NDNSD: Service Publishing and Discovery in NDN

Saurab Dulal

NDNSD: SERVICE PUBLISHING AND DISCOVERY IN NDN

by

Saurab Dulal

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Major: Computer Science

The University of Memphis

December 2020

Abstract

Service discovery is one of the crucial components of modern applications. With the advent of several new systems such as IoT, edge, cloud, etc the world is connected more than ever and smart devices are creeping towards every nook and corner of our surroundings. Not only the new systems are emerging but also the communication pattern is evolving i.e. from one-to-one (host-host) to many-to-many (distributed application, IoT). The definition of service has also changed over time. Unlike their meaning in the past as programs running on some machines, services today can be sensor devices collecting data, mobile devices offering computing service, or it can even be a piece of data generated by some system. To satisfy the changing dynamics and heterogeneity of the services and the demand of these evolving architectures several new protocols are developed on top of the TCP/IP stack. Nonetheless, the fundamental weakness of host-centric TCP/IP to support the need for distributed application (IoT, edge) and many-to-many communication (e.g. publisher-subscriber) have induced several weaknesses in the system and have made it more fragile. Named Data Networking (NDN) is an information-centric networking architecture that does the communication over signed, named content objects. Its pub-sub style of communication, data-centric security at the network layer, in-network caching, etc provides numerous benefits to modern systems and tries to overcome the shortcoming of TCP/IP.

In this thesis, we propose NDNSD – a fully distributed, scalable, and general-purpose, service discovery protocol for information-centric architecture/NDN. It is developed on top of the synchronization protocol (sync) and offers publisher-subscriber API for service publishing and discovery. We present several design features of NDNSD and also establish how it is best suited for modern systems. We also introduce the concept of service-info and how it can be combined with sync and NDN hierarchical names to make service discovery generic. Finally, To substantiate our argument, we design, implement, and evaluate our protocol, and also provide some use-cases (e.g. Building Management System) to show how service discovery can be beneficial.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The world of networks and communications is rapidly changing. It has been projected [1] that by 2023 network devices will inflate to about 29.3 billion, Machine-to-Machine (M2M) connections will reach about 14.7 billion shared among moving cars and IoT applications, there will be enormous growth in mobility, and 5G will have its fair share [1]. These devices will comprise IoT devices such as wireless sensors, actuators, smart home applications, edge computing applications, autonomous systems, and so on, and will exist in almost every nook and corner of our surroundings. So far, TCP/IP has done a wonderful job of maintaining these connections and have kept things moving. But several questions are raised [2] whether the architecture that was envisioned for point-to-point communication will be capable of handling stringent new requirements posed by this plethora of connectivity or not. Nonetheless, the Internet is evolving. It has pushed itself beyond the traditional point-to-point communication mode and has allowed people (researchers, industry, and service providers) to explore several new dimensions of the Internet such as CDN, ICN, and so on. Named data networking (NDN) [3][4], a flavor of Information-Centric Networking (ICN), envisioned as the future of Internet architecture is evolving rapidly and has brought several promises for the present and futuristic Internet. Naming the content rather than the host, securing packets right at the origin, intrinsic mobility support, pervasive caching, publisher-subscriber style of communication, direct use of application-layer names into the network layer are some of the much-needed flavors offered by NDN to support the next billions of devices envisioned above. More about NDN is presented in the section 2.0.1 below.

Service discovery (SD) is a fundamental requirement of contemporary networking systems. Modern web, IoT application, building management, smart device, LAN, WAN, mobile application, etc depends on SD in a way or another.

Furthermore, the proliferation of the applications towards the edge, rapid expansion and advancement of IoT and sensor networks, and cloud computing have pushed the service discovery to the next level of importance. It is also changing the dynamics of how the services and resources were discovered and used in the past. Additionally, the pervasive nature of wireless networks and mobile devices has underscored its importance and is expected to be even more in the future. The most common use of SD can be seen in local area networks such as discovering printers, bluetooth devices, network computers, gateways, access points, service objects, and so on.

Plenty of research has been done in this area and several solutions are proposed for IP and non-IP based i.e. Peer-to-Peer (P2P), data-centric systems. Meshkova, Elena, et al [5] have categories service/resource discovery based on the scale – local, enterprise, Internet, type – wireless, IoT, edge, architecture – client-server, P2P, data-centric, centralized, decentralized, etc. based on the network and platform they are used. However, we found several limitations in these solutions such as requirement for a centralized or cloud servers, IP to name mapping systems etc. In the following sections we will discuss more about the limitations and our proposed solution.

1.1 Limitations of TCP/IP in Service Discovery

The Internet is constantly evolving along with its original point-to-point client/server style of communication. Modern distributed systems such as IoT, cloud computing, CDNs are inherently data-centric and the communication pattern is more complex and diverse (one-to-many, any-to-any). These data-centric systems often require a service/resource discovery mechanism to invoke services offered by the applications. Protocols such as UPnP [6], Bonjour [7], Link Layer Service Discovery (LLSD) [8], Service Location Protocol (SLP) [9], Jini [10], Service Discovery Protocol (SDP) [11], etc exist in different layers of TCP/IP stack that facilitates the discovery process. Most of these solutions depend on DNS [12], multicast-DNS (mDNS) [13] or DNS-based Service Discovery (DNS-SD) [13]. However, these protocols have several limitations. Shang, Wentao, et al. have discussed them in detail in their papers [14][15]. Regardless of the limitations in these protocols, the most important one comes from the network layer failing to directly identify names injected by the application layer. For example, a printer application cannot just advertise itself as "PrinterA" to the network but rather its identification is tied with IP address and port number, because that's what the network layer recognizes. This creates a semantic mismatch between the network layer and application layer, and to resolve it (name to IP address mapping), we need resolution service such as DNS. And this is how most of the discovery is working for decades.

With the advent of modern systems such as IoT and Edge, the service discover has reached to a new horizon. For example, DNS-SD based discover mechanism usually identifies service by its name and port number – defined in SRV records, and mostly assumes the service as a running process in some machine. However, the meaning of services in IoT is can be different. It can be a sensor collecting the data or data itself. This heterogeneity in service needs to be dealt with more general discovery approach. Some protocols such as Constrained Application Protocol (CoAP) [16] identify services by their names instead of DNS records. It uses a dedicated server, resource directory (RD), to store meta info such as name, IP, port, and other details of the services running elsewhere. CoRE-RD [17] is one such protocol that uses RD and is based on CoAP. Nonetheless, infrastructure, as well as a centralized server (RD), is required for the protocol to work and this can be cumbersome for inherently decentralized applications. Oh the other hand, mDNS offers a decentralized solution for service discovery and works without infrastructure support. It is widely adopted by several existing discovery protocols and provides a basis for Zero Configuration Networking [18] in the local subnet. With mDNS, Devices register themselves with a link-local IP Address (self-assign an address and proves using ARP [19] request to check uniqueness) and a name (e.g. *_device-info._tcp****.local***).“.local” is a top-level domain (TLD) attached to the hostnames that are supposed to work locally. All the mDNS queries (for names ending with ”.local”) are sent to a reserved multicast address and devices/resources discovery withing the subnet becomes very easy. However, it only works with link-local addresses and thus won’t be very useful for IoT mesh or multi-hop environment. Similarly, edge applications such as Smart Things [20], AWS IoT [21], Google Chromecast [22], Azure Sphere [23] mostly depend on cloud for service registration and discovery. This dependency incurs few notable problems i) the resources that are purely used in the local environment have to rely on the cloud for service registration ii) extra round-trip overhead even for the devices residing in close vicinity and iii) disruption in the infrastructure connectivity will disrupt the whole service.

Hence, taking into consideration of evolving data-centric system and their demand for efficient, seamless, and scalable discovery mechanism, we believe, the core functionalities needed for the discovery should be embedded in the architecture itself. They should be exposed to application developers to implement their custom logic on top of it rather than depending on external protocols or components. The capability of NDN to use semantically meaningful application names in the network layer simplifies most of these problems. Addressing information-centric services by names and supporting multi-party communication on top makes the discovery process even simpler. Thus, we choose NDN for this project. And in the rest of the sections, we will only present NDN specific discussion.

1.2 Problem Statement

Service Discovery (SD) can be defined as a process of identifying distinct content or services that are offered by applications or by some well-defined network services in the desired network. It should be capable of i) advertising and revoking services ii) discover services offered by other applications and iii) optionally select and invoke desired services.

During our past projects on building management systems [24], sensor networks, and work from our colleagues on distributed mobile applications [25], we realised the need for a better service discovery mechanism and actively looked into some of the existing works (section 2.0.3). We mostly relied on manual configuration but the complexity grow significantly as the network size grows. Looking into the existing work, we found several limitations and issues. Major of which are listed below:

- Limited to a specific environment or domain e.g. IoT, edge, etc.
- Either centralized or non-scalable solution
- Applications forced to deal with lower-level network primitives

Note: More on these problems are explained in the related work (section 2).

1.3 Thesis Contributions

This thesis design and develops a fully distributed, general-purpose, salable service discovery protocol for NDN by leveraging the publisher-subscriber feature offered by the NDN Synchronization protocol. The protocol is developed on top of current NDN stack and sync protocols (PSync [26], ChronoSync [27]) and offers several features such as high-level API for service publishing and discovery, hierarchical namespace, measurement information, and so on. The thesis also presents a strong use-case of service discovery in the Building Management System and its implementation in Mini-NDN [28]. Finally, it also provides several evaluations that were performed in the real-world and emulation environments.

The rest of the thesis is organized as follows. In Chapter 2, we provide a background of NDN Forwarder, Data Synchronization Protocols, and discuss some of the works related to our project. NDNSD protocol overview, design feature, and the use-cases are discussed in Chapter 3. In Chapter 4, first, we provide a brief description of Chronosync and PSync. Next, we discuss the implementation details of NDNSD. We also discuss several components of NDNSD such as State, Service-Info Representation, Data Packet Specification, and more. In Chapter 5, we evaluate the protocol using real-world and emulation experiments. And finally, we conclude our work in Chapter 6 and also discuss possible future directions.

Chapter 2
Background and Related Work

In this section we will talk about the NDN Forwarder, sync and some of the service discovery mechanisms in NDN related to our work.

2.0.1 NDN Forwarder

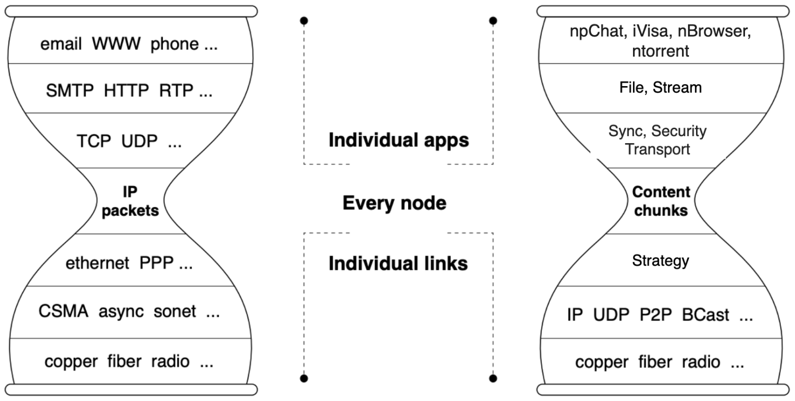


Figure 1 NDN Narrow Waist

Named data networking is an evolving data-centric internet architecture that fundamentally differs with TCP/IP in terms of communication model. Every single piece of content in NDN is named, and networking is done over this named content chunks by using interest and data packets. It completely eliminates the host-destination style of communication by decoupling the packets from the producers location. Once decoupled, they can be served by any intermediate nodes whole stores a copy of it. Authenticity of a packet is ensured by signing it at the time of creation. Signing is done by producer using NDN security schemes. A simple NDN hour-glass model is shown in the right side of figure 1 Forwarder (NFD) is a core module of NDN and implements major Named Data Networking protocol stack needed for named based communication. Network interfaces and application end points as are abstracted as a *Face* on top of several lower-level transport services. NFD uses these *Faces* to receive and forward interests and data packets. Figure 2 shows the latest version – at the time of this thesis – of NDN Packet Format Specification version 0.3 [29]

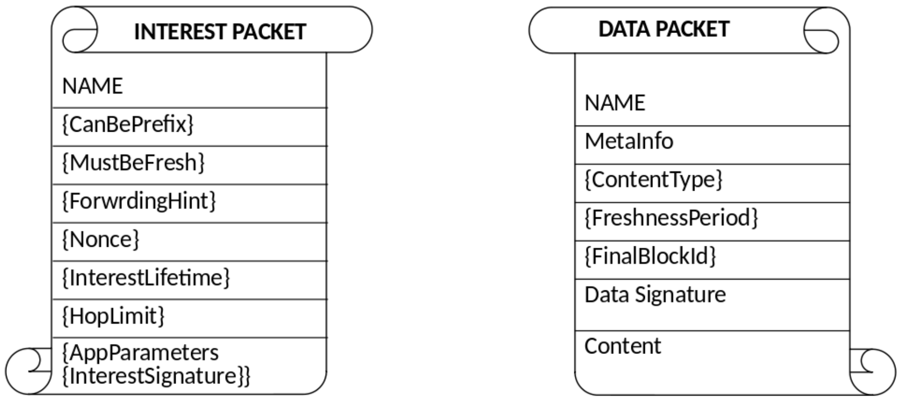


Figure 2 Interest and Data packets in NDN

An overview of the packet forwarding pipeline in the NFD is shown in Figure 3 First, when the interest arrives at the forwarder, it is checked whether it's a looped interest or not and is dropped if found duplicate. Next, an entry is created in the Pending Interest Table (PIT), the interface from where the interest is received is recorded, and the expiry time is set, which is equivalent to interest lifetime. After creating the PIT entry, the content store (CS) is checked to see if the data corresponding to the interest is already available there. CS caches every valid data packet received by the forwarder based on CS policy. If found in CS (cache hit), data is sent back via the same interface the interest was received, else (cache miss) the interest is handed to the strategy which consults Forwarding Information Base (FIB) to figure out appropriate face to the forward the interest. Meanwhile, if another interest arrives for the same data before satisfying the previous one, or if the PIT entry already exists for the interest, the incoming interface is recorded and the lifetime is reset to the new value. This process is called interest aggregation and is a very powerful in-build mechanism for congestion control and loop detection in NDN. Finally, once the data comes back to the forwarder, it is sent to all the recorded interfaces and the PIT is consumed.

