NDN, Technical Report NDN-0071. http://named-data.net/techreports.html
Revision 1: Jan 11, 2021

A Pub/Sub API for NDN-Lite with Built-in Security

Tianyuan Yu
UCLA
tianyuan@cs.ucla.edu

Zhiyi Zhang
UCLA
zhiyi@cs.ucla.edu

Xinyu Ma
UCLA
xinyu.ma@cs.ucla.edu

Philipp Moll
UCLA
phmoll@cs.ucla.edu

Lixia Zhang
UCLA
lixia@cs.ucla.edu

ABSTRACT

Named Data Networking (NDN) is a new data-centric Internet architecture design, and NDN-Lite is an IoT networking framework that aims to enable end user controlled smart homes. This paper presents NDN-Lite Pub/Sub's design and implementation. By using names that carry application semantics to secure data and construct security policies directly, NDN-Lite Pub/Sub provides a usable and high-level API that enable developers to write NDN-Lite applications for home IoT scenarios with built-in security support. We measured the latency and memory overhead of NDN-Lite Pub/Sub on resource-constrained devices, and the results show it can effectively run on home devices.

KEYWORDS

Named Data Networking, Pub/Sub, API

1 INTRODUCTION

In the publish-subscribe messaging pattern (Pub/Sub), producers of a message – referred to as *publishers* – do not specify the message receivers, but assign a topic to every message. Nodes that are interested in a topic – referred to as *subscribers* – express their interest in the topic by subscribing to it and consequently receive all messages labeled with the specified topic. Our current Internet architecture does not allow specifying semantic information, such as message topics, on network level and packet forwarding is limited to endpoint addresses only. This limitation requires the use of higher-layer middleware when building Pub/Sub systems. The novel networking architecture Named Data Networking (NDN), however, utilizes semantic names as message identifiers and for forwarding purposes. This semantic name information can be exploited directly on the network level to realize Pub/Sub systems.

The NDN-Lite [18] framework is an IoT framework that enables building user-controlled smart homes. One component of NDN-Lite is the NDN-Lite Pub/Sub protocol, which is the focus of this report. The key contributions of this paper are:

- the development of a Pub/Sub design and API for NDN-Lite with built-in security support,
- the implementation of NDN-Lite Pub/Sub integrated in the NDN-Lite SDK,
- a measurement of the latency and memory overhead of NDN-Lite Pub/Sub. The results show a minor footprint that allows executing it on resource-constrained devices, such as found in IoT scenarios.

The rest of the paper is organized as follows: In Section 2 relevant background for the design of NDN-Lite Pub/Sub is presented. Section 3 describes the design of NDN-Lite Pub/Sub, including its API design and protocol details. Section 4 provides additional information about the current implementation as a part of NDN-Lite. An evaluation of the implementation's memory and latency overhead is presented in Section 5. Section 6 describes related work with a focusing on high-level APIs for NDN, and Section 7 concludes the paper and addresses future work.

2 BACKGROUND

2.1 Named-Data Networking

2.1.1 Naming. Named Data Networking (NDN) [17] is a data-centric Internet architecture, which directly uses application-layer names for networking; there is no address in an NDN network. Each data piece is identified by the name given by the application, and application naming generally follows a set of established rules – noted as naming convention [16] – agreed between data producer and consumer applications. Naming data enables one to secure data directly: a producer can uses its own key to cryptographically sign every data packet it produces, binding the packet's name to the content. Therefore, an Data packet consists of Name, Content, and Signature as shown in Figure 1, making the packet verifiable by any consumer, independent of the underlying communication channels.

2.1.2 Interest-Data Exchange. In an NDN network, a consumer can fetch Data by sending an Interest packet. Interests contain either the exact data Name or a Name prefix of the requested Data. Once emitted, an Interest is forwarded through

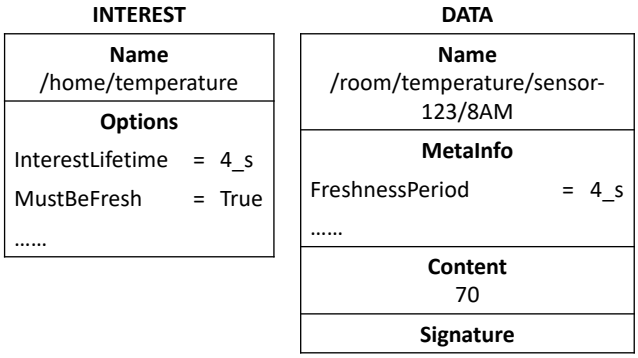

Figure 1: Example for Interest and Data packets

the network based on its name on a hop-by-hop base until it reaches a node holding a copy of the requested Data. The Data travels back on the reverse path of the Interest, realized by the Interest leaving breadcrumbs on each forwarding node, implemented as an entry in the so-called Pending Interest Table (PIT). To preserve flow-balance, one Interest can retrieve at most one Data packet. The example provided in Figure 1 shows the packet structure for Interest and Data packets. Reliability in data delivery is consumer-driven in NDN. The consumer defines a retransmission time for each outstanding Interest; if an Interest's lifetime expires without receiving Data, the consumer may choose to re-express the Interest.

