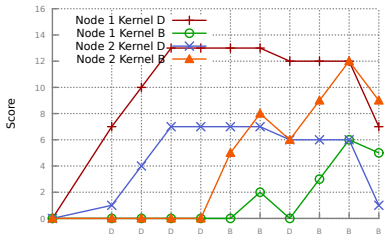


Figure 7: Score function evolution.

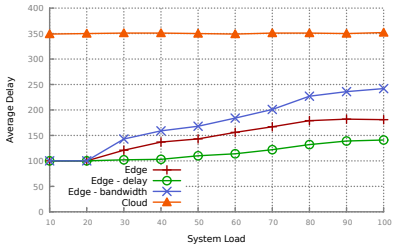


Figure 8: Average delay for different system load values.

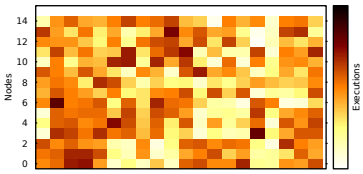


Figure 9: Function executions on different nodes with 100% KS storage size.

Figure 10: Function executions on different nodes with 25% KS storage size.

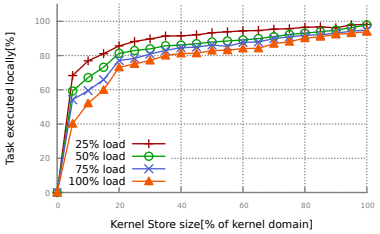


Figure 11: Locally executed tasks for different KS storage size values.

We test the delay between sending an Interest packet and receiving the corresponding data for different values of system load. Results are shown in Fig. 8 with the KS size equal to 25% of the kernel domain. The average response time increases with load, as nodes located closer to the cloud get involved introducing additional delay. Up to 28% load, NFaaSis able to execute all generated tasks directly at the edge nodes introducing almost no delay, apart from the function execution time (100ms). For higher system load values, NFaaShosts the "delay sensitive" tasks close to the sources, keeping their average RTT below 50ms even for full system load. On the other hand, the "bandwidth hungry" tasks are pushed deeper into the network, resulting in higher delay values of up to 250ms.

We then directly test the task satisfaction rate. Fig. 12 shows the importance of differentiating tasks by the KS (Eq. 1). Without this mechanism, the satisfaction rate drops to 64% for the full system load. This is mostly due to delayed "delay sensitive" tasks, that are executed on random nodes in the network. When the mechanism is applied, these tasks are kept closer to the sources reducing the delay.

To investigate the impact of the KS size on the system performance, we run simulations for different system loads (Fig. 11). Choosing the right KS size is a crucial task; we want to execute all the tasks at the edge, while minimizing the storage footprint. For all the scenarios, increasing the size of the KS causes more tasks to be executed locally. When each node can store 20% of services, the edge is able to execute 80% of the tasks. The effect of further incrementing the KS size is much lower. Increasing the load does not have a significant impact on the execution rate. We observe that NFaaS is able to distribute tasks evenly and fully utilize the network resources.

This observation is further confirmed by directly investigating the load share. Fig. 14 shows the CPU power utilisation for each node when the KS size is equal to 25% of the task domain. Even for low system load (20%) the tasks are spread equally in the network. Nodes close to the sources have enough power to execute most of the tasks, but do not have all the images. As shown previously (Fig. 10), each KS specializes in several services, requiring the whole network to be involved. The proportions are kept also for higher system load values. Equal load share is beneficial, especially in IoT scenarios, when it determines equal energy consumption.

We then measure the response time to traffic changes to verify how fast the system can react if new types of tasks are requested. We keep the background traffic from previous experiments (50% system load) and introduce a source of new interest that will jump between source nodes (1, 2, 4, 5, 7, 8) every second. Fig. 13 presents the results for introduced "delay-sensitive" and "bandwidth-hungry" tasks. Both services start by being executed in the cloud, as the images are not present at the edge. Once they are download, the delay drops to 101ms and 146ms respectively. The "delay-sensitive" task resides now at the closest node to the source while the "bandwidth-hungry" - 3 hops further away. When the tasks are switched to Node 2 (1s), the "delay-sensitive" is initially forwarded again to the cloud, resulting in higher delay. However, the scoped-flooding mechanism discovers the kernel present on Node 1 and redirects the traffic there. Finally, the task image is download on Node 2 and the delay drops to the minimal value. During the shift, the "bandwidth-hungry" task is not forwarded to the cloud, but to the node advertising the kernel prefix. This results in much smaller jitter. The delay drops further when the image is moved closer to the new source (Node 2). For the "bandwidth-hungry" task, each transition results in a similar delay change. However, when the "delay-sensitive" task

migrates from Node 2 to Node 3, the scoped-flooding is not able to find the previous node executing the service. All the requests must be thus forwarded to the cloud until the corresponding image is downloaded. For all our scenarios, NFaaSis able to download and instantiate new services in the most optimal places within 400ms.