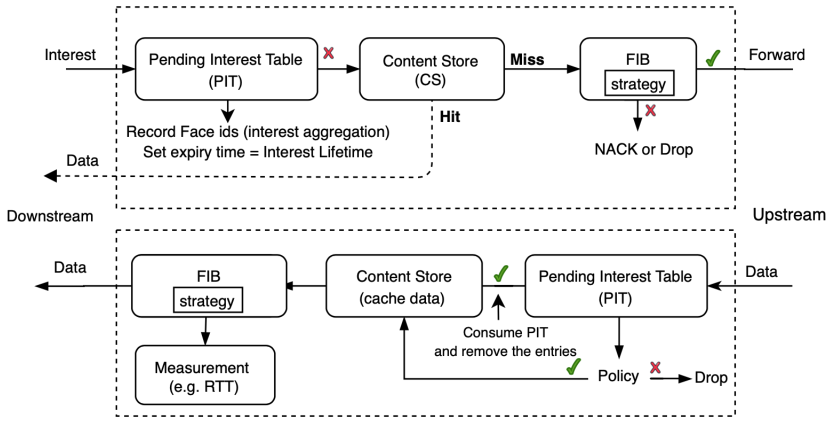
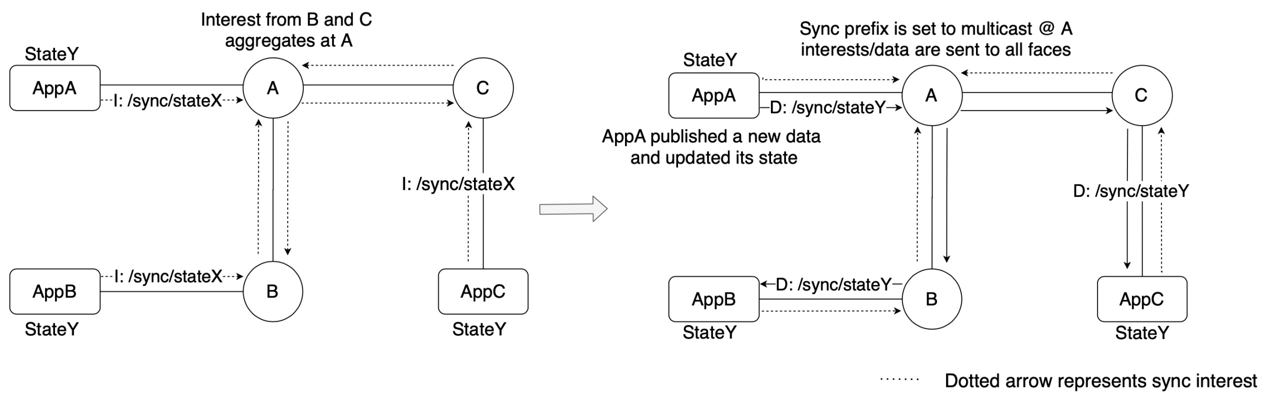

Interest
Pending Interest Table (PIT)
Content Store (CS)
Miss
FIB
strategy
Forward
Hit
Record Face ids (interest aggregation)
Set expiry time = Interest Lifetime
Data
NACK or Drop
Downstream
Upstream
Data
FIB
strategy
Content Store (cache data)
Pending Interest Table (PIT)
Data
Consume PIT and remove the entries
Measurement (e.g. RTT)
Policy
Drop

Figure 3 NDN Forwarding Pipeline Overview.

IP forwarding is done solely based on FIB, and since the data packet can follow any path back to the client, the forwarding module has asymmetric view of the network. Thus, the it cannot record packet performance through the network. NDN forwarding is stateful, every router maintains the state of the forwarding process in the network. The data packet in NDN follows the reverse path as that of interest, so the routers can compute packet processing performances such as RTT, throughput, etc. These measurements can be used by forwarding strategies [30] to make an adaptive decision. Thus, interest received by forwarder is handed to the strategy rather than forwarding it solely based on FIB. The strategy can decide appropriate action for the interest received based on the previous measurement and the FIB.

2.0.2 Data Synchronization In NDN

Communication patterns in modern distributed applications such as IoT, edge and cloud computing, vehicular network, monitoring system, social media, etc are much more sophisticated than just a two-party source-destination model. It can be one-to-many, many-to-many, and any-to-any. These applications are essentially data-enteric but the way it works in today's IP-based system is through some centralized mechanism because of IP's structural limitation to handle any to any


Interest from B and C aggregates at A
StateY
AppA
I: /sync/stateX
A
C
I: /sync/stateX
AppB
I: /sync/stateX
B
AppC
StateY
StateY
Sync prefix is set to multicast @ A interests/data are sent to all faces
StateY
AppA
D: /sync/stateY
A
C
AppA published a new data and updated its state
D: /sync/stateY
AppB
D: /sync/stateY
B
AppC
StateY
StateY
Dotted arrow represents sync interest

(a) Nodes in Same State

(b) Nodes updates to new state with publication at A

Figure 4 NDN Synchronization Overview. "I" and "D" represents sync interest and sync data respectively

communication style. NDN communicating style at the network layer is inherently distributed. The data are not coupled with a host but rather can reside anywhere in the network. Its provenience can be ensured anytime anywhere – thanks to NDN in-build security. NDN synchronization – *sync* in short – provides a powerful abstraction above the interest-data exchange to facilitate multi-party communication by synchronizing the state or data among the participating nodes (sync nodes). An application implementing the sync will not have to worry about how the data or state gets to another node. Once the data is published to the sync, it will take care of synchronization by maintaining a shared distributed dataset containing the latest info, and by propagating the updates to other nodes. Additionally, NDN's special feature of securely binding names with the content makes things, even more, simpler with sync. Dataset maintained by *sync* contains names and their latest state, and the applications, upon requirement, can fetch the corresponding data using names delivered to it by the sync. Several NDN synchronization protocols such as ChronoSync[27], RoundSync[31], VectorSync[32], NDN Sync[33], iSync[34], PSync[26], etc are proposed so far and they provide several unique techniques for data-set synchronization. For our work, we choose PSync and Chronosync as they are openly available as a library, actively maintained, and also are currently used in the NDN testbed. We have presented more details on PSync and Chronosync in section 4.

Figure 4 shows a naive syncronization process in NDN. Figure 4a shows that first, all the applications are at the same state i.e. stateX and there is a sync interest from each application to another in the network. These sync interest are long-lived interests and resides in the network PIT. It's primary purpose is to fetch sync data when published by any node in the network. Next, in figure 4b, AppA publishes new data to the sync and updates its sync state to stateY. All the pending sync interest (from AppC and AppB) at A gets answered, and the new state i.e. stateY is delivered to AppC and AppB. Once they receive the data packet, they will perform set reconciliation and will update their state to the latest i.e. stateY.

Thus, to design a distributed service discovery protocol where metadata data (service-info) synchronization is one of the core requirement, sync makes everything much simpler by offering both multi-party communication and dataset synchronization.

2.0.3 Related Works

An early attempt to use sync protocol for service discovery was proposed by Mark Mosko for CCNx 1.0 [35]. Devices use a well-defined namespace such as *"/parc/printers/"* to advertise the manifest of service details which include TTL as a lifetime. Ravindran, Ravishankar, et al [36] proposed two different service discovery protocols, neighbor discovery protocol (NDP) for locally reachable CCNx nodes neighbors and Service Publish and Discovery (SPDP) for publishing local services and discovering remote services. SPDP used a recursive query that propagates hop-by-hop among the reachable adjacencies running SPDP instances. The data containing the service list is aggregated by the respective instances and is sent back to the original requestor. The solution is very similar to brute-force, services are searched in every hop successively, thus scalability will be a concern in a larger network. Also, if services are very far (multiple-hops away) the discovery process may take a longer time and can be infeasible for frequently updating services e.g. computing services. [37] [38] uses broadcast mechanisms for the service discovery in IoT and edge environments respectively. In [38] utilizes expanded ring search technique along with the broadcast for service discovery. Consumer requests for desired services with attributes such as position, speed, direction, content to be processed, etc. The request is first broadcast to a 1-hop neighbor, if no reply is received within the pre-defined timeout, the request is again sent to 2-hop neighbors, TTL is used for hop count. The process repeats until the service is found or the consumer gives up. However, the solution is proposed for edge application only, also expanded ring search can be expensive and might not scale if there are huge numbers of consumers. Mtibaa, Abderrahmen, et al [39] identified a combination of three mechanisms: proactive, reactive, and passive resources discovery in the edge. Edge compute nodes (ENs) will proactively advertise their resources in the proactive mode which creates FIB entry for others to query the resource. Mobile devices looking for services can query ENs. The reply will update the FIB entry of nodes in the reverse path. And in passive mode, NACK is sent if in case ENs are overloaded to restrict downstream and send more requests. A very close approach to our work is presented in [40][41]. Authors use a similar concept of synchronization meta-info of services, using sync protocol, among multiple edge computing servers (ECS) to facilitate service discovery. ECS are special nodes in the network that synchronize and maintains service information available from the provides. A discovery interest (e.g. */discovery/ecs/*) from an application is replied with a suitable service by the closes ECS. Nevertheless, Maintaining ECS can be an extra infrastructure for service discovery. This can be relevant for a particular edge computing application but not in general.

Chapter 3

Design Overview

In this chapter, first, we introduce the rationale of sync for service discovery. Next, we provide details of NDNSD protocol (Section 3.1). Then, we discuss namespace design for NDNSD (Section 3.2) and several other design features. Finally, we discuss a strong case for the use of NDNSD in the modern Building Management System (BMS).

3.1 Introduction

Synchronization protocol plays a crucial role in NDN. The original NDN architecture [42] envisioned combining sync protocol with application accessible libraries to provide transport functionalities to the applications. The application accessible libraries should hide the core network functionalities and primitives such as interest and data from the applications and the sync should help the transfer data from one application to another. The Internet protocol stack [43] is a well-known example of such a model, best known as the hourglass model. Hundreds of new protocols and several changes are introduced in the top and bottom layers, but the core is kept simple and intact. The success of such models has been argued by the pioneers [44][45] time again. Remaining in the realm of the internet hourglass model and as envisioned by original NDN architecture, we have developed NDNSD, a fully distributed general purpose[1] service discovery protocol for NDN, as well as an application accessible reusable component that uses synchronization protocol for service announcement and discovery.

We view service discovery problems as data synchronization problems, some applications actively look to discover services while others are trying to advertise the services. If we further think about it, the whole process boils down to a pub-sub system, i.e. publishing and subscribing services. At a high level, this can be

[1]Covering a wide range of environments e.g., LAN, WAN, IoT, edge, mobile, etc.

compared with topic-based communication in MQTT [46]. Communication models for SD in this pub-sub system can be of three types i) nodes advertising services via publishing, e.g., printers ii) nodes only discovering services via subscribing, e.g., sensors and devices ii) and nodes doing both, e.g., mobile phones [e.g., nPchat] and laptops. There are several benefits of using sync for SD, such as i) no external dependencies or demanding change in network layer – sync comes with NDN natively, ii) it inherently supports multi-way communications, iii) flexibility to implement application semantic. Two or more parties can agree on a common sync group. This is applicable to local as well as global applications or devices. A mobile application can agree on a sync group *"/letschat"*, whereas printer services can agree on *"/printers"*. Similarly, IoT and edge applications can have their own sync group. Unlike the limitation of TCP/IP, i.e. the required for name resolution services such as DNS or mDNS as discussed above, these names (sync group), injected by the application layer, can be directly used in the network layer. This gives huge flexibility to applications to implement their own semantics and reflective names that can identify their services. Thus, the core concept of NDNSD is to group the services by their service-type and synchronize the metadata (section 3.4.1) of the services using sync protocol. Let us first define some of the terms that will be frequently used in the coming sections.