2.1.3 Security. NDN's security design contains four key components to achieve secure communication. They are trust anchors, identities, certificates, and security policies. The trust anchor needs to be configured into all participants in an NDN network in a secure out-of-band manner. Each participant (which can be a device, an app, or a user) has a name and a certificate issued by the trust anchor; we refer to them as *entities* in the rest of this section [20]. Certificates and security policies also possess semantically meaningful names. Thus they are named, secured data, and can be retrieved via Interest-Data exchange. This allows automating security workflows, as done in our Pub/Sub security workflow, as elaborated in Section 3.2.

Figure 2 visualizes the relation of the four security components in an example application. In this example, a smart homeowner Alice owns both a temperature sensor, which produces temperature data, and an air-conditioner. The latter is responsible for controlling the room temperature, therefore it needs to consume the sensor's data.

- *Trust Anchors:* A trust anchor is the combination of the name of a trusted entity and its pair of trusted asymmetric keys [5]. A trust anchor certificate is a self-signed certificate and installed on all entities via an out-of-band channel. This certificate is the root of trust for all the entities in the system under the trust anchor, and is used to validate certification chains in public-key certificates. With the trust anchor's certificate installed, an NDN entity can verify Data signatures produced by other entities by validating their certificates along the certification chain. In the example application, Alice may generate a self-signed certificate named "`/alice-home/KEY/222/self/001`" as the trust anchor certificate. This certificate is installed on the temperature sensor and the air-conditioner eg. by scanning a QR-code. This allows the verification of certificates issued by Alice and ultimately, to verify Data produced by each other.

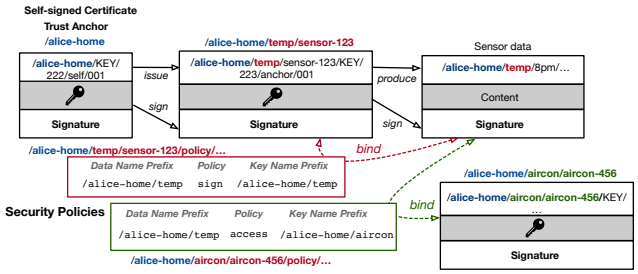

Figure 2: Relationship between /alice-home, /alice-home/temp/sensor-123, and the data produced by the sensor.

- *Identities and Certificates:* In NDN, every entity that produces Data can be identified by its identity. An identity is realized as a public-private key pair bound to a name, where the public key is embedded in named certificates. The system's trust anchor certifies identities by binding an entity's public-key to a certificate name, under which it becomes available for other entities. Certificate names match the following naming convention "`/<prefix>/KEY/<key-id>/<issuer-info>/<version>`". In the example in Figure 2, the temperature sensor's certificate is issued by the trust anchor. The certificate name conveys the temperature sensor as the certificate owner and a key ID (ie. *223*) identifying the public-private key pair belonging to the certificate. Besides, it indicates that the trust anchor is the signer of the certificate and includes *001* as the certificate's version number.
- *Security Policies:* In order to authenticate Data and enable access control, trust rules [15] and named-based access control [19] policies are also needed. Accordingly, applications can define security policies to ① restrict Data under specific namespace only allowed be *signed* by specific keys. ② restrict Data under specific namespace only allowed be *accessed* by specific identities. Assuming Alice only wants to authenticate temperature sensors to produce temperature data, and allow the air-conditioner to access the temperature data.

Therefore, Alice specifies a signing policy (the red box in Figure 2) that Data under prefix "`/alice-home/temp`" can only be signed by keys under the same prefix. Meanwhile, an access policy (the green box in Figure 2) is also defined to restrict Data under "`/alice-home/temp`" can only be accessed by identities under prefix "`/alice-home/aircon`", limiting them only accessible by "`aircon-456`". Then Alice can use the trust anchor to sign two Data containing the above two policies respectively, and let the the temperature sensor and air-conditioner to fetch and install their policies.

The security policies, together with trust anchor, identities and certificates enable NDN to achieve the data authenticity goal. These four components have to be installed into an NDN entity in its bootstrapping step. To achieve the data confidentiality goal and realize access control based on security policies, NDN employs encryption keys to protect the data content.

2.2 NDN-Lite

NDN-Lite is a lightweight version of the NDN implementation that fits into resource-constrained devices, commonly used in smart home environments. Its design aims to enable a secure, local user-controlled home IoT system. NDN-Lite unifies home devices and applications as *participants*. When using NDN-Lite, the homeowner sets up a controller as the local trust anchor, which issues certificates for participants and can control the whole system on behalf of the home users. Besides, NDN-Lite implements high-level application support, such as security bootstrapping, access control, trust management. NDN-Lite's design makes the following assumptions.

- Broadcast Network: All system participants are connected to a broadcast network (i.e., single-hop home wireless network).
- Device capabilities: All home devices are capable of being wireless connected, performing asymmetric key signing (e.g., ECDSA), and keeping adequate memory to store information for NDN-Lite.
- Security: The controller is trustworthy, and all system participants can be securely bootstrapped.

New smart home participants are bootstrapped by the homeowner through the controller using bootstrapping protocols, such as SSP [8] or NDNViber [12]. As indicated in Section 2.1.3, participants obtain the trust anchor, the identity, and security policies from this bootstrapping step.

To restrict the participants' access to Data under permitted name prefixes only, NDN-Lite uses named-based access control [19] and symmetric encryption. After the bootstrapping process, participants request encryption and decryption keys for specific name prefixes from the controller. After verifying that the security policies allow access, the controller grants access by securely delivering the corresponding symmetric keys to the participant.

