Keyword-based information retrieval for the WoT

George Xylomenos, Evangelos Zafeiratos, Marios Prokopakis
Department of Informatics
Athens University of Economics and Business
Greece

***Abstract*—The Internet of Things (IoT) is expected to contain huge numbers of "things" producing vast amounts of information. To turn these raw data to useful services for the Web of Things (WoT), we have previously proposed KIOT, a keyword-based scheme for gathering and processing IoT information using Information-Centric Networking (ICN) techniques. In KIOT data items, such as sensor readings, can be named with arbitrary sets of keywords, while users can retrieve all data items matching a desired set of keywords and (optionally) process them with arbitrary functions. In this paper we focus on a prototype implementation of the data retrieval part of KIOT. To maximize flexibility in diverse settings, our implementation automatically configures the network and its routing tables, allowing arbitrary sets of keywords to be used for both data items and queries. Our implementation can be used on any IoT device supporting Java, and is also available for large scale testing over emulated networks using Mininet.**

***Index Terms*—WoT, IoT, ICN, NDN, KIOT**

I. Introduction

The term *Internet of Things* (IoT) refers to the interconnection of everyday objects, from industrial machinery to wearable devices, for the purpose of obtaining information and acting upon it. Due to the large amount of data involved in IoT applications, both due to the number of available devices and their continuous production of information, many cloud-based solutions have been proposed to aggregate and process such data. However, these solutions require continuous connectivity to the cloud, impose lengthy response times on applications and involve large amounts of data transfer. We instead propose moving (at least part of) that computation to the edge, creating a *Web of Things* (WoT) layer to refine the raw data extracted from devices.

To move processing to the edge, we need methods for naming data, gathering them for processing and applying functions to them. Rather than naming data items with URLs, which are hard to maintain when huge numbers of devices are available, we instead propose naming only the data, as in *Information Centric Networking* (ICN) [1]. In the most prevalent ICN approach, *Named-Data Networking* (NDN) [2], data names are hierarchical. For example, data names could be organized by location (e.g., `/aueb/troias/floor1/...`) to help applications get all data items from a building. This naming architecture, however, offers limited expressiveness. If an application wants to gather information by sensor type, for example temperature, it would prefer to organize names by type (e.g., `/temperature/floor1/troias/...`). Ideally, each application should be able to use its own naming scheme, thus allowing the same IoT devices to be used for entirely different purposes, rather than trying to fit all applications into a single naming hierarchy.

To achieve this goal, we have proposed an alternative naming scheme, KIOT, which uses keywords for data naming [3]. In KIOT, data items produced by an IoT device are described by an unordered set of keywords such as {`temperature, aueb, troias, floor2`}, which could indicate a temperature sensor located at the 2nd floor of the Troias building of AUEB. With KIOT, queries can use any subset of keywords, so we can ask for all the temperature readings on the 2nd floor of all AUEB buildings with the set {`temperature, aueb, floor2`}, or for all the temperature readings on the Troias building with the set {`temperature, aueb, troias`}. The KIOT scheme is completely oblivious to the keywords used and their meaning, only using set operations to match data with queries. Specifically, if the keywords in a query are a subset of the keywords in a data item, then the data item is considered to match the query.

While KIOT also includes the application of functions to the data items gathered, which involves locating executable functions and bringing them together with the data to an appropriate node for execution, our work in that area is still at the modeling stage [4]. This paper presents a prototype implementation of the data retrieval part of KIOT, which involves naming data, discovering the network topology, creating routing tables, sending queries and gathering responses. Rather than a simulated model as in [3], we present an implementation in Java with a practical topology discovery and routing protocol that requires minimal configuration and covers many issues involved in actual deployments.

The rest of this paper is structured as follows. In Section II we provide background information on KIOT and related work, while in Section III we describe our prototype implementation. In Section IV we present an initial evaluation of the performance of our prototype with keyword-based queries. Finally, we describe ongoing and future work in Section V.

II. Background and Related Work

While our naming scheme is more flexible than the hierarchical naming of NDN, our design intends to be compatible with NDN as far as possible, to allow interfacing wide-area NDN networks with IOT networks using KIOT. To this end, we borrow the basic communication model of NDN which consists of sending *Interest* packets to ask for a specific data item and responding with *Data* packets holding the matching content. Our nodes also rely on a *Forwarding Information Base* (FIB) which serves as a routing table, allowing Interests to be forwarded towards the appropriate data items, and a *Content Store* (CS) which allows data items from previous queries to be cached so as to answer new queries faster. However, our messages and data structures use encoded sets of keywords rather than the hierarchical names of NDN.