- **Service Type** Unique identifier for the type of service offered by an application. This can be existing standard services, e.g., printers or an entirely new one. It is like DNS Service Type [47].
- **Service Name** Application name used by the service provider to advertise their service information, e.g., /uofm/printer1/NDNSD/service-info. These names should be routable in the network. Here, *printer1* is a unique identifier associated with the service and is also known as a service identifier.

- **Service Details** Metadata that contains specific details about the service (more on section 3.4.1) .
- **Publish** Advertising service to the network.
- **Update** Receiving service updates published by others in the network. Note: The term "publish" and "update" in a sync level means publishing data to the sync and receiving updates via sync.

In the next sections, we go into detail of namespace design, NDNSD protocol overview, and design specifications.

3.2 Hierarchical Namespace

One of the major motivations for choosing NDN instead of TCP/IP for service discovery is because of its strength to process application names directly at the network layer, i.e., the packet level. As discussed in the limitation of TCP/IP (section 1.1), TCP/IP not being able to do so forces it to depend on mapping services such as DNS and mDNS for the name resolution. This creates an extra burden and incurs several security challenges [14].

Naming is one of the core benefits of NDN. Application data objects are represented using immutable names, and more importantly, these names are directly used at the network layer. An application can say, "I am printer" instead of saying "I am 192.168.0.71" to the network layer. This provides huge flexibility to the applications in choosing semantically meaningful names. Once these names are received by the other applications, [2] it will be very simple to identify the type of service associated with the name. However, the namespace needs to be designed carefully because they are directly used for routing and forwarding; and a bad design can lead to bad performance and scalability issues.

[2]Most of the communication in today's world is application-driven i.e. application to application

NDNSD offers hierarchical namespace design (figure 3.2) for discovery and application data prefixes which we call “discovery” and “application service” prefix, respectively.

3.2.1 Discovery Prefix

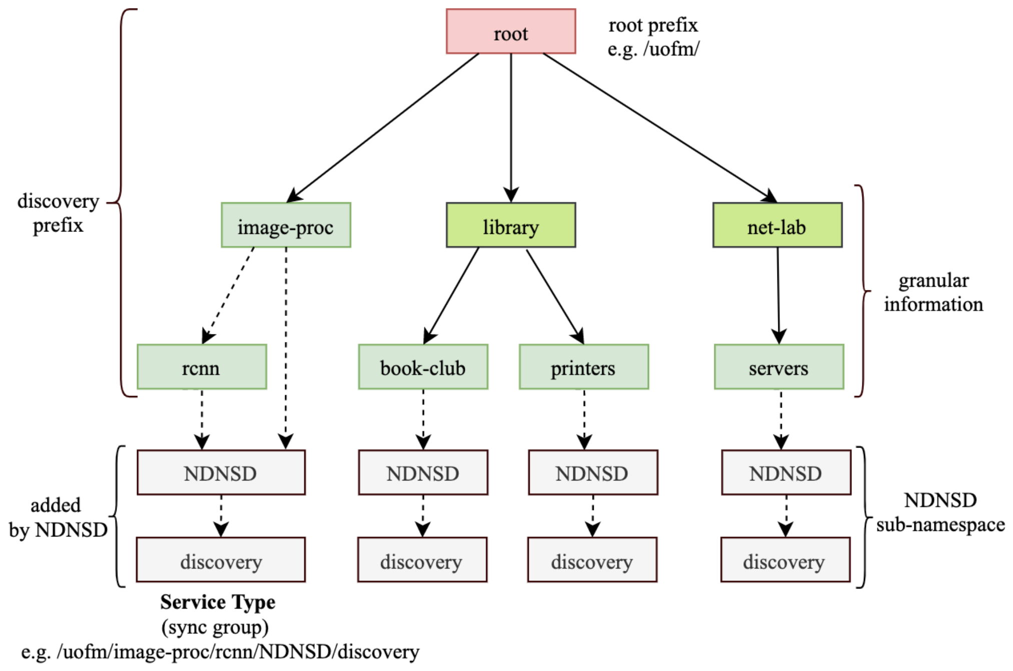


Figure 5 NDNSD Discovery Namespace Design

Discovery prefix, also known as service-group, is constructed in a way to reflect the semantics of the services it groups. It has a root on the top, which is also a trust anchor [48]. Next, the service granularity is added as per the need to make the service type more meaningful. For example, the prefix can be */uofm/image-proc* or */uofm/image-proc/rccn*. The first one contains all the image processing services available to this group regardless of the type of algorithm used, but the second one contains specifically those who run RCNN (Regions with Convolution Neural Networks). Similarly, logical grouping of people (e.g., net-lab) or location-based (e.g., printer) information can also be added to the prefix to make it more sensible. Finally, two extra components, “NDNSD” and “discovery”, are added by NDNSD.

The first component is to distinguish the packet generated by NDNSD, and the second one is to recognize it as a discovery packet. One of the examples in the figure 5 has a root, */uofm*, a trust anchor, , and an organization name in this context, as well as service-type, *image-proc*, specifically RCNN. So, the final discovery-prefix will be */uofm/discovery/image-proc/rccn/* whereas the discovery data name (a.k.a sync prefix) will be */uofm/discovery/image-proc/rccn/NDNSD/discovery*.

The discovery prefix design is very flexible because it has no strict limitation on the levels of hierarchy and service-type name, meaning applications can decide how many components to attach to the prefix. The only strict requirement is that the application should have a valid certificate for the prefix it wants to use to advertise the service. NDN requires every data packet to be signed by the certificate issued for the name.

Additionally, publishers can also join the existing pool of service-type (already known) or create a new one on the fly. For example, a gaming application can create a new group of service-type */memphis/cs/gamer-001* or even choose a longer semantically meaning full name and let others join the group. Note: NDNSD assumes that the service-type is known to the locator application before evoking the discovery request.

3.2.2 Application Service Prefix

Application service prefix is constructed by adding a service-identifier (service-id) to the discovery prefix. For example, with a discovery prefix */<root>/printers* and service identifier *printer1*, the application prefix will be */<root>/printers/printer1*. Similar to discovery names, "NDNSD" and "service-info" are added to the application prefix to obtain the service-info data name. These components help avoid name collision and make it easier to identify and authenticate data generated by NDNSD. Figure 6 shows an example of a basic application prefix hierarchy. It can be coarse such as */uofm/printers/printer1* or more granular (e.g., containing

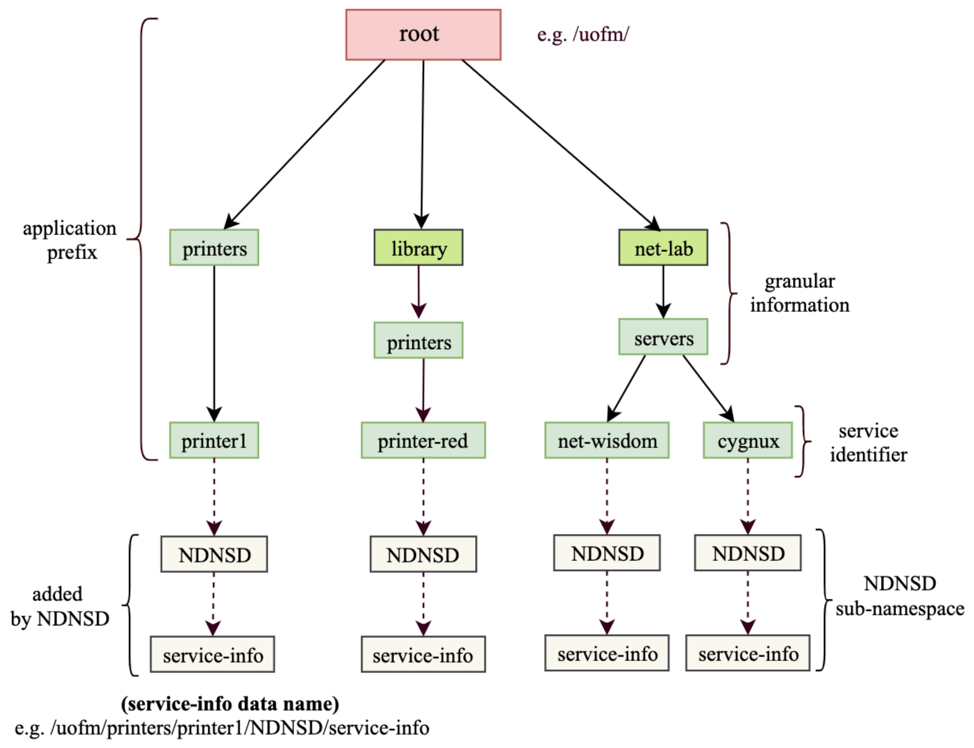


Figure 6 NDNSD Application Service Namespace Desing

location information) such as */uofm/net-lab/servers/cygnux* or any other relevant names.

As shown in the figure 6, sometimes there can be multiple discovery prefixes *(/uofm/printers, /uofm/library/printers)* for the same type of service. What prefix should an application choose if it is authorized to advertise its service under more than one? In this case, it is up to the applications to decide under which domain they advertise the service. One obvious factor could be the semantics of the service. If a printer is in the library, */uofm/library/printers* can be a meaningful choice. One important thing to note is that NDNSD doesn't have any strict requirements for application service names, meaning, NDNSD can work with any other names; "discovery prefix + service-id (/<discovery-prefix>/<service-id>)" is not a strict requirement because these names are carried by sync and are handed to the service finder upon request. The finder does not need to know anything about them, but the names should be valid and routable. Thus, it is entirely a developer's choice of how they want to implement the protocol.

Example NDNSD Namespace

Table 1 NDNSD prefixes and data names example

Organization prefix:	/uofm
Discovery Prefix:	/uofm/netlab/servers
Discovery Data Name (sync prefix):	/uofm/netlab/servers/NDNSD/discovery
Service identifier:	cygnx
Application Service Prefix:	/<discovery-prefix>/<service-id> e.g., /uofm/netlab/servers/cygnux — this should be routable
Service-info data name:	/uofm/netlab/servers/cygnux/NDNSD/service-info — also known as application service data name

To advertise a service, a user just needs to supply a discovery prefix (eg /uofm/netlab/servers) and the service identifier (eg cygnux), along with other service details. To find the service, the user needs to supply a discovery prefix.

3.3 Protocol Overview

NDNSD provides two higher-level interfaces, i.e. Service Publisher (SP) and Service Finder (SF) to facilitate the publishing, updating, and discovery processes. Let us now go into the detail of each of these interfaces.

3.3.1 Service Publisher

- Receive and store service advertisement and updates from the user. Each advertisement (service info, section 3.4.1) must contain a type (e.g., /<prefix>/printers, /<prefix>/sensors), name, and details of a service.

- Construct an appropriate sync group prefix using the service-type received from the service publisher and join the corresponding sync group if it already exists (or is known) or create a new one if it's not. For example, the sync group name will be "/<prefix>/printers/NDNSD/discovery" for the service type "/<prefix>/printers".

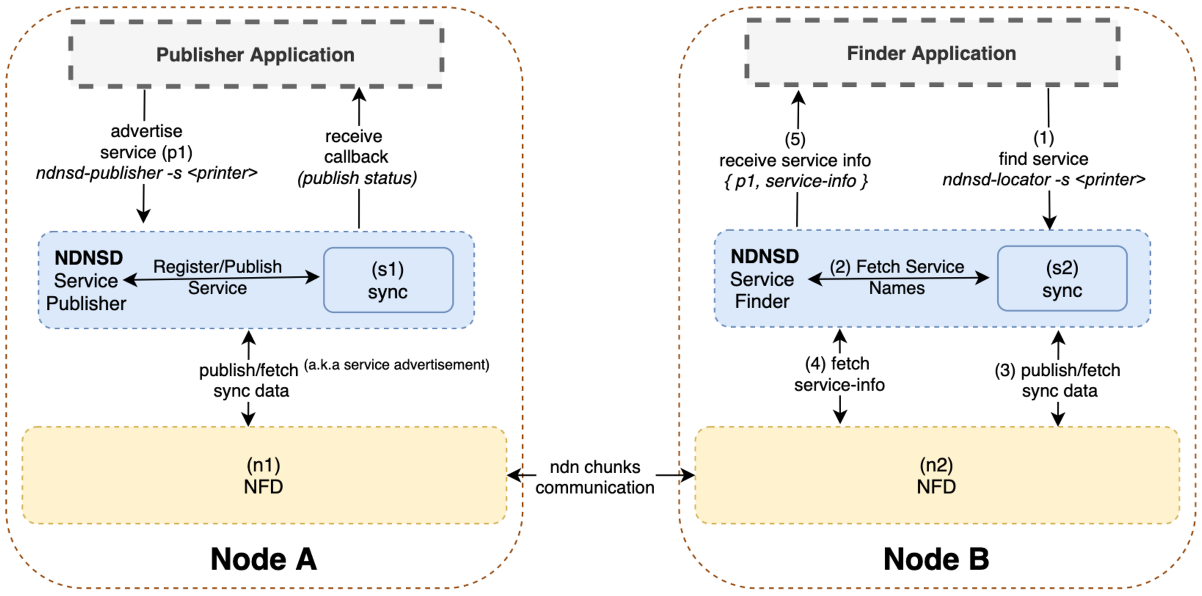

Publisher Application
advertise service (p1) ndndsd-publisher -s <printer>
receive callback (publish status)
NDNSD Service Publisher
Register/Publish Service
(s1) sync
publish/fetch sync data (a.k.a service advertisement)
(n1) NFD
Node A
ndn chunks communication
Finder Application
(5) receive service info { p1, service-info }
(1) find service ndnsd-locator -s <printer>
NDNSD Service Finder
(2) Fetch Service Names
(s2) sync
(4) fetch service-info
(3) publish/fetch sync data
(n2) NFD
Node B

Figure 7 Flow diagram showing interaction between several components

- Serve service-info data (3.4.1) under the application service data name. The name can be "/<prefix>/printers/printer1/NDNSD/service-info", and the corresponding data can be key-value pairs containing service details and other information.