3 DESIGN OF NDN-LITE PUB/SUB

In this section, we show how NDN-Lite Pub/Sub provides a high-level API for NDN IoT with built-in security support. Among numerous messaging patterns, the *Publish-Subscribe (Pub/Sub)* pattern allows decoupling publishers and subscribers. Data is published by attaching it to *resources* and notifying the system about the production of such data. Intended recipients can subscribe to *resources* by asking the system to deliver data of interest without actually knowing publisher addresses[1]. This property naturally fits into IoT applications' need for data-centric many-to-many communication. Therefore, we chose publish-subscribe as the messaging pattern for our API.

How to realize Pub/Sub in NDN: To realize Pub/Sub with NDN as the network layer, we rely on the application's use of semantic data naming. Semantic information in Data names allows specifying *resources* as name prefixes. Hence, subscribers can express the resources they are interested in by subscribing to a set of name prefixes. Focusing on security, NDN-based security models (cf. Section 2.1.3) provide the foundation for automating security workflows and make them transparent for developers. Thus, an NDN-based Pub/-Sub systems should embed name-based security models and guarantee that the received Data is verified and authenticated.

Overview of NDN-Lite Pub/Sub: NDN-Lite organizes application data with a name tree (cf. Section 3.1). Such a name tree describes the low-level application logic so that developers can focus on implementing higher-level application components. Security policies are expressed by the homeowner in a controller application and specify trust relationships between name tree nodes (cf. Section 3.2). Home IoT functions are categorized into *collecting sensor readings* and *actuating commands*. NDN-Lite Pub/Sub provides a unified API for these function categories (cf. Section 3.3). The network-level realization of the API is described in Section 3.4. In Section 3.5, we show NDN-Lite Pub/Sub's built-in security support. This security support is realized by automating security workflows for publishers and subscribers so that low-level cryptographic primitives are hidden from developers.

Type	Naming Convention
Device&Application Certificate	/<home-prefix>/<service>/<location>*/<participant-id>/KEY/<key-id>
Device&Application Policy	/<home-prefix>/<service>/<location>*/<participant-id>/POLICY/<timestamp>
Application Data	/<home-prefix>/<service>/<location>*/<data-id>/<timestamp>
Encryption Symmetric Key	/<home-prefix>/<service>/ACCESS/EKEY/<key-id>/<timestamp>
Decryption Symmetric Key	/<home-prefix>/<service>/ACCESS/DKEY/<key-id>/<timestamp>

Notation: A component with <> represent a variable, e.g., /<> can be /TEMP representing temperature service. A component with <>* represent a variable that may consists of multiple name components, e.g., /<location>* can be /kitchen or /kitchen/sensor-123.

Table 1: Naming Conventions

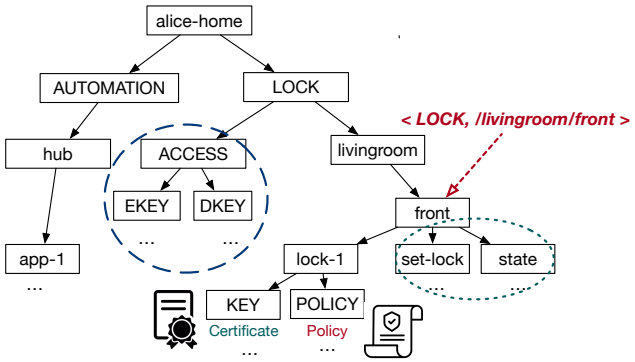

Figure 3: Sketch of an example name tree

3.1 Name Tree

In NDN-Lite Pub/Sub, we group smart home data based on two high-level properties: *service* and *location*. The *service* describes the abstract home service to which the Data is related, while the *location* specifies the physical scope the Data is associated with.

With the above understanding, we design our name tree with a functionality-driven principle in mind. Table 1 summarizes the structure of names and the overall namespace used by NDN-Lite Pub/Sub, and Figure 3 shows a sketch of an example name tree based on the afore-mentioned naming conventions.

The overall namespace in the example from Figure 3 starts with *alice-home* as the system's root prefix. The name tree has the branches *AUTOMATION* and *LOCK* to represent two example services of the smart home[2]. The *AUTOMATION*-service represents the control module of the smart home and aggregates all home automation application data (eg., applications that trigger a fire alarm when smoke is detected). Data of door locks is aggregated by the LOCK-service. Under the *LOCK* name tree node, the name components *livingroom* and *front* represent locations[3]. A tuple *<service, location>* can be used to specify a name tree node with which publishers and subscribers may interact. For example, Data containing self-state reports produced by the living room's front door lock can be considered under the name tree node specified by the tuple *<LOCK, /livingroom/front>* (cf. dashed red arrow).

The name tree nodes *state* and *set-lock* further differentiate Data based on application semantics (cf. the subtrees highlighted by the dashed green circle). The periodic self-state report of door lock is published with *state* as data-id. A command to control the door (eg. lock) uses *set-lock* as data-id. A *timestamp* is attached as the last name component to achieve uniqueness among all application data names. Meanwhile, the *lock-1* tree node represents the door lock's participant identifier and is represented in another subtree under the door lock's location. This subtree holds certificates and security policies for the participant *lock-1*.

The Data content produced by the door lock is encrypted by symmetric encryption keys. These keys are provided by the controller, and authorized subscribers are capable to decrypt Data with decryption keys obtained from the controller. Such encryption and decryption keys are held under the *EKEY* and *DKEY* subtree in the *ACCESS* branch of the *LOCK* service (highlighted by the dashed blue circle).