On the other hand, KIOT does not employ the *Pending Interest Table* (PIT) of NDN which stores information on forwarded Interests. NDN uses the PIT to return Data messages to the source of the matching Interest and to ensure that at most a single Data is returned in response to an Interest. In KIOT we overlay a logical tree over the network, where all Interests originate at the root and all Data are returned there. As our eventual goal is to process the Data returned, we expect multiple Data packets to be returned in response to a single Interest. Furthermore, rather than expecting a specific number of Data items to be returned as in [5], we intentionally leave open the number of returned Data items, relying on timeouts to limit the waiting time for responses to an Interest. The rationale is that we should not need to know the exact number of (say) temperature sensors in a building in order to ask for its average temperature.

The KIOT scheme for representing keywords was inspired by TagNet [6], an ICN architecture using tags to name content items at the global level. TagNet proposed representing sets of tags as Bloom filters [7]. KIOT uses the same idea, but while in TagNet a query matches a data item if the tags in the request are a superset of those in the data item, in KIOT a query matches a data item if the keywords in the request are a subset of those in the data item. This allows the FIBs in KIOT to be very simple, since a node can simply summarize the keywords of all its children by merging the Bloom filters representing their keywords in a new Bloom filter. Furthermore, while TagNet uses multiple trees for Data routing, in KIOT only a single logical tree is used, therefore Data routing is trivial: nodes always pass Data to their parent.

III. The KIOT Implementation

In this section we describe the parts of the KIOT architecture [3] that have been included in our prototype implementation, explaining how the initial design has been adapted for implementation purposes.

A. Name Encoding

The KIOT naming scheme is based on using an arbitrary set of keywords to represent each data item and each query, for example, {`temperature, aueb, troias, floor2, mmlab`}. We say that a data item *matches* a query when the keyword set of the query is a subset of the keyword set of the data item. As keywords are application dependent and the KIOT network is oblivious to them, we need a space efficient and generic way of encoding keyword sets that allows sets to be created, merged and tested for membership. Following TagNet [6], we decided to use *Bloom Filters* (BFs) [7] to represent the keyword sets, as BFs are fixed sized bitmaps supporting quick implementations of all the desired operations.

Initially, all bits of a BF are zero, representing an empty set of keywords. To insert a keyword into a BF, the string representation of the keyword is hashed by a number of hash functions, each of which returns a certain bit position to be set to one. To add a keyword to an existing set, we set to one all the bit positions set to one either in the original BF or in the BF of the new keyword, that is, we perform a bitwise OR between them. The same method (bitwise OR) is used to merge larger sets. To test whether a keyword is part of a set, we create a BF for the keyword and see whether all the bits set to one in that BF are also set to one in the BF of the set. This is implemented by performing a bitwise AND between the two BFs and checking the result for equality with the BF for the keyword. The same method can be used to determine whether a set is a subset of another set.

While the BF approach is a simple way to encode arbitrary keyword sets into a fixed size bitmap, its probabilistic nature makes it prone to false positives, that is, answering that an element is in a set when it is not. This means that the bits set to one corresponding to that keyword may have been set in the BF due to the insertion of other elements. The probability of this happening depends on the size of the BF in bits, the number of hash functions used and the number of elements inserted into the BF; generally, for a specific BF size, the probability grows as more elements are added into the BF, therefore the BFs need to be sized based on the expected number or keywords used by an application so as to ensure a low false positive probability. On the other hand, BFs do not suffer from false negatives, that is, if an element is part of a set, then we will never get the response that its BF is *not* a subset of the BF of the set.

B. Network model

In KIOT, routing always takes place over a single tree, with its root being the border gateway with other networks. All Interests originate from the root and propagate towards the leaves, and all Data items sent in response need to be returned to the root. This means that each node only needs to know which of its links lead to the root in order to return results. To locate data items, each data item is labeled with a BF that indicates the keywords it is associated with, and each node in the tree maintains a FIB entry for each of its downstream links, that is, a BF indicating what content items are available via this link. Each node reports to its parent that it has access to data items represented by a BF produced by merging the BFs of its children and any locally produced data items.