3.3.2 Service Finder

- Receive discovery request from the user. Requests must contain a service type, e.g., /<prefix>/printers.

- Construct an appropriate sync group prefix using the service type received from the user and fetch all the service names (i.e. application names) belonging to the group via sync. e.g., service name: "/<root-prefix>/printers/printer1/NDNSD/service-info"

- For each service name, send an interest to fetch the corresponding service-info. NDNSD assumes that there is at least one service provider serving data for a given service name.

- Parse data packet to obtain service information and send it back to the user.
- Provide several pieces of measurement information, such as round-trip time, link-stability, re-transmission count, etc. via finder interface upon the request of the client.

Looking at some of the existing designs, we found a few tradeoffs in two functionalities offered by the discovery protocol. The first is proactively fetching and storing service-info from other producers and the second is parsing services on the behalf of a client to choose an appropriate one based on their need. Proactive fetching and storing can speed up the discovery process in some cases, but it also has a few notable downsides. For frequently updating services, pre-fetching can be a huge work catch up and storing can be a concern for devices with lower memory. Thus, to achieve our goal of making a general-purpose discovery protocol, we decided not to do proactive fetching because applications always remain up-to-date about the services, thanks to the sync protocol, and can fetch fresh service-info instantly if needed. However, service-info that are already fetched (upon request) are stored. Nonetheless, the service finder can also enforce NDNSD to fetch fresh info and not serve it from the storage or network cache.

Parsing services to choose an appropriate one based on the client's needs certainly benefits them in terms of processing and possible overhead. On the other hand, there are a few downsides to this approach. First, it weakens the purpose of making a general-purpose discovery by limiting its scope. Second, it forces clients to rely on the results provided by the NDNSD, causing inflexibility to implement their own logic. Finally, incurring changes directly needs to be implemented on the library. Thus, NDNSD does not parse the results on behalf of the client but provides flexibility to the developers to implement their logic and parse the results (list of services) to figure out the appropriate one based on the needs. This strengthens our goal of making the discovery process more generic. It also allows applications to model service details based on their need. For example, edge computing applications can have details such as computing power, CPU, Memory, availability, and so on for IoT, wireless, etc. Note: Standard formats such as JSON, XML, etc. can be used for the service-info.

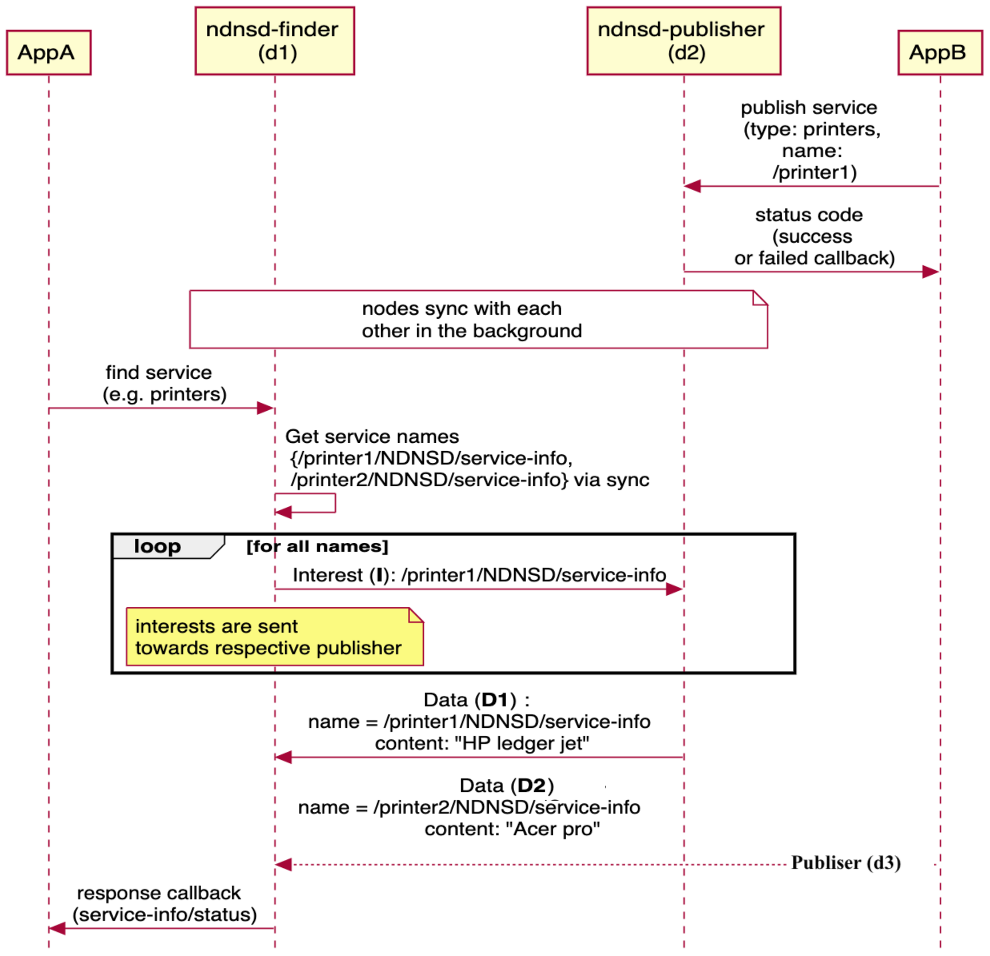

AppA
ndnsd-finder (d1)
ndnsd-publisher (d2)
AppB
publish service (type: printers, name: /printer1)
status code (success or failed callback)
nodes sync with each other in the background
find service (e.g. printers)
Get service names {/printer1/NDNSD/service-info, /printer2/NDNSD/service-info} via sync
loop
[for all names]
Interest (I): /printer1/NDNSD/service-info
interests are sent towards respective publisher
Data (D1) : name = /printer1/NDNSD/service-info content: "HP ledger jet"
Data (D2) name = /printer2/NDNSD/service-info content: "Acer pro"
Publiser (d3)
response callback (service-info/status)

Figure 8 NDNSD Sequence diagram, publishing service and fetching names and service-info

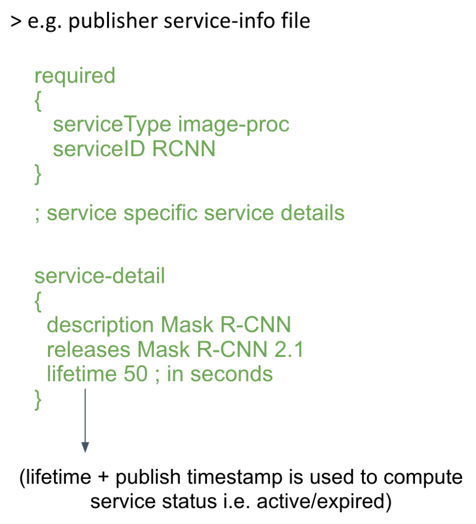
Flow Diagram (figure 7) and Sequence Diagram (figure 8) shows NDNSD in action. The following paragraph describes the process in brief.

First, a AppB advertises a service using a publisher application. It provides service-info that is composed of service-type: printers, service name (or service identifier): /printer1, and service details to the publisher. It then constructs an application service data name (*/printers/printer1/NDNSD/service-info*) using service-type and service-name and publishes it to sync group: */printers/NDNSD/discovery*, the name of which is constructed using the service-type. Thanks to sync, this publication is multi-casted to other nodes of the same sync group, meaning they get notified about the latest updates in within the group. Finally, the publisher will start listening on the application name (*/printer1/NDNSD/service-info*) for incoming interests. It will also send a callback to the AppB notifying whether the service registration succeeded or failed. There can be many other publisher applications (e.g., printer2) publishing to the same sync group. On the locator side, AppA requests for the service ”printer”. This means AppA is interested in discovering printers available in the network. When the locator receives the request, it first fetches all the application names (*/printer1/NDNSD/service-info, /printer1/NDNSD/service-info*) using sync. Next, it sends interests to each of these names and fetches the corresponding service info. Finally, the service-info received from the network is sent back to the AppA. Additionally, the locator also records the round-trip time (RTT), re-transmission count, and other measurement information that occurs while fetching the service-info. This information is provided to the users on demand.

3.4 Design Features

3.4.1 Service Info

Service info is a collection of information used by a service provider to advertise its service and is a crucial component of NDNSD. It is composed of required and optional fields. As shown in figure 9, the current design identifies service-type and serviceID as required filed. Service-type is used to construct the discovery



> e.g. publisher service-info file

```
required
{
   serviceType image-proc
   serviceID RCNN
}
; service specific service details

service-detail
{
  description Mask R-CNN
  releases Mask R-CNN 2.1
  lifetime 50 ; in seconds
}
```

(lifetime + publish timestamp is used to compute service status i.e. active/expired)

Figure 9 Sample service publisher configuration

name (sync prefix), and the serviceID is used to
construct the application prefix. Both prefixes
are used to advertise and serve the discovery data (section 3.2). Even though the required fields are a must, the actual meat of the service-info lies in the optional field, i.e. service-details. A service provider can choose details specific to its service, information they think might be useful to others. They can have as many key-value pairs as needed. For example, an image-processing service at the edge can provide details about models used to process the image, its released version, lifetime (how long the service is available), and so on. Constraint IoT applications can provide details about sensors (e.g., humidity, temperature), its location, memory, sleep time, processing capabilities, etc. Similarly, other applications can have details as per their need. This feature provides great flexibility to the application developers, and also underscores the purpose of making the protocol a general-purpose.

3.4.2 Measurement Information

Sometimes, service-info alone can be insufficient for a client to make a better judgment when choosing a service provider. For example, let's assume "A" and "B" are the providers of the same type of service, i.e storage. However, if A is far and the link between the client and A is unstable, it would be better for the client to avoid A and choose B. Measurement information such as round-trip time (RTT), re-transmission count, timeouts, link stability, etc, will come in handy while making such decisions. The NDNSD locator computes these parameters while fetching the service info or by periodically probing the service provider. The computation is not enabled by default for all the prefixes but the one's client request. However, sometimes the measurements can be misleading. For example, if the service-info data is served from the cache and not from the actual service-provider, the RTT can be significantly low. To mitigate this problem, we compute the parameters over a period to obtain a cumulative value. Also, NDN supports fetching fresh data from the publisher by setting the MustBeFresh flag with the interest. Thus, interest sent to fetch service info can use this feature to fetch data directly from the publisher and get better measurements.

3.4.3 Service-info Accessibility

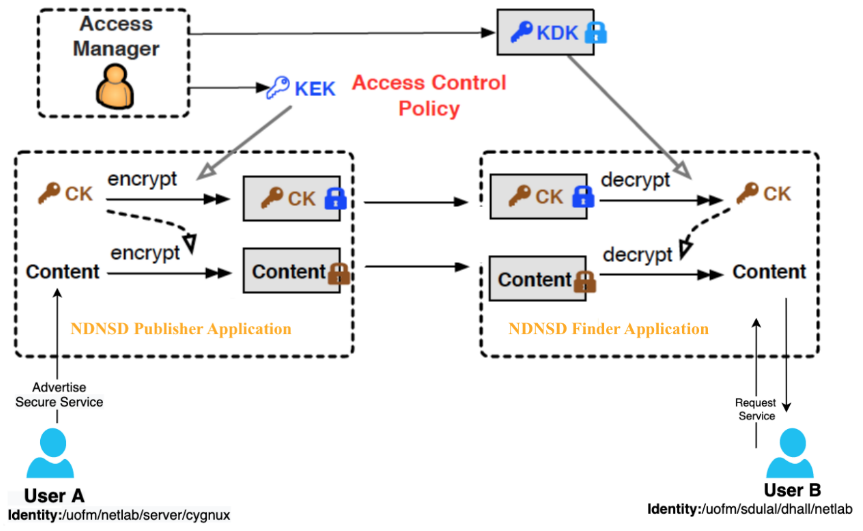


Figure 10 Use of Name Based Access control to prevent service-info accessibility from unauthorized users

The NDN data-centric security model helps to secure the data packet during its creation. However, most of the distributed application requires some form of access control scheme such that only authorized uses will be able to access the actual content. This is true for NDNSD as well. There can be several services advertised in the network of which service-info should be accessible to the authorized members only. For example, a printer from the manager's office might want to restrict public use. Some service-info might as well contain specific details on how to use the service and thus, may possess risk revealing it to the public. We use Name-Based Access Control (NAC) [49] to achieve this. NAC is an access control scheme that utilizes names and data-centric features of NDN to provide access control and data confidentiality. It is based on asymmetric and symmetric encryption algorithms. In the following paragraph, we will describe how it works with NDNSD.