4.1.2 RocketFuel topology. We continue the evaluation on a large network using the RocketFuel 1239 - Sprintlink topology [23], containing 319 nodes. We choose routers having only one link as request sources (31 nodes) and one with the highest number of links (54 links) as a gateway towards the cloud. We consider a domain of 10 000 services, and that each node can store up to 2000 kernels.

We measure the percentage of satisfied tasks and present the results in Fig. 15. For small values of system load, NFaaSis able to satisfy all the generated requests. However, with the bigger and less regular topology the "delay sensitive" tasks start to be delayed even with 30% system load. The scoped flooding mechanism is not able to reach a large part of the network and the interests are directly forwarded to the cloud. However, in this topology, the majority of nodes are located closer to the core of the network, making it impossible to satisfy all the "delay sensitive" requests even with an optimal solution. On the other hand, "bandwidth hungry" tasks get satisfied even for the highest system load values. The prefix advertisement mechanism is able to distribute the interests between multiple parts of the network.

We continue by verifying the impact of increasing the KS size (Fig. 16). Similarly to the tests with the small topology, increasing the size, increases also the amount of tasks executed at the edge. However, the positive effect is much smaller. We observe a significant impact of the system load on the results that was not visible in the small topology. Executing more than 80% tasks locally is possible only for the scenario with 25% load. Again, the majority of tasks forwarded to the cloud are "delay sensitive".

The same problems influence the delay (shown on Fig. 17). For system load up to 20% the network is able to execute all the "delay-sensitive" tasks close to the sources. However, when the load increases more and more Interest packets are forwarded towards the cloud increasing the average deadline. The prefix advertisement mechanism performs better, distributing tasks to a greater number of nodes. Because of that, "bandwidth-hungry" functions, even being kept further from the sources, can achieve a lower average delay.

In summary, scoped flooding achieves extremely low delays and good satisfaction rates for small and regular topologies, but struggles in bigger and less regular ones. Prefix announcement offers slightly higher delays, but remains a much more flexible solution for keeping tasks at the edge. Nevertheless, NFaaSexecutes the majority of the requested tasks at the edge in all our scenarios and significantly reduces the delay.

4.2 Prototype

To confirm the system performance we created a real-world prototype of our system deployed on 2 nodes (Dell XPS13 laptops). NFaaSimplementation was written using the most recent release of the NDN stack (v. 0.5.1). We create our unikernels using Rumprun [14]. The system is based on Unix systems and can compile sources written in the most popular languages into unikernels, including only the required system components.

Each of the created services is written in C, takes a corresponding interest as an input, stays busy for a period of time indicated in the packet and sends a data packet as an output. Unikernels are run directly on XEN hypervisor. Each image requires 19MB of storage and takes 36ms to be instantiated when the image is already loaded into RAM and has a dedicated core. Transferring an image on top of Gigabit Ethernet between two, directly connected nodes takes up to 200ms. The instantiation time lowers with each new release of Rumprun and is expected to be further reduced. On a standard 512GB hard driver, we are able to store 26947 unikernels.

5 RELATED WORK

ICN was designed as an alternative to the current Internet routing architecture. Its emphasis is on building a data-centric network infrastructure. Instead of the classic point-to-point communication paradigm, ICN proposes accessing and disseminating content by name and offers universal, native caching natively at the routing layer. Initially ICN allowed naming only static content [24]. However, with the recent advance in cloud computing and virtualisation technologies, multiple proposals for ICN-based service invocation emerged.

Shanbhag et al. presents SoCCer [5] - a control layer on top of CCN for the manipulation of the underlying Forwarding Information Base (FIB) so that it always points to the best service instance at any point in time. Braun et al. introduce Service-Centric Networking [6], an ICN stack enhancement for service execution. The authors use uniform naming of services and content by using an object-oriented approach that introduces object names for both services and content. Users can thus request service execution that will be routed towards the closest server hosting them. These solutions focus on bringing an execution interest to the closest hosting server. However, in many scenarios the closest node can still be many hops away introducing a significant overhead and services cannot be easily migrated on demand. What is more, scalability remains an issue. One server can be overloaded by requests, while another one will remain idle.

Another approach consist of enhancing the network to support λ-expressions [7]. The names are extended from identifying only static content, to orchestrate computations. The system allows also for creating complex recipes involving many sub-operations. This empowers the network to select internally places for fulfilling user expression using forwarding optimization and routing policies.