In our implementation, network nodes are divided into three classes:

- Leaf Nodes are IoT devices producing named data via their sensors which are only aware of their parent (specifically, its IP address). The data produced by each sensor are associated with a set of descriptive keywords, which are encoded in a BF called a BF-Keyword. A union of all the BF-Keywords made available by a node form the BF-Catalog, which is a BF describing all the contents of the node. For example, when using BF filters of size 8 bits, a leaf node may have information about temperature (BF-Keyword: 01001000) and humidity (BF-Keyword: 01000001), so its BF-Catalog will have the value 01001001.

- Interior Nodes are network devices used to interconnect their parent node with their children nodes, allowing the root to communicate with the leaves even in large networks. Interior nodes, in addition to the IP address of their parent, contain a FIB with two entries per child: the BF-Catalog of the child and the IP address of the child. Interior nodes merge the BF-Catalogs of all their children to produce their own BF-Catalog, which is propagated to their parent.
- The Root Node is simply an interior node without a parent, serving as the entry point to the network . It should be noted that the root node does not create queries itself, receiving them from an external entity (an application or another gateway node). In addition to a FIB, the root node maintains some state about each query issued, in order to match it against the data returned (see Sec. III-D).

Note that, for simplicity, in our prototype implementation we assumed that sensors are only attached to leaf nodes.

C. Network Configuration

In an actual IoT setting, very large numbers of nodes may exist, therefore it is important for configuration to take place automatically as far as possible, as new nodes enter the system and old nodes go out of service. If the network is wireless, then each node may potentially be in radio reach of many other nodes, creating a mesh of feasible connections. To simplify routing, KIOT overlays a tree on top of that mesh.

Imposing a tree over an arbitrary topology requires each node to automatically find its place in that tree and attach itself to other nodes. To achieve this, our implementation requires that each node added to the network is configured with its level on the tree, which is 1 for the root and $n > 1$ for any other node. This only requires a rough awareness of the structure of the network; for example, in a campus we could have level 1 for the root, level 2 for building gateways, level 3 for floor gateways, level 4 for wing gateways and level 5 for actual IoT devices. In addition, each leaf must be configured with the keywords describing each of its data items, so as to create their BF-Keywords. The rationale is that after setting up a rough outline of the network, new IoT nodes will just need their keywords and level to be added to the network.

Every node entering the system initially tries to locate a parent so as to become part of the tree; later on, other nodes may become its children in the same manner. The configuration protocol relies on the following messages:

- Parent messages: Every node except the root node tries to locate a suitable parent armed only with its level on the tree. Each node sends a broadcast MSG_PARENT message with its IP address and tree level which, in a wireless network, will reach all nodes within transmission range. Each node receiving this message can decide if it is a potential parent for that node, depending on its tree level, which must be one less than the level reported by the new node. Each potential parent responds with an MSG_PARENT_ACK indicating its IP address and its number of children. The new node can then select its parent using one of two policies:
  - Load Balancing: This policy tries to equalize the number of children, thus balancing the tree. In this policy, the new node chooses as a parent the node with the fewest children; if many nodes are tied, one is chosen randomly.
  - Delay Minimization: This policy tries to reduce the delay in answering queries, by minimizing the *Round-Trip Time* (RTT) between nodes. In this policy, the new node measures the RTT between sending and receiving the parent response, and chooses as a parent the node with the lowest RTT.

  Finally, the new node sends an MSG_PARENT_CHILD message to its chosen parent to inform it of its decision.
- Advertisement messages: When a node joins the tree, it starts advertising the data it has available to its parent node, by sending its BF-Catalog to its parent via an MSG_ADVERTISEMENT message, which is acknowledged by an MSG_ADVERTISEMENT_ACK message. The parent updates its FIB by using the BF-Catalog of its child, and updates its own BF-catalog; if it has changed, then its also notifies its parent. Whenever a new leaf node joins the tree, it passes its BF-Catalog to its parent, which leads to recursively updating all FIBs up to the root. Advertisements should be repeated periodically to allow the detection of removed nodes.