The overall design of the scheme is shown in figure 10. It consists of three major components: access manager, publisher, and locator. Publishers and locators are also known as encryptors and decryptors based on their tasks. The access manager is responsible for managing the access policies and publishing them as named Key Encryption Key (KEK, public key) and Key Decryption Key (KDK, private key). The service locator receives KEK from the access manager, uses it to encrypt a Content Key (CK, a symmetric key), and finally, encrypts the content (service-info) using CK. Both the service-info data packet and CK are published in the network. Once the locator receives the packet and the respective content keys, it extracts the key names from the encoded data packet and obtains KDK. Using the KDK, first, it decrypts the CK, and next, the data packet using CK. Thus, only authorized locators can obtain KDK and access the service-info.

NDNSD doesn't use NAC by default for all the advertised services. It is up to the application to decide whether to use it or not based on the sensitivity of the service they offer and the content of their service-info. For example, public room switches and printers might not need access control. Additionally, the access control scheme does not apply to the discovery prefix or sync group. Everyone in the network (with a valid certificate) can query the group and obtain a list of service-names it carries. We do not protect this information because service-names are used to construct service-info interest names that are transparent to the network. Thus, we believe protecting the discovery prefix doesn't servers any useful purpose to our current design. However, we perform an in-depth analysis of its importance and use-cases in our future work.

3.5 Use Case

This section demonstrates the usability of the proposed protocol.

3.5.1 Service Directory

Knowing the service type prior (a.k.a sync group or discovery prefix) can be challenging for both Service Publisher and Service Locator applications. The Service Publisher needs to know the appropriate service type before joining and publishing its service. Similarly, the locator needs to know the service type before querying for services. The simplest approach can be to define and maintain a list of standard service-types similar to the DNS-SD. However, this still requires users to know the service-type beforehand. Additionally, NDNSD provides flexibility to the applications to create a new service type. Thus, for the service types that are created ad-hoc, the standard list will not be very helpful. To solve this problem, we propose a distributed Service Directory (SD)[3]. In a simplistic term, SD is just another sync group or service-type that is used to list all other service-types available in the network. A publisher application can opt to join the SD group and announce the specific service-type it is using to advertising the service. We assume the SD group prefix (*e.g., /<prefix>/service-directory*) is known to all the applications. For example, if a service publisher is offering a gaming service (*/<prefix>/gamers-001/fifa20*) and is advertising its service to the group (*/<prefix>/gamers-001/*), it can simultaneously join SD group, and let know SD about /<prefix>/memphis-gamers-001/. Similarly, other applications can do the same. Now, if a user trying to discover gaming services but doesn't know the actual name (service-type) of the gamer group, it can simply use the locator application, which, in turn, will join SD and discover the name */<prefix>/gamers-001/.* Next, the user can join */<prefix>/gamers-001/* and discover the actual services listed under this group. One of the services is */<prefix>/gamers-001/fifa20.* The process is not automated and requires judgment from the user to select an appropriated service-type returned by the SD group.

[3]Service Directory is an experimental idea and still in discussion phase

Importantly, give the enormous number of services that can exist today, currently, we image SD for the local environment only. However, SD can be extended beyond local hop but requires a deeper analysis, a better design, and implementation. Hence, we leave this for future work.

3.5.2 Building Management System

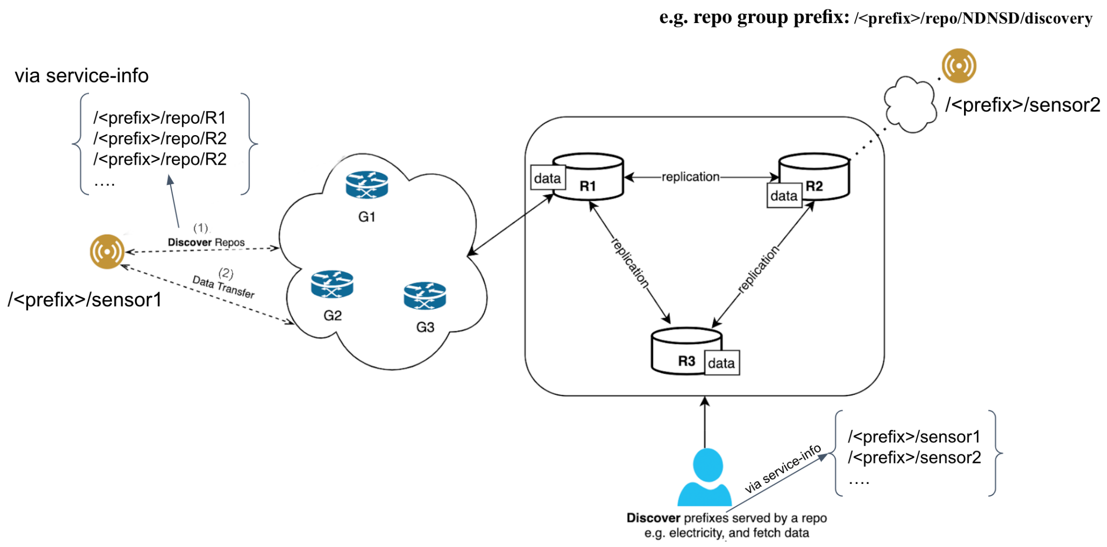


Figure 11 Simple Building Management System (BMS) [24] architecture highlighting the use of discovery service. R1, R2 and R3 are NDN repository (*repos* in short)

Building Management System (BMS) is an IoT application composed of hundreds of sensors, actuators, storage units, and services. It is used by enterprises as a cost-effective measure to monitor temperature, humidity, electricity, lighting, etc, across the buildings. Figure 11 shows a basic type of BMS architecture for an NDN enabled environment. It is a network of sensors, gateways, ndn repository (”repos” for short), and users. The sensors periodically produce the data and insert it into one of the repos. Once the data is received by a repo, it is replicated across multiple repos and made available to the uses to fetch it. The overall process looks very simple, but it poses enormous complexity in terms of management if the network size starts getting bigger. Where does the complexity come from? The first complexity is the on-boarding of the devices, i.e. plugging the devices into the network. This was not particularly the problem for us because we found some solid work in this area and our work assumes the devices are already on board with the network. The second and more significant problem we faced was the discovery of devices and services. It includes i) sensors discovering appropriate repos for the communication and vice-versa if repos want to discover sensors, ii) users discovering repos and the prefixes they serve the data for, ii) repos advertising services, and so on. In the absence of a discovery mechanism, all these setups need to be done manually, which is complex, time-consuming, and even impossible for bigger typologies consisting of hundreds of sensors and repos. We faced this problem while setting up an IoT testbed for BMS at the University of Memphis. This became a major motivation for this work.

Now, with the discovery protocol in action, the whole process becomes much more simplified and almost removes the manual configuration portion. Let us take figure 11 as an example and explain how it works.

- First, repos use NDNSD publisher to publish their service. The Service-type /<prefix>/repo, and the discovery data name is /<prefix>/repo/NDNSD/discovery.

- Next, sensor **s1** uses NDNSD service finder application to discover all the potential repos {/<prefix>/repo/R1, /<prefix>/repo/R2, /<prefix>/repo/R3} it can insert data to.

- Next, it chooses one of the repos from the list, and the Sensor **s1** uses repo insertion protocol to insert the data into the repo.

- On the user's side, like the sensor's, it uses NDNSD finder application to discover the repos and all the corresponding prefixes. e.g., /<prefix>/sensor1, /<prefix>/sensor2 they serve the data for.

- Finally, the user will fetch the desired data from the repo.

We have implemented and tested the BMS use-case in Mini-NDN. Please refer to the link[4] for more details.

[4]https://github.com/dulalsaurab/NDNSD/tree/master/experiments/bms-usecase

Chapter 4

Implementation

In this section, first, we will introduce some of the libraries (Chronosync, PSync) that are used to develop NDNSD. Subsequently, we will discuss the implementation detail NDNSD.

4.1 Chronosync

Chronosync [27] tries to improve the performance of dataset representation and synchronization by using a two-level Merkel[50] tree as shown in (figure 12). This tree is maintained locally and is constructed based on the updates they receive for each prefix belonging to the sync group. The level-one child in the tree contains the digest of the leaf knows as the node digest. The level-two child or the leaf node is assigned a unique name prefix (e.g., /a) that represents the node and a sequence number. Collectively they represent the state of the node. Each node uses its prefix to publish the data which is associated with a chronologically increasing sequence number that starts from zero at the beginning. Note that, as shown in the figure, lexicographical ordering of the node's name in the tree is important to compute the same root digest.

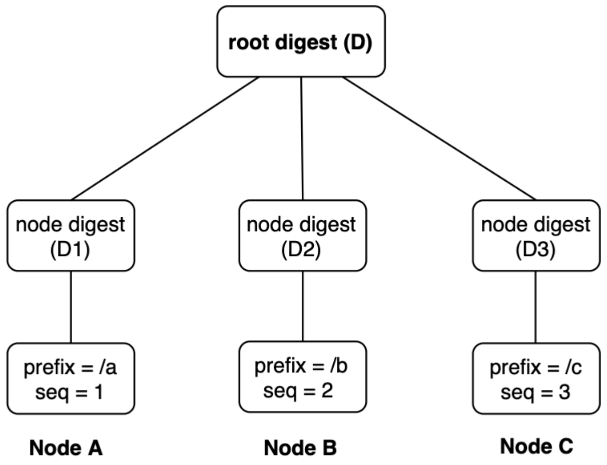


Figure 12 State Representation in Chronosync

Chronosync uses a long-lived sync interest, also known as pending interest, to fetch updates from the other nodes in the network. These interests always reside on the forwarder's PIT and get renewed if expired or satisfied by the sync data. The name (e.g. /<sync-prefix>/root-digest) consists of sync prefix and the state or root digest. Pending interest serves two important purposes, i) it helps to migrate the latest publish happening in the sync group from one node to another, and ii) since it carries the sender's digest, the receiver can compute inconsistencies between itself and the sender.

The dataset synchronization process is very similar to the one shown in figure 4. Once the new data is published by an application, the sync tree gets updated by updating the node and root digest. The publish will reply to the pending interest with its newly computed digest and the content (prefix + latest seq number e.g., /a/2). It will also renew its pending interest. Other nodes receiving the sync data will update their state accordingly and will also renew the sync interest with the latest digest.

4.2 PSync

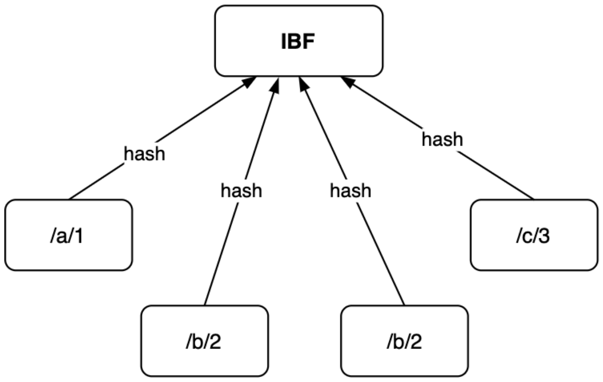


Figure 13 State Representation in PSync

PSync uses the Invertible Bloom Filter (IBF) to represent the shared dataset or state. Originally, PSync was designed to synchronize a subset of a dataset containing a larger number of prefixes, i.e., to satisfy the need of the consumer interested in a few prefixes only rather than the whole dataset, also known as

PartialSync. However, the specification supports both partial and full synchronization as full sync is only being a special case of partial sync where a consumer is interested in all the prefixes. Similar to Chronosync, the name prefix is attached with the latest sequence number. Thus, the IBF will only store a fixed-length hash of the latest names (figure 13) from the continuous streams. This significantly reduces the size of IBF and improves performance (lookup, reconciliation, insertion, etc.).