3.2 Security Policies

Security policies (Section 2.1.3) enable authorization, in addition to access control within the name tree structure. Therefore, participants are limited to create or access Data only under certain name tree nodes, specified by the policies. A security policy expresses the relationship between name tree nodes and key names, where key names can again be seen as nodes in the name tree. Formally, a security policy is defined as a triple *<DataNamePrefix, PolicyType, KeyNamePrefix>*, where the *DataNamePrefix* defines the targeted node in the name tree. The *PolicyType* can either allow *signing* data matching the targeted node, or allow *access* to a service

[1]In IP-based communication, the Pub/Sub systems we refer to here are often realized using a rendezvous node. However, as discussed later, a Pub/Sub system in NDN can be realized without such dependencies by semantically naming data.

[2]One can have more branches if having more services (e.g., temperature, motion) in the system.

[3]A location consists of zero to two name components (e.g., "/", "`/livingroom`", "`/livingroom/front`"), depending on the required granularity.

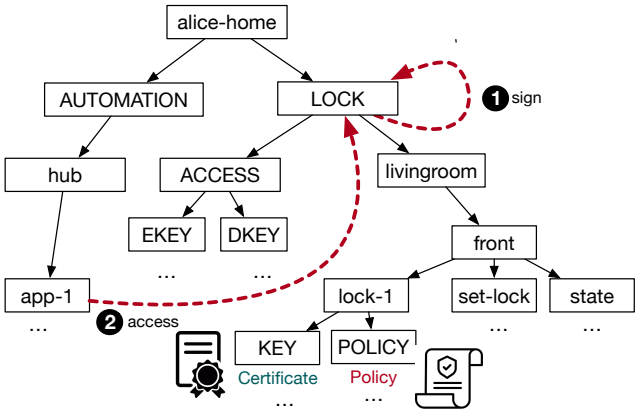

Figure 4: Security policies over name tree nodes

defined by the node. The *KeyNamePrefix* defines the prefix of key names for which the policy applies. While the definition of security policies is left to the application layer[4], NDN-Lite Pub/Sub supports integrity and authenticity protection based on security policy definitions. Each participant fetches its security policies, which include both signing and access policies from the bootstrapping step. Security policies must be signed by the controller thus verifiable with participants' installed trust anchor by looking into the signing certificate name and signature value.

The name tree in Figure 4 provides two cases as explained below to show how our Pub/Sub automatically enforces security policies leveraging names. Later, we show the realization of automated security workflow in Section 3.5.

Automatic signing policy enforcement: The security policy ❶:

</alice-home/LOCK, sign, /alice-home/LOCK>

restricts the *LOCK*-service Data to be signed only by identity keys under the *LOCK*-service prefix. The signing relationships of published Data can be automatically verified by examining the each step along the certificate chain. Later when receiving newly published Data, subscribers can extract signers' certificate name by looking the KeyLocator field of Data, then check against fetched signing type policies to determine if the signing key is authenticated to publish with received Data name. If the signing key is authenticated, subscribers fetch the signer's certificate, and the retrieved certificate is examined with KeyLocator field to see if signed by the installed trust anchor, and later the public key of installed trust anchor is used to validate the signer's certificate. After, public keys contained in the validated signer's certificate is used to verify the signature of the published Data.

Automatic access policy enforcement: The policy ❷:

</alice-home/LOCK, access, /alice-home/AUTOMATION/hub/app-1>

allows all participants holding certificates under prefix "`/alice-home/AUTOMATION/hub/app-1`" to access Data of the *LOCK*-service. Having learned the permission to access from the policy definition, "`app-1`" fetches its decryption key to get access. Guided by access policies and the identity information carried in the decryption key request, controller chooses either accepting that request by returning the decryption key or denying by returning the error code. The decryption key follows the naming convention in Table 1. After getting the Data, subscribers can extract the service name *LOCK* from the published Data's name components, and use corresponding decryption key to access Data content.

3.3 Providing API for Different Applications

After designing the name tree where all application data are organized, we now consider developing easy-to-use API to enable developers interacting with the name tree. Our API design starts with two observations on home IoT applications and ends up with unified API that can serve diverse applications.

A key observation is that IoT applications have two fundamental publishing behaviors when adopting a publish-subscribe design pattern: publish periodically, or publish at random times. Data publishing at random times are triggered events (e.g., fire alarms and commands to turn the camera off/on). The publishing of such data is primarily triggered by sudden changes in the physical environment and users' input. In contrast, periodically published data are periodic events. For example, a temperature sensor might be scheduled to report the sensing data on a minute basis. NDN-Lite Pub/Sub provides a unified API for both types of publishing behaviors by letting publishers decide their publishing modes.

A second observation is applications have various real-time requirements. For instance, the primary goal of temperature monitoring application is to collect sensor readings from various rooms, which has a relatively low real-time requirement. In contrast, the smoke detection application, which needs to promptly react to abnormal events (e.g., raise alarms as soon as possible when smoke detected), has a relatively high real-time requirement. Thus API design should respond to this difference in real-time requirements with easy-to-use interfaces.

Based on these two observations, along with the concept of name tree, NDN-Lite Pub/Sub provides API as follows:

```
Publish API: publish(service, location, data-id, content, pattern)
Subscribe API: subscribe(service, location, callback, interval)
```

[4]Eg. policy definition via the GUI of a controller, as envisioned by NDN-Lite.