To tackle the scalability problem, several works focus on service migration outside the ICN stack. SCAFFOLD is an architecture that provides flow-based anycast with (possibly moving) service instances. SCAFFOLD allows addresses to change as end-points move, in order to retain the scalability advantages of hierarchical addressing [25]. The authors of "Service Oriented Networking" [26] propose a multi-tier architecture for environments with multiple cloudlets. The system has a global view of the topology, can migrate services and route packets to the optimal server. However, the project focuses on a different, more managed environment with inter-domain routing. Similar to NDN, Serval [27]resides above the

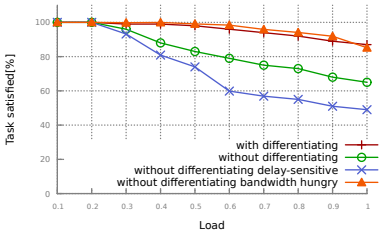


Figure 12: Task satisfaction rate for different system load values.

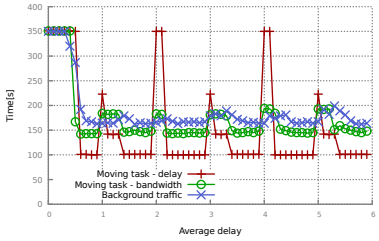


Figure 13: Delay evolution for moving tasks.

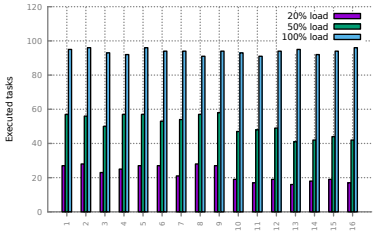


Figure 14: Tasks executed on different nodes.

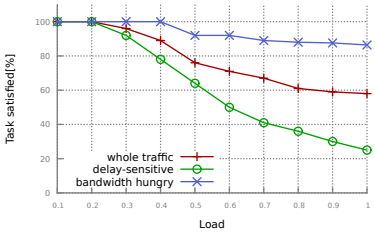


Figure 15: Task satisfaction rate for different system load values.

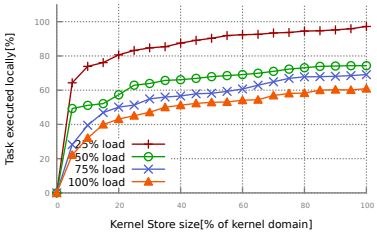


Figure 16: Locally executed tasks for different KS storage size values.

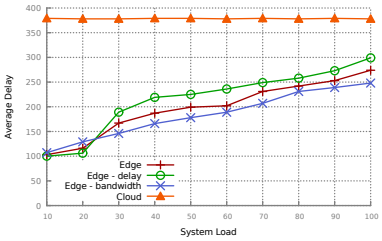


Figure 17: Average delay for different system load values.

unmodified IP network layer and enables applications to communicate directly on service names. Serval connects clients to services via diverse discovery and allows end-points to seamlessly change network addresses, migrate flows across interfaces, or establish additional flows for efficient service access.

Finally, an approach similar to ours is proposed by Sathiaseelan et al. [28]. SCANDEX is a Service Centric Networking framework, albeit for challenged decentralised networks. The system also represents services as unikernels that can be easily migrated and executed on any node. However, the authors propose a DNS-like system of brokers to keep track of all deployed services in the local network. It forces nodes to resolve service IDs before contacting the hosting node introducing additional delay and control traffic required by the registration process.

Wang et al. propose "C3PO" - a computation congestion management system [29]. The authors consider a scenario in which incoming tasks should be distributed between nodes. The paper evaluates two strategies: i) a passive one (where a node does as much as it can and when overloaded, it passes tasks to its neighbour) ii) a proactive one (introducing cooperation between nodes). "C3PO" does not deal with other aspects of service execution, but can be a valuable addition to our framework in the future.

6 CONCLUSION

We introduced the concept of *Named Function as a Service* (NFaaS), which we argue is ripe to become a main building block of the NDN architecture as we move to an era where *the network is seen as a computer* and computation moves closer to the network edge. NFaaSenables seamless execution of stateless microservices, which can run at any node in the network. We argue that the newly-proposed *serverless architecture* is the right vehicle to realise named functions at the network edge, given that state is kept at rich end-clients.

The resulting platform includes new components, such as the *Kernel Store*, which stores function code and also makes decisions on which functions to execute. The KS also incorporates (currently two) forwarding strategies in order to resolve the functions as they migrate in the network of edge-computing nodes. Although more forwarding strategies can be incorporated in NFaaS, the ones introduced here present promising first results.

We ran extensive simulations of our system, created a real-world prototype and made the implementation available to the research community.

According to the two forwarding strategies (on delay-sensitive and bandwidth-hungry services), functions move to the right direction (*i.e.,* the majority of delay-sensitive towards the very edge of the network, while bandwidth-hungry towards the core). Our prototype implementation of NFaaSindeed validates our design goals: services can be instantiated in less than 40ms, while the KS can store tens of thousands of functions.