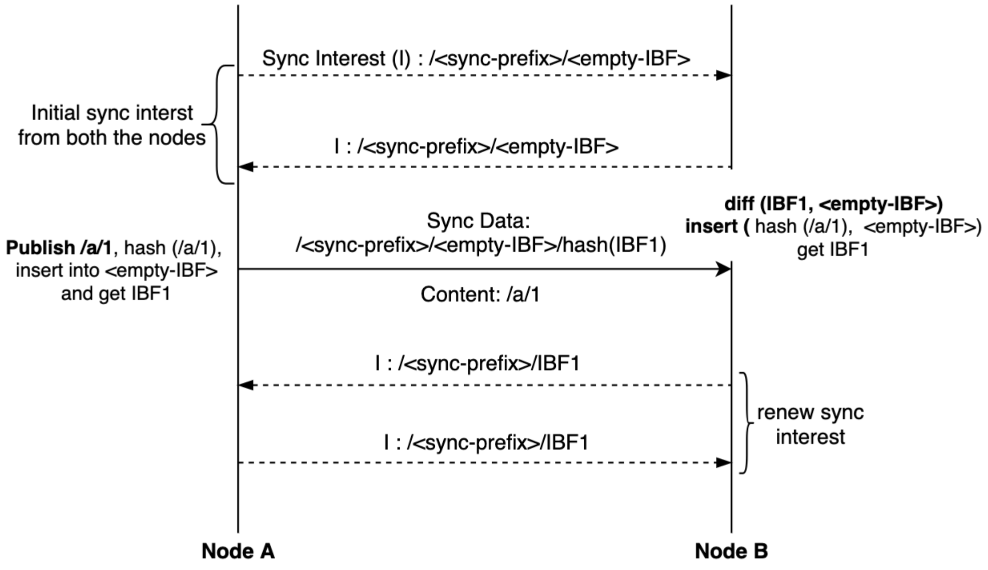

Sync Interest (I) : /<sync-prefix>/<empty-IBF>
Initial sync interst from both the nodes
I : /<sync-prefix>/<empty-IBF>
diff (IBF1, <empty-IBF>)
insert (hash (/a/1), <empty-IBF>)
get IBF1
Sync Data:
/<sync-prefix>/<empty-IBF>/hash(IBF1)
Publish /a/1, hash (/a/1), insert into <empty-IBF> and get IBF1
Content: /a/1
I : /<sync-prefix>/IBF1
renew sync interest
I : /<sync-prefix>/IBF1
Node A
Node B

Figure 14 Psync Sequence Diagram

A simple PSync state synchronization[1] is shown in figure 14. First, “NodeA” and “NodeB” join the sync group, construct an empty IBF, and exchange the sync interest containing an empty IBF. After a while, “NodeA” publishes data for the prefix “/a” with sequence number 1, computes the hash of the data name (i.e. /a/1) and inserts it into the initial empty IBF to obtain IBF1. Thereafter, NodeA replies to the pending sync interest from NodeB, with a sync data (/<sync-prefx>/<empty-IBF>/has(IBF1)) that contains the latest published prefix and its sequence number (/a/1). NodeB, upon receiving the sync data, performs a

[1]The diagram is based on the current release version 0.2.0 of PSync library

lookup and figures out the missing sequence number, i.e., 1 for /a in this case. Finally, NodeB updates it's IBF to IBF1 and renews its sync interest.

Both the protocols are implemented in C++ and provide a well-defined full-synchronization pub-sub API.

4.3 NDNSD Implementation

We have implemented[2] NDNSD protocol, explained in section 3.3 in C++, as a higher-level pub-sub library on top of some of the existing synchronization protocols. Several NDN synchronization protocols [51] exist today. As discussed earlier, we chose PSync and Chronosync for our implementation because of their availability and maintenance. Our implementation is adaptive to both libraries, and the protocol choice is exposed via the API for the application developers, meaning the application can decide what synchronization protocol to use for the discovery process. It is very important that both publisher and locator applications need to choose the same sync as they have their own data encoding scheme and mixing the protocol choice would result in data decoding failure. Our implementation also provides flexibility to incorporate more synchronization protocols, if needed in the future should they provide consumer-producer API similar to PSync and Chronosync. We have also designed a simple data publisher and locator application for testing and evaluation.

4.3.1 Interfaces

NDNSD library provides two high-level APIs for service publishing and discovery. The details of the APIs are shown in table 2

4.3.2 Service Info Representation

We use the Boost INFO format to represent service-info. The main rationale for using an info formatted file is because of its simplicity, standard key-value representation such as JSON, XML, etc., which are commonly used in NDN conf

[2]Some of the features such as measurement info are still in development phase

Table 2 NDNSD Implementation of Pub-Sub Interfaces

Publisher API	servicePublisher(File&& serviceInfo, List& pFlags, PublishCb cb)
Subscriber API	serviceFinder(Name&& serviceName, List&pFlags, DiscoveryCb cb)
Update/Reload	setUpdateProducerState(prodObject* obj, PublishCb cb)
Publisher/Subscriber uses sync protocol to announce and discover services. pFlags is used to pass protocol choice, enable measurement information, etc. serviceInfo file (3.4.1) is used to load the details into the publisher's state. DisocveryCb will deliver list of services discovered to the client. PublisherCb notifies client about service registration success or failure	

files (NFD, NLSR etc), and an already available tool [3] to edit the file. Following is an example of a service-info file.

```
required ;
{
  serviceType printer
  serviceID /printer2 ;service identifier
}
details ;user can have as many key-value as needed
{
  description "Hp Ledger Jet super xxx"
  make "2016"
  lifetime 1 ; in seconds
}
```

Listing 1. Sample Example of a Service-Info file in a Boost INFO format

As mentioned earlier in section 3.4.1), the service-info file is read by the publisher application to create its state and advertise the service. Any change in the service is made via this file and is re-loaded/re-advertised by the application using *setUpdateProducerState()* function.

[3]https://github.com/NDN-Routing/infoedit

4.3.3 State Representation

The latest information about the service is stored in the dataset called publisher's state or just *State*. It is constructed using the service information (obtained from service-info) and a service publication timestamp. State is a core data structure of the publisher application and is used in all the important operations, such as creating sync-group or discovery data-name, advertise the service, serving interest that comes in to fetch service-info, reloading the state via comparison, and so on.

4.3.4 Data Packet Specification

NDNSD data packet is constructed using NDN Packet Format Specification version 0.3 [29]. The packet is encoded using three different Type-length-value (TLV). The first TLV helps to identify whether the packet is from NDNSD or not, the second contains encoded service information in key-value pairs, and the last one is reserved for publication timestamp, which is used to compute service status (active or expired)

Status = current_time - publish_time > lifetime ? EXPIRED : ACTIVE.

If the lifetime is not provided in the service-info data packet, the service will be assumed to be ACTIVE.

Chapter 5

Evaluation

We performed both real-world and emulation experiments to evaluate NDNSD. We use PSync for all of the experiments because of its better performance [52] than Chronosync. In this section, we will discuss both types of experiments, our findings, and some of the challenges faced during the experiments.

5.1 Evaluation Terms and Metrics

Metrics

- **Sync Delay:** Time between a new data published by the producer and its update received by the consumer, also known as data synchronization delay.

- **Service-info Delay:** Time spent in fetching the service-info once the update is received from the sync.

- **Success Ratio (SR):** The percentage of interest that have received corresponding replies.

- **Aggregate Delay:** Overall service-info retrieval time after it's published. This is also a service discovery time and is equivalent to Sync delay + Service info delay.

Terms

- **Publish Interval:** Time between each successive publish, i.e., updating service-info or state by the Publisher.

5.2 Real world Experiments

For this experiment, Raspberry PI's (model 3b+) were distributed across Dunn Hall, University of Memphis, and were connected via an Access Point (AP). An example of the distribution along with the dimension of the room is shown in figure 15.

Time synchronization is very important for real-world experiments, especially when measuring delays and throughput. Thus, we use a Pi Zero as an NTP server

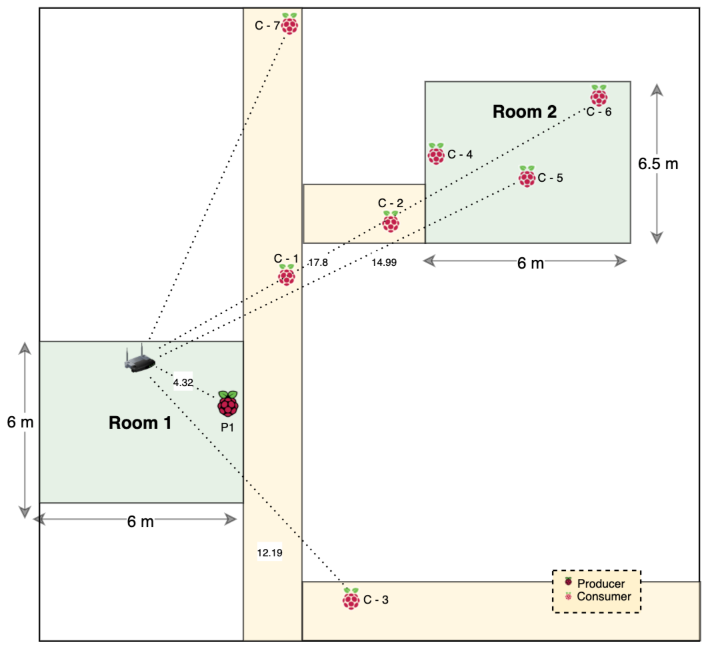


Figure 15 Raspberry Pi distribution across Dunn Hall, the University of Memphis for distance experiment. Here C represents service finder, and the number on the right side of C after "–", is a successive experiment count

for time synchronization among the devices. We perform two different types of experiments with this setup to measure delay distribution based on i) **Distance** and ii) **Congestion**. The main goal of both the experiments is to see how the protocol performs in a realistic environment and, more importantly, how it recovers the losses caused by congestion and low signal strength, a scenario for IoT.

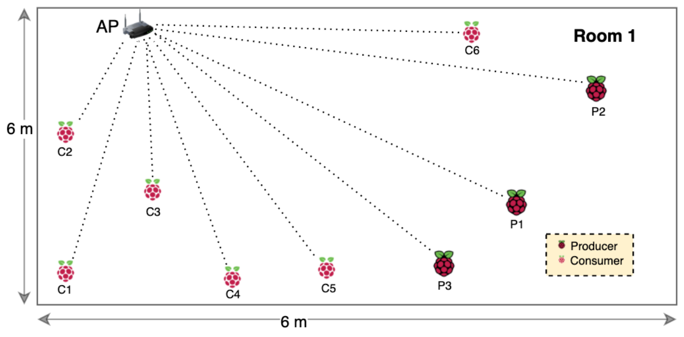


Figure 16 Device distribution for the congestion experiment. All the devices were placed in the same room and were connected to the access point AP

5.2.1 Distance experiment

In the distance experiment, the service publisher is kept at a constant distance from AP (in Room 1), while the service finder is at an increasing distance for each iteration (figure 15). Publisher reloads its state 300 times every 500ms. For each

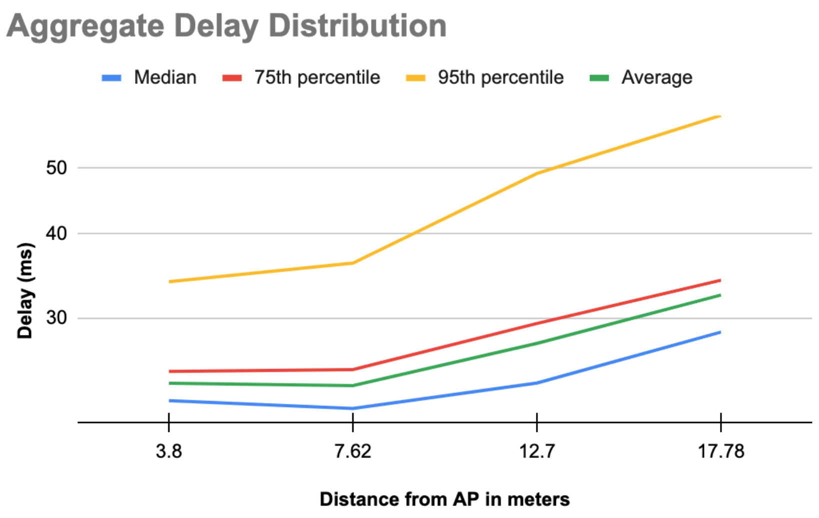


Figure 17 Aggregate percentile and average delay distribution of distance experiment

iteration, the experiment is repeated three times, and the average value is computed. The result of the experiment is shown in figure 17. We can see that the aggregate delay increases linearly as the distance between Locator and AP increases. We also observed <1% loss when the distance reached beyond 13m, but it was gained back by locator's re-transmission. The re-transmission count is configurable, meaning the Locator App can decide how many times it wants to try before giving up.

5.2.2 Congestion experiment

For the congestion experiment, all the devices are kept in the same room (Room 1), but the number of devices is increased with each pass, starting from one Publisher and one Finder all the way until there are two SP and six SF. Publish interval is set to 150ms (aggressive to increase congestion) and its service is updated 300 times. The result of the experiment is shown in figure 18. Like the distance experiment, the resultant percentile distribution increases with the increase in the number of devices connected with AP.

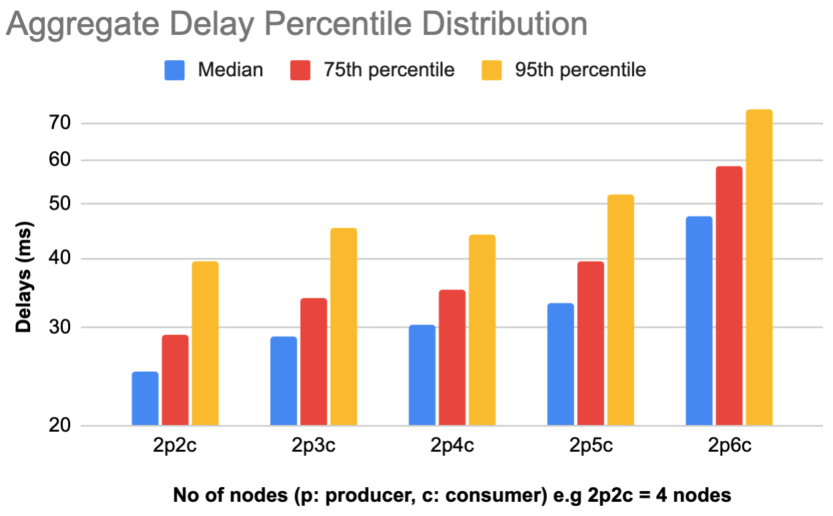


Figure 18 Aggregate delay percentile distribution obtained from congestion experiment

5.3 Emulation Experiments

We performed wired and wireless experiments using Mini-NDN[28] – a lightweight network emulator tool that runs real instances of NLSR and NFD. It virtualizes the whole environment by creating a node-specific container; all the processes are run within the container. Inter-node networking is achieved via virtual ethernet pairs (a.k.a. links).