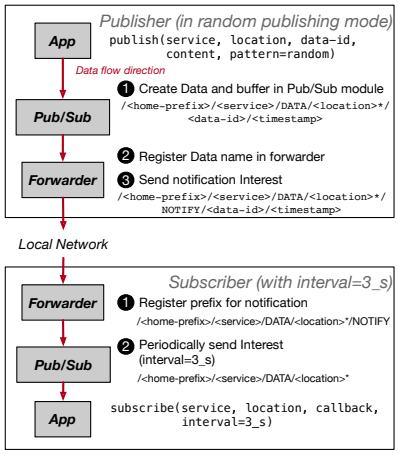

Figure 5: Application-to-application workflow showing all layers involved in the Pub/Sub process.

Publish API: The Publish API allows publishers to specify a name tree node by using *<service, location>* tuple, and attach Data with specified names (derived from the input parameters and naming convention) and *content*. It requires publishers selecting their publishing modes (either periodic or random) by *pattern* when calling.

Subscribe API: The Subscribe API allows subscribers to specify a name tree node, and expressing Interest with *callback* and set the real-time requirement by *interval*. For example, a *3_s* interval means the subscriber expects to receive data no later than 3s after its publishing time.

3.4 Protocol Design of the Pub/Sub API

While the previous section described our API's design, this section presents the proposed protocol used to realize Pub/Sub with Interest-Data exchange. Figure 5 visualizes the workflow from the application perspective, including the tasks necessary in the lower layers of the protocol stack.

We begin by describing the protocol on the publisher-side of the system. The publisher-side can choose among random and periodic publishing, with the protocol design differing only slightly. When a publisher application publishes new Data using the *publish API*, the *service* and *location* parameters define the Data's name prefix. The *data-id* is used as an additional name component to uniquely identify the new publication. The API creates and buffers the Data locally and registers the Data's exact name at the forwarder.

In *random publishing* mode (e.g., used for sending control commands to devices), the publisher generates a Notification Interest to notify all subscribers about the new publication. The Notification Interest's name starts with the prefix defined by the *service* and *location* parameters and adds a `/NOTIFY`-suffix, followed by the exact name of the published Data. This Interest name allows subscribers to fetch the randomly published Data on receipt of the Notification Interest. To preclude packet loss, Notification Interests are repeated until acknowledged by at least one subscriber[5].

In *periodic publishing* mode (e.g., used when producing sensor data), the Notification Interest is omitted. In this case, periodically sent Subscribe Interests from subscribers (explained in the following) are used to retrieve published Data.

When focusing on the subscriber-side of the system, subscriber applications express interest to Data under a name tree node by using the *service* and *location* parameters of the *subscribe API*. Besides, the application provides a callback method that defines the application's action on receiving a publication. The *interval* parameter defines the frequency the applications want to receive updates. Calling the subscribe API triggers two actions: First, to receive Data, the subscriber emits periodic *Subscribe Interests* holding the *service* and *location* parameters as name prefix. The configured interval defines the frequency of those Subscribe Interests[6]. Whenever published Data matches the *service* and *location* parameters, the Subscribe Interests initiate delivery of thereof. Second, the *subscribe API* configures the forwarder to listen to Notification Interests matching the *service* and *location* parameters. Therefore, the name prefix holding the *service* and *location* parameters is registered with the suffix */NOTIFY* appended. This enables the Notification Interests sent by publishers in random publishing mode be forwarded to the subscriber applications.

Note that subscribers do not need to know the publisher's publishing behavior ahead of time. A subscriber may receive a Notification Interest that enables the subscriber to immediately fetch randomly published Data, and additionally, actively pulls periodic publications by emitting Subscribe Interests. Active pulling, however, might not be reasonable in all application use cases (eg. pulling infrequent control commands) and, hence, can be disabled, eg. by setting the *interval* parameter to zero. In this case, Data transmission is triggered by Notification Interests only.

3.5 Security Workflow

The NDN-Lite Pub/Sub API provides built-in security support for applications by automating the security workflows required for Data publishing and subscribing. In this section, we demonstrate the NDN-Lite Pub/Sub's security workflow by an example (cf. Figure 6), where an application ("`/alice-home/AUTOMATION/hub/app-1`") running on a home controller subscribes to the self-state reports of the *LOCK*-service at the living room's front door. We assume that the

[5]A pre-set maximum number of repetitions prevents infinite notifications in the case of zero subscribers. When no acknowledgment is received, a failure is reported to the calling application.

[6]Different subscriber applications may have different intervals set

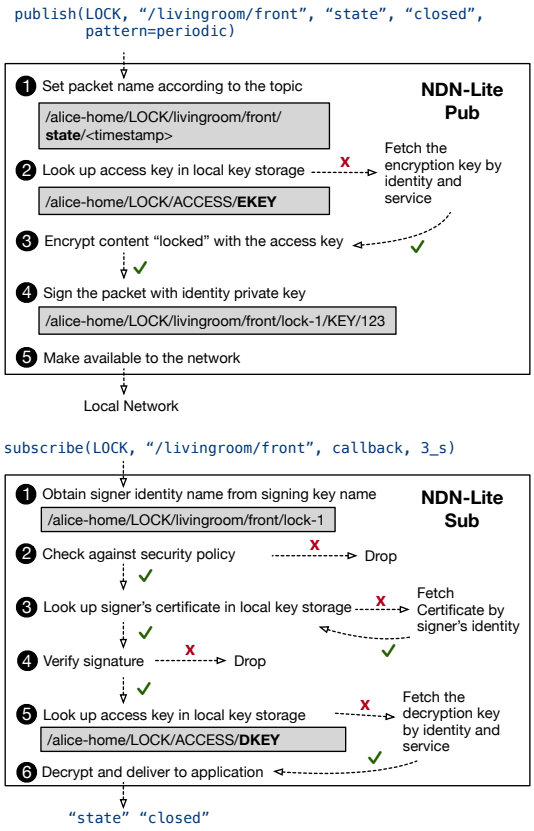

Figure 6: Subscribe to periodically published Data

security policies ❶ and ❷ from Figure 4 were retrieved by the application during the bootstrapping phase.