To the best of our knowledge, NFaaSis the first framework enabling to migrate functions closer to the user in ICN environment without a global view of the network.

REFERENCES

[1] J. Rodriguez, *Fundamentals of 5G mobile networks.* John Wiley & Sons, 2015.
[2] J. Rivera and R. van der Meulen, "Gartner says the internet of things installed base will grow to 26 billion units by 2020," *Stamford, conn., December*, vol. 12, 2013.
[3] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 37–42, 2015.
[4] A. Ahmed and E. Ahmed, "A survey on mobile edge computing," in *Intelligent Systems and Control (ISCO), 2016 10th International Conference on.* IEEE, 2016,

pp. 1–8.
[5] S. Shanbhag, N. Schwan, I. Rimac, and M. Varvello, "Soccer: Services over content-centric routing," in *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*. ACM, 2011, pp. 62–67.
[6] T. Braun, A. Mauthe, and V. Siris, "Service-centric networking extensions," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 583–590.
[7] M. Sifalakis, B. Kohler, C. Scherb, and C. Tschudin, "An information centric network for computing the distribution of computations," in *Proceedings of the 1st international conference on Information-centric networking*. ACM, 2014, pp. 137–146.
[8] M. Arumaithurai, J. Chen, E. Monticelli, X. Fu, and K. K. Ramakrishnan, "Exploiting icn for flexible management of software-defined networks," in *Proceedings of the 1st ACM Conference on Information-Centric Networking*, ser. ACM-ICN '14. New York, NY, USA: ACM, 2014, pp. 107–116. [Online]. Available: http://doi.acm.org/10.1145/2660129.2660147
[9] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *Queue*, vol. 11, no. 11, p. 30, 2013.
[10] J. Spillner, "Snafu: Function-as-a-service (faas) runtime design and implementation," *arXiv preprint arXiv:1703.07562*, 2017.
[11] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How to make your application scale," *arXiv preprint arXiv:1702.07149*, 2017.
[12] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *Elastic*, vol. 60, p. 80, 2016.
[13] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.
[14] A. Kantee and J. Cormack, "Rump kernels: No os? no problem!" USENIX.
[15] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*. IEEE, 2015, pp. 250–257.
[16] "Mirage os," https://mirage.io.
[17] M. Plauth, L. Feinbube, and A. Polze, "A performance evaluation of lightweight approaches to virtualization," *CLOUD COMPUTING 2017*, p. 14, 2017.
[18] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam *et al.*, "Jitsu: Just-in-time summoning of unikernels." in *NSDI*, 2015, pp. 559–573.
[19] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto *et al.*, "Nfd developer's guide," *Technical Report NDN-0021, NDN*, 2014.
[20] A. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "Nlsr: named-data link state routing protocol," in *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*. ACM, 2013, pp. 15–20.
[21] L. Wang, S. Bayhan, J. Ott, J. Kangasharju, A. Sathiaseelan, and J. Crowcroft, "Prodiluvian: Understanding scoped-flooding for content discovery in information-centric networking," in *Proceedings of the 2nd International Conference on Information-Centric Networking*. ACM, 2015, pp. 9–18.
[22] A. Afanasyev, I. Moiseenko, L. Zhang *et al.*, "ndnsim: Ndn simulator for ns-3," *University of California, Los Angeles, Tech. Rep*, 2012.
[23] N. Spring, R. Mahajan, and D. Wetherall, "Measuring isp topologies with rocketfuel," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4, pp. 133–145, 2002.
[24] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 2009, pp. 1–12.
[25] M. J. Freedman, M. Arye, P. Gopalan, S. Y. Ko, E. Nordstrom, J. Rexford, and D. Shue, "Service-centric networking with scaffold," DTIC Document, Tech. Rep., 2010.
[26] D. Griffin, M. Rio, P. Simoens, P. Smet, F. Vandeputte, L. Vermoesen, D. Bursztynowski, and F. Schamel, "Service oriented networking," in *Networks and Communications (EuCNC), 2014 European Conference on*. IEEE, 2014, pp. 1–5.
[27] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman, "Serval: An end-host stack for service-centric networking," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX association, 2012, pp. 7–7.
[28] A. Sathiaseelan, L. Wang, A. Aucinas, G. Tyson, and J. Crowcroft, "Scandex: Service centric networking for challenged decentralised networks," in *Proceedings of the 2015 Workshop on Do-it-yourself Networking: an Interdisciplinary Approach*. ACM, 2015, pp. 15–20.
[29] L. Wang, M. Almeida, J. Blackburn, and J. Crowcroft, "C3po: Computation congestion control (proactive)," in *Proceedings of the 2016 conference on 3rd ACM Conference on Information-Centric Networking*. ACM, 2016, pp. 231–236.