First, we repeated the wireless congestion experiment (section 5.2.2) using *Wifi* module of Mini-NDN. Results obtained were very similar to the ones presented in figure 18, thus, they are not presented here. Next, we designed a multi-hop wired topology (figure 21) to verify the benefit of in-network caching of NDN and multi-cast efficiency of sync to NDNSD. Topology contains two Service Publisher (SP) and five Service Finder (SF) distributed with a minimum of one and maximum of four hops between SP and SF, e.g., C1 is four hops away from P2. The delay between each link is set to 10ms. In the experiment, SP publishes 300 times at an interval of 1000ms each.

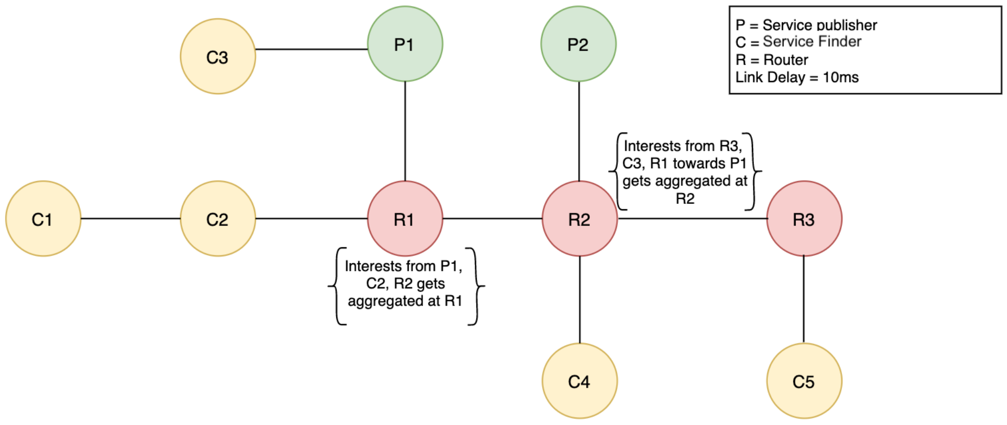

P = Service publisher
C = Service Finder
R = Router
Link Delay = 10ms
C3
P1
P2
C1
C2
R1
R2
R3
C4
C5
Interests from R3, C3, R1 towards P1 gets aggregated at R2
Interests from P1, C2, R2 gets aggregated at R1

Figure 19 10 node Mini-NDN topology with 5 service finders, 2 service publisher and 3 routers

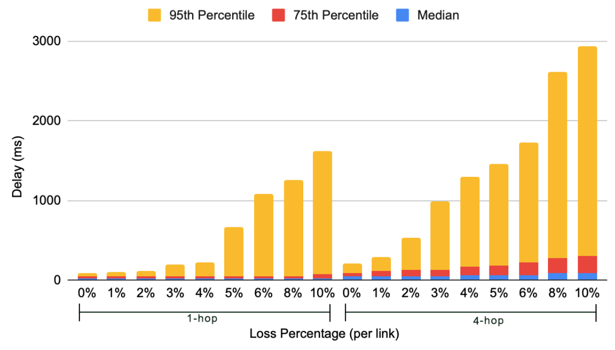
Table 3 Overall Service Info retrieval time in a multi-hop environment. Expected Minimum Link Cost (EMLC) = total assigned link cost for sync and service-info., e.g., consider 1-hop nodes C3 & P1: EMLC = 1 C3P1 (for sync) + 2C3P1 (for service-info) = 3*10 = 30. EMLC is without processing delay.

	Expected Minimum	Obtained Delays (RTT + processing delay)		
Hops	Link Cost (ms)	Median (ms)	75th Percentile (ms)	95th Percentile (ms)
1-hop	30	38.10	41.22	52.85
2-hop	60	73.38	81.93	128.93
3-hop	90	87.37	101.93	147.20
4-hop	120	100.07	119.77	161.78

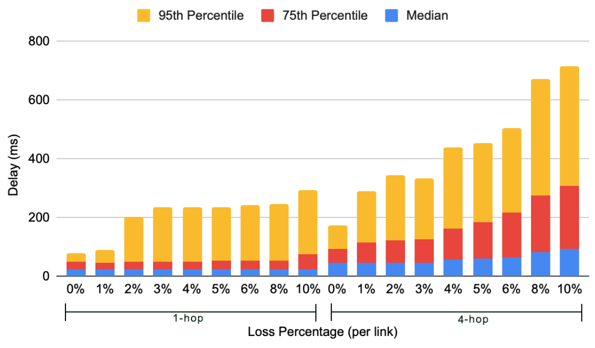
The result of the experiment, i.e., Aggregate Delays (5.1) is presented in table 3. For 1 and 2 hops, the obtained median delay is comparable to the expected minimum link cost. For 3-hops though, the value is about 3ms, and for 4-hops, it is almost 20ms below the actual expected minimum. This is, in fact, a significant gain and is achieved because of the in-network caching of NDN. For example, when C2 fetches data from P1 or P2, it also caches a copy of it in its content store. So, when C1 tries to fetch the same data, the interest will likely hit and get served from the CS of C1.

5.3.1 Loss and Partition Experiment

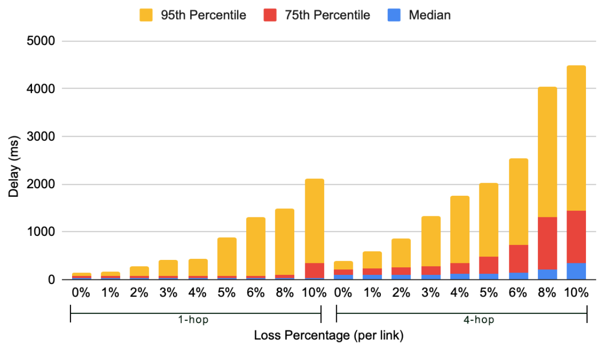
In this experiment, we took the topology 21 and assigned a certain constant percentage of loss across all the links. The loss rate was set to 1% at the beginning and was increased chronologically in each successive experiment up to 10%. The published frequency was set to 300, service publisher published new data 300 times at an interval of 1s each.



(a) Sync delay



(b) Service-info delay



(c) Aggregate delay

Figure 20 Sync, service-info, and aggregated delay percentile distribution for different per-link loss rates for 1-hops and 4-hops nodes. Note: we do not included results for 3 and 4 hops because they were identical with 1 and 4 hops results

Figure 20a and 20b shows an expected small increase in median and 75th percentile delay with the increases in loss percentage for the both 1-hop and 4-hop nodes. However, the 95th percentile sync delay increases quite significantly. In the extreme case i.e., 4-hop and 10% loss rate, the delay reaches almost 3000ms. The aggregate delay (figure 20c) follows the same pattern as that of sync delay because it is dominated by the sync delay numbers. At this moment, we don't have a better explanation of what is causing these extreme sync delays. We are still investigative the problem and will include the finding in our future work. It is important to note that the higher delay has nothing to do with the NDNSD protocol. It is very specific to the sync protocol that we have used to develop and experiment NDNSD. As the sync gets better with time, the performance will automatically improve.

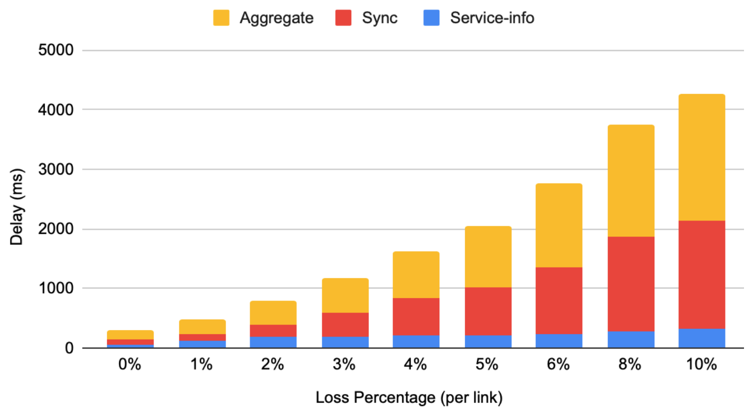


Figure 21 Average aggregate delay for all-hops with various per-link loss rate

5.3.2 Scaling Experiment

We design typologies of various sizes, listed in table 4, for the scaling experiment. The number of service publishers and service finders is doubled successively, starting from 1 SP, 3 SF until 16 SP and 48 SF, whereas the number of routers is kept constant. The SF and SP are connected evenly to the routers to distribute the packets fairly among the nodes. A general pattern for the creation of typologies of different sizes, mentioned in table 4, is shown in figure 22. However, the diagram (figure 22 (a)) doesn't hold for the two condition, i.e., 1SP and 3SF and 2SP and 6SF. For 1SP and 3SF, SF is connected with R1, and SP's with the rest of the routers, one at each. Similarly, for 2SP and 6SF, one pair of SP and SF are connected with R1, the next pair with R2, and the remaining 4SF are connected with R3 and R4, two in each of them. The typologies are designed considering the wireless scenario where there can be a few service publishers and a large number of service finder.

We used Tshark [53] to collect the data and processed it using ndndump [54]. To avoid the unreliable and tedious de-fragmentation work, we increase the MTU size to 9000 from its default size of 1500 and prevented the UDP packet (sync packet) fragmentation.

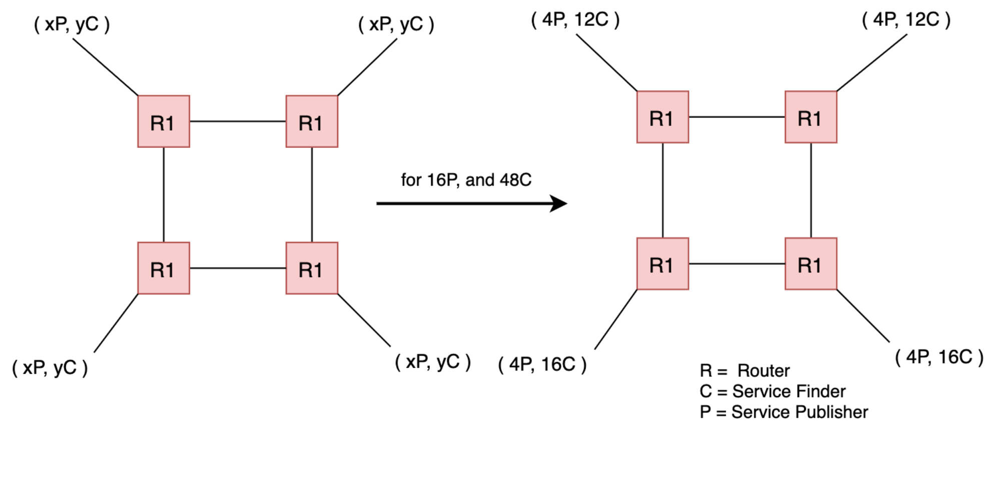


Figure 22 : Fig (a) is a generic topology that shows how service publisher (P) and service finder (C) are connected with the routers (R). "x" and "y" represents number of C and P respectively. Figure (b) is a case when 16P and 84C are connected with the routers to form a 64-node topology. Here, x = 4 and y = 12, meaning, 4 service publisher and 12 service finders are connected with each router (R).
Note: SF = C, SP = P are used interchangeably.

Table 4 Typologies used in scaling experiments. SP = Service Publisher and SF = Service Finder, R = Router

Nodes	Links	Descriptions
8	8	1 SP, 3 SF, 4 R
12	12	2 SP, 6 SF, 4 R
20	20	4 SP, 12 SF, 4 R
38	36	8 SP, 24 SF, 4 R
68	68	16 SP, 48 SF, 4 R

The table 5 presents an average number of packets received by each publisher. It shows that the publishers aren't much overloaded even if the number of finders grows significantly. For example, in the case of 68 node topology (16 SP, 48 SF), altogether 48*300 (number of SF * publication count) = 14400 service-info interests ae sent by all the SFs towards each SP. However, on average, each SP only receives

Table 5 Average number of packets per service publisher in scaling experiment. IPC = Interest Packet Count, DPC = Data Packet Count

Topology	Service Info IPC	Service Info DPC	Sync IPC	Sync DPC
8	302	302	3096	320
12	608	608	13142	1513
20	1219	1219	34822	7070
38	2444	2444	105866	30324
68	4954	4952	364966	146643

4954 service-info interests, almost 66% less than the actual number of interests sent towards it. The performance gain is because of the interest aggregation and data-caching that took place at each router. Say, if multiple interests arrived at the router "R1" for the same piece of data, all of them will get aggregated and only one will be sent out towards the producer. Interests are also served from the router's cache if the corresponding data is already available there. Thus, the router can act as a proxy and help reduce the upstream traffic significantly.