The participant *lock-1* periodically produces the self-state report by calling the publish API with *state* as *data-id* and the current status (eg. open, or closed) as value. The Pub/Sub module constructs the Data name following naming conventions, retrieve the encryption key, sign the Data packet with the private key of its identity, and make it available on the network.

The *app-1* subscribes to Data associated with *LOCK*-service and */livingroom/front* location with a callback and interval being 3_s. Upon receiving new Data, *app-1*'s Pub/Sub module checks security policy ❶ to examine if allowed keys sign the Data. Its Pub/Sub module will then verify the signature carried in received Data against the fetched signer's (i.e., *lock-1*) certificate. Afterwards, the Data content is decrypted by Pub/Sub using corresponding decryption key since the security policy ❷ indicating *app-1* has permission to access. Pub/Sub module will only deliver received Data to the application if it passes all security checks. In this example, data-id *state* and content *closed* is the final result delivered to the callback.

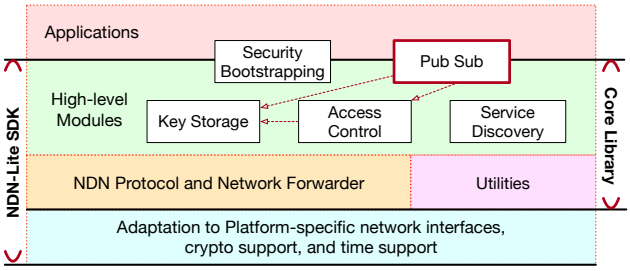

Figure 7: NDN-Lite Architecture

Note that the use of application names for data and keys enables NDN-Lite Pub/Sub, and the security workflow is automated based on the reasoning of relations between names.

4 IMPLEMENTATION

While the NDN-Lite Pub/Sub protocol's design is not limited to specific application use-cases, the current implementation focuses on IoT settings. NDN-Lite Pub/Sub is realized as part of the NDN-Lite framework[7]. The focus of the codebase is supporting constrained devices as well as conventional computers as execution platforms. Therefore, the implementation is built on C and uses static memory allocation only. To use NDN-Lite Pub/Sub in home IoT development, developers can download the NDN-Lite SDK (see Figure 7), build their applications with our API (Section 3.3), and flash the compiled binaries to their IoT devices. Figure 7 visualizes the structure of NDN-Lite and shows how individual components work together. NDN-Lite *Pub/Sub* and *security bootstrapping* are the only two components exposed to the application and make primitives like Interest-Data exchange, or the use of cryptography transparent to application developers. However, NDN-Lite Pub/Sub assumes that the security bootstrapping (eg. setup of the trust anchor and the definition of security policies) has been completed before use. NDN-Lite Pub/Sub embeds security in terms of integrity protection and authenticity. For accessing the required cryptographic keys, the modules *access control* and *key storage* of the NDN-Lite framework are used. These components are based on the concept of name-based access control and securely retrieve keying material if not yet available (cf. Section 2.1.3).

5 MEMORY AND LATENCY OVERHEAD

In this section, we evaluate NDN-Lite Pub/Sub's performance regarding latency and memory overhead. The results show that NDN-Lite Pub/Sub only has a small memory usage and acceptable latency in communication. These results mean that it can effectively operate on constrained devices.

Latency of Common Operations: To assess the latency of NDN-Lite Pub/Sub, the execution time of two typical home IoT functions is compared to the popular IoT platform AWS

[7]https://github.com/named-data-iot/ndn-lite

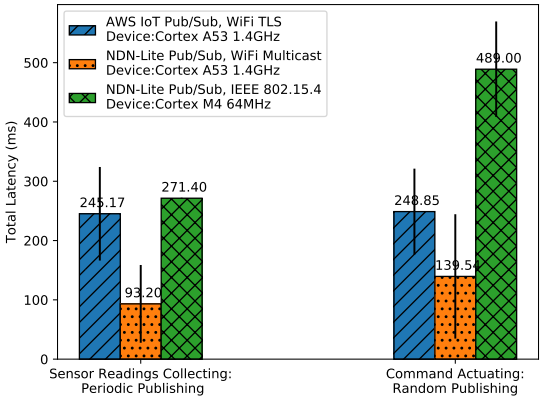

Figure 8: User-perceived Latency of Common Operations in NDN-Lite Pub/Sub and Amazon AWS IoT Pub/Sub