D. Queries and Responses

The queries and responses are encoded into Interest and Data messages that are routed from the root towards the appropriate leaf nodes and vice versa, using the routing state presented above. Specifically, we have two types of messages:

- Interest messages: Whenever a query for information is made, expressed as a BF, the root node generates a nonce (a random number or a sequence number) to identify the request and sets a timeout value for the responses matching that nonce. Then, it adds the nonce and the requested BF to an MSG_INTEREST message and propagates it towards all of its children whose BF is a superset of the query's BF, using the BF and IP address of each child in its FIB. The same procedure is repeated by each internal node, until we reach the leaves.
- Data messages: When an Interest reaches a leaf, its BF is checked against the BF-Keyword of each of its data items, and any matching data items (that is, those whose BF-Keyword is a superset of the Interest's BF) are returned as MSG_DATA messages. These messages contain the data item, its BF-Keyword and the nonce from the Interest, and are sent to the leaf's parent node. Each interior node simply propagates these messages unchanged towards the root. The root adds all the Data messages received to a list for the nonce contained in the message, until the nonce's timeout expires. At this point, matching messages are passed to the querier.

Simply put, each query generates an Interest message which is duplicated where appropriate so as to explore the tree, and a set of Data messages to be returned to the root. Since we cannot know in advance how many Data messages we will receive in response to an Interest,[1] our implementation uses a timer to determine for how long we will wait for responses. This allows having many Interests pending at the same time, possibly matching the same data items, therefore our implementation uses the nonce to match each response with the appropriate query.

We note that as routing state is aggregated towards the root, the probability for false matches between the BF in an Interest and a BF in the FIB increases for two reasons: first, false positives may appear, since when too many keywords are inserted into a BF it may match keywords never entered into it; second, nodes higher up in the hierarchy naturally match a wider range of queries. For example, in a building with two floors, temperature sensors only in the first floor and humidity sensors only in the second floor, the BF-Catalog of a node connected to both first and second floor sensors and passed to its parent will include the keywords {`temperature, humidity, floor1, floor2`}, since its children actually do contain all these keywords. As a result, a query for {`temperature, floor2`} will be received by that node, as it matches the FIB entry of its parent, but will not be forwarded further downstream, as it does not match the BFs of any one of its children. Both types of false positives lead to redundant Interest messages being propagated, thus creating a small overhead.

E. Code structure and limitations

The Java implementation of KIOT consists of two basic modules: the KIOT module, which executes at all network nodes and IoT devices, and the Guest module, which represents an application issuing queries or a gateway to an NDN network; the Guest is attached to the root node of the KIOT network. The KIOT module configures itself, finds its position in the network, sets up its routing tables and then handles Interest and Data messages. Only the Guest and the leaf nodes are aware of the actual keywords used, since they need to encode them into BFs; the root and all internal nodes only deal with BFs. The KIOT module currently consists of 1500 lines of code (excluding comments), while the Guest module consists of 650 lines of code (also excluding comments), which are partially shared with the KIOT module. In addition to being highly configurable (e.g., on the size of the Bloom filters used), the code is modular in order to allow extensions, as the currently in progress code execution module.

In order to test large numbers of KIOT nodes without actual hardware, we have built an emulation environment using Mininet[2]. A set of Python scipts allow setting up a network topology with appropriate keywords describing each leaf node, as well as instantiating the network and executing queries on it, gathering up the results produced and various statistics.

In our current implementation, each leaf can only hold a single data item (e.g., sensor) and interior nodes do not hold any data items; these limitations will be removed, since many IoT devices host multiple sensors and can be used both for sensing and data transport. In addition, FIB entries do not time out, which means that dead children nodes are not detected; we will eventually make all FIB entries timeout, requiring periodic advertisements to refresh them, thus ensuring that stale state is removed. Since advertisements are acknowledged, this will also allow nodes to detect that their parent is dead, which can be solved by sending new parent messages. Finally, the timeouts set for queries in order to wait for responses are set statically, ignoring the status of the network; in the future we will introduce adaptive timeouts based on the TCP RTT estimation algorithm.