The number of sync packets is significantly higher compared to service-info. This is because sync operates on multicast, every single packet generated or received by sync are multicasted throughout the network. The number of sync interest packets is even higher compared to the data packets because PSync uses long-lived sync interest. These interests always reside on the router's PIT and immediately get renewed if satisfied by the sync data or timeout. We used the available multicast strategy for our experiment. It helped to reduce the number of interest packets by dropping the duplicate ones but is still insufficient. We believe a better interest and data suppression mechanism at the NFD's face level can improve multicast communication, and finally, the performance of the application using it.

Chapter 6

Conclusion and Future Works

In this thesis, we first discussed the requirement of service discovery in the modern application and provide rationales for choosing NDN for service discovery instead of TCP/IP. We presented NDNSD, a fully distributed, general-purpose service publishing, and discovery protocol for NDN. We realized a few key requirements such as multi-party communications and data synchronization of modern discovery protocols to operate in a distributed environment. We discovered that these needs are mostly satisfied with NDN synchronization. Thus, we decided to design NDNSD on top of the sync protocol. We extended the pub-sub feature offered by sync and made it, even more, simpler high-level API for the application to publish and find services. The hierarchical namespace design of NDNSD prefixes provides the applications a fine-grained control and freedom to choose the domain where they want to publish their service, and under the desired name. We introduced the concept of service-info, service-specific information. We showed how service-info is be combined with sync and hierarchical names to achieve our goal of designing a general-purpose service discovery protocol. Network measurement information (RRT to service, link-stability) is crucial for applications to determine the Quality of Service. Higher-level applications are opaque to the network layer and are clueless about lower-level performance. Thus, to help the finder application make a better decision in selecting service providers, NDNSD exposes network metrics via the API to it. Sometimes service-info being public can cause problems. It can give service accessibility to an unauthorized user and possibly can reveal information intended for specific users to the public. We use Name-Based Access Control (NAC) to control the visibility of service-info. NAC encrypts the service-info and enables only those who have the appropriate keys to decrypt the packet and view the service info. However, NDNSD doesn't obfuscate the application service prefix. Meaning, every user can see the names belonging to a service group. At this moment, we consider exposing the names as harmless because even obscuring cannot protect its visibility. Because, these names are used to construct interests, and interest names are easily visible throughout the network. However, this still requires a more detailed discussion and analysis and will be our future work.

Knowing service-type or sync groups before advertising or discovering the service can be difficult. We provide a use-case (distributed Service Directory (SD)) that explains how NDNSD itself can be used to list service-type available in the network. While SD may need a detailed feasibility study to employ it on a larger scale give a huge amount of service that possibly can exist, it can be easy to deploy and used in the local environment. Large scale use also relates to another problem. NDNSD assumes every single node participating in the sync group responds to sync interest. Sync prefixes are set to multicast strategy, the synchronization process generates an enormous amount of multicast packets. Even though interest aggregation, duplicate suppression (by the multicast strategy), and in-network caching can help to reduce the traffic to some extent, this may still cause significant scalability issues, especially in a constraint environment. One possible solution can be role assignments: different nodes in a sync group have different roles. Some nodes are designated as responders, only reply to sync interest. While others are designated as receivers, only received sync data but never reply to the sync interest. We will explore the possibility of role assignment in our future work. Due to the lack of multicast suppression in the current version of NFD (0.7.1) [55], we observed an enormous amount of unsolicited data and interest packets, and a significant increase in network congestion. It was practically impossible to scale beyond 10 nodes with a high publication frequency i.e., < 100ms. Thus, we will also investigate face level multicast suppression in our future work.

REFERENCES

[1] Cisco, "Cisco annual internet report (2018–2023) white paper," 2020, accessed: 2020-04-22. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[2] V. Jacobson, M. Mosko, D. Smetters, and J. Garcia-Luna-Aceves, "Content-centric networking," *Whitepaper, Palo Alto Research Center*, pp. 2–4, 2007.

[3] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos, *et al.*, "Named data networking (ndn) project," *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC*, vol. 157, p. 158, 2010.

[4] X. Zhang, B. Zhao, A. Chakraborti, R. Ravindran, and G.-Q. Wang, "Name-based neighbor discovery and multi-hop service discovery in information-centric networks," Dec. 6 2016, uS Patent 9,515,920.

[5] E. Meshkova, J. Riihijärvi, M. Petrova, and P. Mähönen, "A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks," *Computer networks*, vol. 52, no. 11, pp. 2097–2128, 2008.

[6] M. Jeronimo and J. Weast, *UPnP design by example: a software developer's guide to universal plug and play.* Intel Press, 2003.

[7] M. Boucadair, R. Penno, and D. Wing, "Universal plug and play (upnp) internet gateway device-port control protocol interworking function (igd-pcp iwf)," *RFC 6970*, 2013.

[8] "Ieee standard for local and metropolitan area networks – station and media access control connectivity discovery," *IEEE Std 802.1AB-2005*, pp. 1–176, 2005.

[9] E. Guttman, "Service location protocol: Automatic discovery of ip network services," *IEEE Internet Computing*, vol. 3, no. 4, pp. 71–80, 1999.

[10] L. Smith, C. Roe, and K. S. Knudsen, "A jini/sup tm/ lookup service for resource-constrained devices," in *Proceedings 2002 IEEE 4th International Workshop on Networked Appliances (Cat. No.02EX525)*, 2002, pp. 135–144.

[11] E. A. Gryazin, "Service discovery in bluetooth," *Group for Robotics and Virtual Reality. Department of Computer Science. Helsinki University of Technology, Helsinki, Finland. Published at NEC CiteSeer, Scientific Literature Digital Library*, 2006.

[12] P. Mockapetris *et al.*, "Domain names-implementation and specification," 1987.

[13] S. Cheshire and M. Krochmal, "Dns-based service discovery," RFC 6763, February, Tech. Rep., 2013.

[14] W. Shang, Y. Yu, R. Droms, and L. Zhang, "Challenges in iot networking via tcp/ip architecture," *Technical Report NDN-0038. NDN Project*, 2016.

[15] W. Shang, A. Bannis, T. Liang, Z. Wang, Y. Yu, A. Afanasyev, J. Thompson, J. Burke, B. Zhang, and L. Zhang, "Named data networking of things," in *2016 IEEE first international conference on internet-of-things design and implementation (IoTDI)*. IEEE, 2016, pp. 117–128.

[16] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," 2014.

[17] Z. Shelby, "Core resource directory; draft-ietf-core-resource-directory-02."

[18] D. H. Steinberg and S. Cheshire, *Zero Configuration Networking: The Definitive Guide: The Definitive Guide.* " O'Reilly Media, Inc.", 2005.

[19] D. C. Plummer *et al.*, "Ethernet address resolution protocol: Or converting network protocol addresses to 48. bit ethernet address for transmission on ethernet hardware." *RFC*, vol. 826, pp. 1–10, 1982.

[20] SmartThings Inc., "Smartthings," https://www.smartthings.com/., (Accessed on 10/21/2020).

[21] Amazon Inc., "AWS IoT," https://aws.amazon.com/iot/solutions/connected-home/., (Accessed on 10/21/2020).

[22] S. Weber, *Chromecast Users Manual: Stream Video, Music, and Everything Else You Love to Your TV.* USA: Weber Systems Inc., 2014.

[23] Mircosoft Inc., "Azure iot edge, 2020," https://azure.microsoft.com/en-us/services/iot-edge/., (Accessed on 10/21/2020).

[24] W. Shang, A. Gawande, M. Zhang, A. Afanasyev, J. Burke, L. Wang, and L. Zhang, "Publish-subscribe communication in building management systems over named data networking," in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2019, pp. 1–10.

[25] A. Gawande, J. Clark, D. Coomes, and L. Wang, "Decentralized and secure multimedia sharing application over named data networking," in *Proceedings of the 6th ACM Conference on Information-Centric Networking*, 2019, pp. 19–29.

[26] M. Zhang, V. Lehman, and L. Wang, "Scalable name-based data synchronization for named data networking," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[27] Z. Zhu and A. Afanasyev, "Let's chronosync: Decentralized dataset state synchronization in named data networking," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*. IEEE, 2013, pp. 1–10.

[28] Mini-NDN, "Mini-ndn – a light weight ndn emulator tool," 2020, accessed: 2020-04-22. [Online]. Available: http://minindn.memphis.edu/

[29] "Ndn packet format specification 0.3 documentation," https://named-data.net/doc/NDN-packet-spec/current/index.html, (Accessed on 10/27/2019).

[30] C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang, "A case for stateful forwarding plane," *Computer Communications*, vol. 36, no. 7, pp. 779–791, 2013.

[31] P. de-las Heras-Quirós, E. M. Castro, W. Shang, Y. Yu, S. Mastorakis, A. Afanasyev, and L. Zhang, "The design of roundsync protocol," *Technical Report NDN-0048, NDN, Tech. Rep.*, 2017.

[32] W. Shang, A. Afanasyev, and L. Zhang, "Vectorsync: distributed dataset synchronization over named data networking," in *Proceedings of the 4th ACM Conference on Information-Centric Networking*, 2017, pp. 192–193.

[33] T. Li, W. Shang, A. Afanasyev, L. Wang, and L. Zhang, "A brief introduction to ndn dataset synchronization (ndn sync)," in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM).* IEEE, 2018, pp. 612–618.

[34] W. Fu, H. Ben Abraham, and P. Crowley, "isync: a high performance and scalable data synchronization protocol for named data networking," in *Proceedings of the 1st ACM Conference on Information-Centric Networking*, 2014, pp. 181–182.

[35] M. Mosko, "Ccnx 1.0 collection synchronization," in *Technical Report.* Palo Alto Research Center, Inc., 2014.

[36] R. Ravindran, T. Biswas, X. Zhang, A. Chakraborti, and G. Wang, "Information-centric networking based homenet," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013).* IEEE, 2013, pp. 1102–1108.

[37] ndn lite, "Ndn-lite service discovery," 2018, accessed: 2020-04-22. [Online]. Available: https://github.com/named-data-iot/ndn-lite/wiki/Service-Discovery

[38] M. Amadeo, C. Campolo, and A. Molinaro, "Ndne: Enhancing named data networking to support cloudification at the edge," *IEEE Communications Letters*, vol. 20, no. 11, pp. 2264–2267, 2016.

[39] A. Mtibaa, R. Tourani, S. Misra, J. Burke, and L. Zhang, "Towards edge computing over named data networking," in *2018 IEEE International Conference on Edge Computing (EDGE).* IEEE, 2018, pp. 117–120.

[40] S. Mastorakis and A. Mtibaa, "Towards service discovery and invocation in data-centric edge networks," in *2019 IEEE 27th International Conference on Network Protocols (ICNP).* IEEE, 2019, pp. 1–6.

[41] S. Mastorakis, A. Mtibaa, J. Lee, and S. Misra, "Icedge: When edge computing meets information-centric networking," *IEEE Internet of Things Journal*, 2020.

[42] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[43] B. A. Forouzan, *TCP/IP protocol suite.* McGraw-Hill, Inc., 2002.

[44] V. Jacobson, “Watching ndn’s waist: How simplicity creates innovation and opportunity,” 2019, accessed: 2020-04-22. [Online]. Available: https://pollere.net/Pdfdocs/ICN-WEN-190715.pdf

[45] S. Deering, “Watching the waist of the protocol hourglass,” 2001, accessed: 2020-04-22. [Online]. Available: https://www.iab.org/wp-content/IAB-uploads/2010/11/hourglass-london-ietf.pdf

[46] MQTT Org., “MQ Telemetry Transport, 2020,” http://mqtt.org/, (Accessed on 10/21/2020).

[47] DNS-SD Org., “DNS SRV (RFC 2782) Service Types, 2020,” http://www.dns-sd.org/ServiceTypes.html, (Accessed on 10/21/2020).

[48] Z. Zhang, Y. Yu, H. Zhang, E. Newberry, S. Mastorakis, Y. Li, A. Afanasyev, and L. Zhang, “An overview of security support in named data networking,” *IEEE Communications Magazine*, vol. 56, no. 11, pp. 62–68, 2018.

[49] Z. Zhang, Y. Yu, S. K. Ramani, A. Afanasyev, and L. Zhang, “Nac: Automating access control via named data,” in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018, pp. 626–633.

[50] R. C. Merkle, “Protocols for public key cryptosystems,” in *1980 IEEE Symposium on Security and Privacy*. IEEE, 1980, pp. 122–122.

[51] W. Shang, Y. Yu, L. Wang, A. Afanasyev, and L. Zhang, “A survey of distributed dataset synchronization in named data networking,” *NDN, Technical Report NDN-0053*, 2017.

[52] A. Gawande, “Improvements to psync: Distributed full dataset synchronization in named-data networking,” Ph.D. dissertation, University of Memphis, 2019.

[53] G. Combs, “Tshark—dump and analyze network traffic,” *Wireshark*, 2012.

[54] NDN Project Team, “NDN Essential Tools,” https://github.com/named-data/ndn-tools, (Accessed on 10/29/2019).

[55] The NDN Team, “Named Data Networking Forwarding Daemon,” https://github.com/named-data/NFD, (Accessed on 11/28/2020).