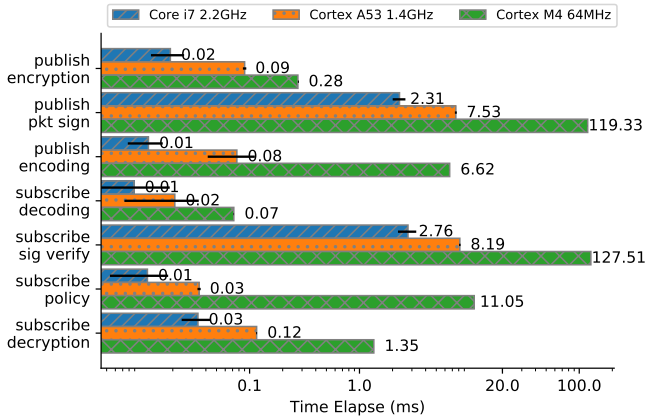

Figure 9: Breakdown of the execution time into individual operations in NDN-Lite Pub/Sub

IoT [1]. We measure the latency for collecting sensor readings, and for actuating commands in both frameworks. In AWS IoT, collecting sensor reading is implemented by one device subscribing to a pre-defined MQTT [4] topic under which another device publishes data. In the NDN-Lite implementation, two devices, a publisher and a subscriber are running in the same room. The publisher is evaluated in random publishing mode for actuating commands and in periodic publishing mode for collecting sensor readings. The subscriber employs an interval setting of 4 seconds.

The comparison between NDN-Lite Pub/Sub and AWS IoT was conducted on a Raspberry Pi 3B (ArM Cortex A53 @1.4GHz). The results, visualized in Figure 8, show that NDN-Lite Pub/Sub reduces latency by 62% on collecting sensor readings, and by 42% on command actuating. This is because packets are exchanges locally, instead of exchanged over cloud servers. Further, NDN-Lite broadcasts packets to the local network. The local broadcast means that the sender directly populates data to the receiver without going through the broker first.

In addition, we evaluated NDN-Lite Pub/Sub's latency overhead on the resource-constrained nRF52840 chip (Arm Cortex M4 @64MHz) with IEEE 802.15.4 link-layer broadcast [14]. Due to limited computation power, the latency on the nRF52840 is larger compared to the more powerful Raspberry Pi. However, a delay of 0.5 seconds for Pub/Sub operations seems acceptable for common smart home scenarios.

Execution Time Breakdown: A breakdown of Pub/Sub's operation time into individual operations is provided in Figure 9. The breakdown is shown for devices having varying computation power. The visualized operations include data preparation before broadcasting to the network and data processing after receiving. This includes digital signature signing/verification (ECDSA), content encryption/decryption (AES-128 CBC), checking security policies (regex match), and NDN packet encoding/decoding. The results show that asymmetric cryptography consumes most of the computation time. This trend is observable across all evaluated devices.

ROM and RAM Footprint: We measured NDN-Lite Pub/Sub's memory overhead on nRF52840 boards with 1MB ROM, 250KB RAM. In terms of memory configuration, such boards represent typical hardware setups found in smart home devices.

Program/Modules	ROM Use	RAM Use
Subscriber in total	**62KB**	**47.3KB**
Publisher in total	**52.4KB**	**38.2KB**
Application	1.8%	7.3%
High-level Modules	**20.7 %**	**34.2 %**
Utilities	3.3%	14.4%
Crypto Tools	25.1%	0.2%
Network Forwarder	24.1%	25.0%
OS and Adaptation	25.1%	18.9%

Table 2: ROM and RAM Consumption

Table 2 shows the RAM and ROM footprint of a publisher and a subscriber implementation deployed on the RIOT operating system [3]. We detail the memory and flash usage of individual modules of NDN-Lite. Pub/Sub is considered as a high-level module. When breaking the memory usage down to module granularity, these high-level modules account for 20.7% of the total RAM usage and 34.2% of the total ROM usage. In total, the publisher and subscriber implementations require below 70KB of memory and flash space. This indicates the ability to use NDN-Lite Pub/Sub on resource-constrained devices in smart home environments.

6 RELATED WORK

There is only a few research focusing on providing high-level APIs for NDN. The *Consumer-Producer API* [10] presents a generic communication API for NDN applications. The API provides multiple functions, such as reliable data transmission, packet reassembly, and packet ordering. However, this work focuses on data transmission functionality. The Consumer-Producer API treats security as a plug-in solution. Thus, additional efforts from the developer-side are required to secure Data.

The *Named Data Networking Common Name Library* (NDN-CNL) [13] offers an abstraction called *Generalized Object*. The Generalized Object abstraction allows creating mutable application objects (eg., image, video segment) over immutable network layer Data packets. The NDN-CNL library synchronizes the namespace of Generalized Objects with NDN synchronization protocols [7]. NDN-CNL develops the name tree concept to organize application data. NDN-Lite Pub/-Sub adopts this name tree concept. In NDN-CNL's design, applications can attach/fetch a Generalized Object to/from a specific name tree node. Therefore, applications interact with the name tree over high-level mutable objects instead of via low-level Interest-Data exchange. NDN-CNL includes security-related operations for applications, such as Data signing and encryption. However, compared to NDN-Lite Pub/Sub, security in NDN-CNL is not natively supported and depends on applications providing hooks to perform security measures.