IV. Evaluation

For our initial evaluation of KIOT, we emulated a campus monitoring system using Mininet to create multiple nodes running our implementation. In this system, different types of sensors are spread around buildings. We defined four categories of tags to characterize each sensor: building (`troias, evelpidon, patision`), wing (`north, east, south, west`), floor (`floor0, floor1, floor2, floor3`) and sensor (`light, temperature, humidity`). Each leaf node contained a sensor that was tagged with a random combination of these four categories, e.g., {`old, north, floor1, temperature`}. We then created a tree by adding to the leaf nodes a root and one or more intermediate levels, and let the system self-organize into a tree: each node selected its parent using the load balancing strategy (see Section III) and the KIOT routing tables were created. Note that the allocation of nodes in space was random, that is, nodes were not grouped in space following their geographic tags (e.g., building, wing and floor).

Our goal was to compare the number of Interest messages sent with KIOT, where Interests are only forwarded by a node to its children matching the BF in the query, against broadcasting an Interest from the root towards all leaves. The rationale for broadcasting is that in a large deployment it is impractical to track all IoT devices; even if we do track the devices, contacting them via unicast requires also tracking the current state of the network, since the topology changes as new nodes are added and old nodes fail. As a result, only broadcast is guaranteed to reach the matching leaves (by, of course, reaching *all* leaves). We did not count the Data messages returned, as these are exactly the same for both schemes: each sensor matching a query will return its Data in unicast mode towards the root, that is, regardless of how many Interests were sent, if n nodes match the query, then n unicast messages will be returned to the root.

For the first experiment, we used a 3-level tree, with a root, 3 intermediate nodes and 9 leaf nodes (average degree of 3). In each repetition of the experiment, we sent 10 queries from the root, with each query containing 1 to 3 tags from different categories (for example, {`troias, temperature`} would gather all temperature readings from the Troias building). Each query returned 1 to 4 different results, depending on the sensors matching the tags in the query, for a total of 21 results across all 10 queries. We repeated the experiment 10

[1]This would require tracking all available IoT devices, which is impractical in large networks.

[2]http://mininet.org/

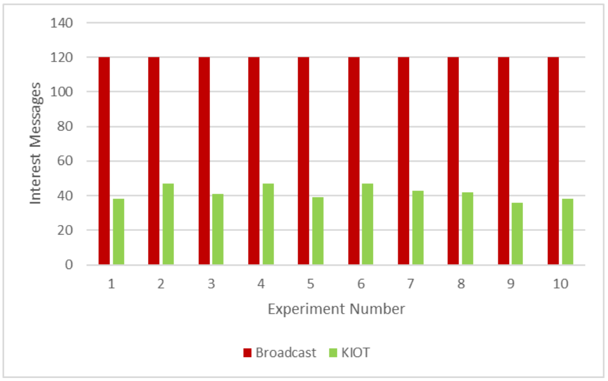

Fig. 1. Interest messages sent with Broadcast and KIOT in a 3-level tree.

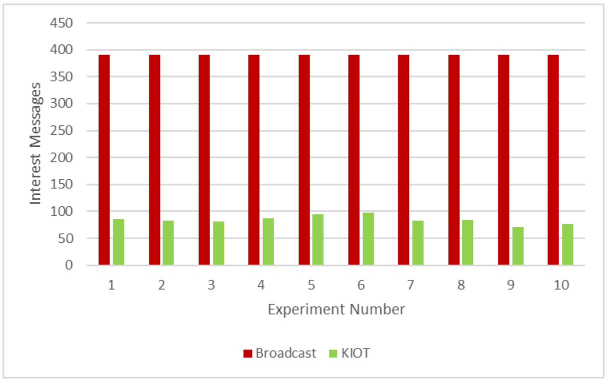

Fig. 2. Interest messages sent with Broadcast and KIOT in a 4-level tree.

times, using in each run the same leaf nodes and the same queries, but letting the tree self-organize in a different way each time. Figure 1 shows the number of Interests sent in each repetition of this experiment. With broadcast, each Interest must be propagated from the root over all links in the three, therefore we need 12 messages per query and 120 messages per experiment, regardless of the tree (an n-node tree will always have $n-1$ links). With KIOT we only forward Interests based on matching BFs, therefore the exact number of Interests also depends on how the leafs are organized in each run. The average number of Interests was 41.8 for the 10 queries, or 4.18 per query, on average.