The *Distributed Network Measurement Protocol* (DNMP) [11] is a distributed measurement framework for NDN. It offers high-level APIs for conducting specific measurement commands (e.g., probe an interface status). DNMP's API is realized by a system component called *Shim* and automates the security workflow with the runtime library called *VerSec*. Trust schemas, specified in a declarative language, are loaded into VerSec during runtime and enable VerSec to validate Data. When signing, VerSec locates appropriate signing keys acceptable to the system's trust schema. For verification, VerSec uses the trust schema to verify that the signature was created using an authorized signing key for the given Data packet's name. DNMP uses the sync protocol *syncps* for realizing as a lightweight publish-subscribe protocol. This protocol enables topic-based communication between publishers and subscribers. The syncps module announces the set of currently available publications in an Interest containing an invertible bloom filter [6]. This data structure allows consumers to infer whether new publications are available or not. In contrast, subscribers in NDN-Lite are informed about the availability of new publications by either periodically emitting Subscribe Interests (Section 3.4), or by listening to Notification Interests that directly reveal the exact names of published Data.

7 SUMMARY AND FUTURE WORK

This paper presents NDN-Lite Pub/Sub – a high-level publish/subscribe API for applications with security built-in. Security is embedded by using name-based access control. Application semantic in names allows defining security policies based on application names and enforcing fine-granular rules for access control. NDN-Lite Pub/Sub's solutions for encryption and authentication run on resource-constrained devices with strict computation power limits. This makes NDN-Lite suitable for a decentralized Internet of Things deployment. Future work includes improving reliability by adding networked Data repositories. Repositories reduce the memory overhead and make it feasible to run NDN-Lite Pub-/Sub on ultra-constrained devices that possess less than 32KB of RAM (eg., SAMR21-XPRO). Further, we plan to compare NDN-Lite Pub/Sub's memory and latency overhead with other existing platforms, such as Azure IoT Edge [9] and AWS Greengrass [2]. Besides, a comparison between NDN-Lite Pub/Sub's energy consumption and regular IP-based solutions is planned.

ACKNOWLEDGMENT

This work is partially supported by the National Science Foundation under awards CNS-1629922 and CNS-1719403.

REFERENCES

[1] Amazon Inc. 2020. AWS IoT, Connected Home. https://aws.amazon.com/iot/solutions/connected-home/. Accessed: 2020-12-6.

[2] Amazon Inc. 2020. AWS IoT Greengrass. https://aws.amazon.com/greengrass/. Accessed: 2020-12-6.

[3] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*. IEEE, 79–80.

[4] Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. 2019. MQTT Version 5.0. *OASIS Standard. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html* (March 2019).

[5] Matt Cooper, Yuriy Dzambasow, Peter Hesse, Susan Joseph, and Richard Nicholas. 2005. Rfc 4158-internet x. 509 public key infrastructure: Certification path building. *Online at ftp://www.ietf.org/rfc/rfc4158.txt* (2005).

[6] David Eppstein, Michael T Goodrich, Frank Uyeda, and George Varghese. 2011. What's the difference? Efficient set reconciliation without prior context. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 218–229.

[7] Tianxiang Li, Wentao Shang, Alex Afanasyev, Lan Wang, and Lixia Zhang. 2018. A brief introduction to ndn dataset synchronization (ndn sync). In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 612–618.

[8] Y. Li, Z. Zhang, X. Wang, E. Lu, D. Zhang, and L. Zhang. 2019. A Secure Sign-On Protocol for Smart Homes over Named Data Networking. *IEEE*

Communications Magazine 57, 7 (July 2019), 62–68. https://doi.org/10.1109/MCOM.2019.1800789

[9] Microsoft Inc. 2020. Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge/. Accessed: 2020-12-6.

[10] Ilya Moiseenko and Lixia Zhang. 2014. Consumer-Producer API for Named Data Networking. In *Proceedings of the 1st ACM Conference on Information-Centric Networking*. 177–178.

[11] Kathleen Nichols. 2019. Lessons learned building a secure network measurement framework using basic ndn. In *Proceedings of the 6th ACM Conference on Information-Centric Networking*. 112–122.

[12] Sanjeev Kaushik Ramani, Proyash Podder, and Alex Afanasyev. 2020. NDNViber: Vibration-Assisted Automated Bootstrapping of IoT Devices. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 1–6.

[13] Jeff Thompson, Peter Gusev, and Jeff Burke. 2019. Ndn-cnl: A hierarchical namespace api for named data networking. In *Proceedings of the 6th ACM Conference on Information-Centric Networking*. 30–36.

[14] Wireless Personal Area Network (WPAN) Working Group. 2016. IEEE Standard for Low-Rate Wireless Networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)* (April 2016), 1–709. https://doi.org/10.1109/IEEESTD.2016.7460875

[15] Yingdi Yu, Alexander Afanasyev, David Clark, KC Claffy, Van Jacobson, and Lixia Zhang. 2015. Schematizing trust in named data networking. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking*. 177–186.

[16] Yingdi Yu, A Afanasyev, Z Zhu, and L Zhang. 2014. Ndn technical memo: Naming conventions. *NDN, NDN Memo, Technical Report NDN-0023* (2014).

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

[18] Zhiyi Zhang, Edward Lu, Yanbiao Li, Lixia Zhang, Tianyuan Yu, Davide Pesavento, Junxiao Shi, and Lotfi Benmohamed. 2018. NDNoT: a framework for named data network of things. In *Proceedings of the 5th ACM Conference on Information-Centric Networking*. 200–201.

[19] Zhiyi Zhang, Yingdi Yu, Sanjeev Kaushik Ramani, Alex Afanasyev, and Lixia Zhang. 2018. NAC: Automating access control via Named Data. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 626–633.

[20] Zhiyi Zhang, Yingdi Yu, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang. 2018. An overview of security support in Named Data Networking. *IEEE Communications Magazine* 56, 11 (2018), 62–68.