For the second experiment, we used a 4-level tree, again with an average degree of 3 (root, 3 level 2 nodes, 9 level 3 nodes and 27 leaf nodes). We again sent 10 random queries per experiment, this time with 2 to 4 tags from different categories. Each query returned 1 to 3 different results for a total of 16 results across all 10 queries. Figure 2 shows the number of Interests sent in each repetition of this experiment. With the 4-level tree, we have 39 links, therefore with broadcast we need 39 messages per query and 390 messages per experiment. With KIOT, the average number of Interests was 84 for the 10 queries, or 8.4 per query, on average.

It should be clear that KIOT offers large savings compared to broadcast during the query propagation stage, since in the 3-level treee we need less than 35% of the messages sent with broadcast, while for the 4-level tree we need less than 22% of the messages sent with broadcast. In a more realistic network, with nodes placed in space based on their geographic tags, KIOT would be even more beneficial, as queries would be sent to smaller parts of the tree. In general, the gains grow with larger trees and more specific queries, that is, when KIOT only needs to propagate Interests to small subsets of a tree. Regarding other costs, setting up the routing tables requires two messages per link (equal to two broadcast queries), paid once when the system comes up; whenever a leaf is added, removed or modified, we need 2 messages per affected link, possibly up to the root, that is, up to 4 messages for a 3-level tree and up to 6 messages for a 4-level tree.

V. Conclusion and Future work

We have presented an implementation of a keyword-based scheme for gathering information from IoT systems (KIOT) that uses ICN principles, enabling a WoT layer to be created from simpler IoT devices. KIOT uses a highly flexible scheme for describing arbitrary sets of data which allows powerful applications to be built in a very natural manner, without customizing the network for each potential application or requiring complex network configuration. We explained how our general design was mapped to an actual implementation that can auto-configure itself over an arbitrary network with unknown IoT devices. Our implementation is simple and has a very small footprint, but it is constantly being extended with additional features for robustness and flexibility.

Our current work focuses on testing our implementation in a large scale using more realistic networks emulated with Mininet, so as to measure its performance and fine tune its policies and parameters; issues we are working on is balancing the tree, minimizing response times, adapting the timeout values to balance responsiveness and completeness of results, allowing nodes to operate both as data sources and as network relays and repairing the tree and routing tables as nodes join and leave the network.

Our future work will add function execution in the network; we have already implemented a small set of functions to be executed at the root (e.g., SUM, AVERAGE, etc.), but we plan to allow arbitrary function execution at any capable node in the network, following ideas in [8] for function placement. Further work will focus on selecting function execution points by jointly taking into account the costs for moving the function code and the data items to the same node for processing.

Acknowledgments

This research was supported by the RC-AUEB funded *Original Scientific Publications* project under contract ER-3013-01.

References

[1] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos, "A survey of information-centric networking research," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 1024–1049, Second 2014.

[2] V. Jacobson, D. Smetters, J. Thornton, M. Plass, N. Briggs, and R. Braynard, "Networking named content," in *Proc. of the ACM International Conference on Emerging Networking Experiments and Technologies (CoNext)*, 2009, pp. 1–12.

[3] O. Ascigil, S. R. né, G. Xylomenos, I. Psaras, and G. Pavlou, "A keyword-based ICN-IoT platform," in *Proc. of the ACM Conference on Information-Centric Networking (ICN)*, 2017, pp. 22–28.

[4] G. Xylomenos, G. Pavlou, I. Psaras, and I. Karakonstantis, "Named functions at the edge," in *Proc. of the IEEE International Symposium on Computers and Communication (ISCC)*, 2019.

[5] M. Amadeo, C. Campolo, and . Molinaro, "Multi-source data retrieval in IoT via named data networking," in *Proc. of the ACM Conference on Information-Centric Networking (ICN)*, 2014, pp. 67–76.

[6] M. Papalini, A. Carzaniga, K. Khazaei, and A. Wolf, "Scalable routing for tag-based information-centric networking," in *Proc. of the ACM Conference on Information-Centric Networking (ICN)*, 2014, pp. 17–26.

[7] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of Bloom filters for distributed systems," *IEEE Communications Surveys Tutorials*, vol. 14, no. 1, pp. 131–155, First 2012.

[8] M. Król and I. Psaras, "NFaaS: Named function as a service," in *Proc. of the ACM Conference on Information-Centric Networking (ICN)*, 2017, pp. 134–144.