Decentralized and Secure Multimedia Sharing Application over Named Data Networking*

Ashlesh Gawande, Jeremy Clark, Damian Coomes, Lan Wang†
University of Memphis
{agawande,jclark2,dmcoomes,lanwang}@memphis.edu

Abstract

Named Data Networking (NDN) thrives in peer-to-peer data sharing scenarios, through naming data and decoupling data from its containers. Meanwhile, social media applications have come under increased criticism for excessive centralization and opacity. We present npChat, an Android application that allows users to capture and share multimedia with friends in a secure and fully decentralized way, while still giving users complete control over their data. We propose using namespaces owned by users instead of a shared application namespace and establish trust using multiple trust models. We use an application-level pub-sub model to share friend information and publish data, as well as a per-object access control scheme to share content with selected friends. Our evaluation demonstrates the application's data sharing performance and low overhead in various scenarios.

CCS Concepts

• **Networks** → *Social media networks*; *Mobile ad hoc networks*; • **Human-centered computing** → *Mobile phones*;

Keywords

Named Data Networking, Decentralized Social Media Applications, Information Centric Networks

ACM Reference Format:
Ashlesh Gawande, Jeremy Clark, Damian Coomes, Lan Wang. 2019. Decentralized and Secure Multimedia Sharing Application over Named Data Networking. In *ICN '19: Conference on Information-Centric Networking, September 24–26, 2019, Macao, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3357150.3357402

1 Introduction

The Named Data Networking (NDN) architecture has great potential in improving the robustness and performance of data distribution, especially when the network environment is challenging and when peer-to-peer data sharing is desirable. For example, in comparing HTTP and NDN content retrieval, NDN shows promise in lossy environments, performing 4 times better than the HTTP implementation [32]. As another example, nTorrent [18], an NDN version of BitTorrent, showed nearly a 50% reduction in network traffic and faster download speeds compared to BitTorrent.

While performance improvement is an important reason for developing the NDN architecture, we believe an equally important reason is that NDN has the necessary architectural building blocks, such as retrieving data using names and schematized trust based on data names, to support fully decentralized applications. Removing the need to rely on a central entity to store and serve data, provide user information, and control access to content will significantly improve users' privacy and security, and prevent the dominance of a few application service providers which is common today.

We believe more research effort needs to be focused on designing a framework and the specific mechanisms to support decentralization in NDN applications. To this end, we designed an NDN application named npChat (NDN Photo Chat) that provides similar functionality as today's media-sharing based social networking applications without requiring any centralized service providers, and implemented it on the Android platform over NFD Android [21]). This application-driven exercise aims to shed some light on various aspects of developing a fully decentralized application. The major contributions of this work include identifying the specific requirements for a fully decentralized application, and designing and implementing NDN-based mechanisms to enable users to discover other users in the local network and through mutual friends, build friendship via multi-modal trust establishment mirrored from the real world, subscribe to friends' multimedia data updates via pub-sub, and control access to their own published media. We hope that the design patterns in this application will provide a reference for developing other decentralized applications.

2 Design

A good NDN application for social networking needs to include the basic functionality that users have come to expect from such an application. However, our goal is not to simply replicate the functionality, but rather to break away from the typical centralized design and implementation of such applications by taking advantage of NDN's architectural features (Section 2.1). Therefore, npChat provides the major functions that are common in popular social networking applications such as Facebook, Instagram, SnapChat, and WeChat (Section 2.2), but *it realizes these functions without relying on a single shared namespace, a centrally managed data service, a single user directory, a specific trust model/anchor, or a single access control point*, as we will describe in Section 2.3 to Section 2.9.

*This work was supported by NSF CNS-1629769.
†Damian Coomes has graduated from the University of Memphis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICN '19, September 24–26, 2019, Macao, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6970-1/19/09…$15.00
https://doi.org/10.1145/3357150.3357402

2.1 Requirements

Our goal is to ***achieve a fully decentralized application design in terms of data storage, communication, trust establishment, and access control***. Below we elaborate our design requirements derived from this goal.

First, since giving users, not third parties, exclusive control over their own data is important for ensuring the users' privacy and other rights, **the application should enable each user to store and serve his/her data**. In other words, there is *no single entity* that collects and serves all the users' data (even if such a service is provided from many distributed data centers). Note, however, that the above requirement does not mean that all the data produced by a user must be stored in the user's phone. In fact, *the user can set up a personal repo for better data availability and persistence*. By "personal repo", we mean either a persistent storage device owned by the user running NDN repository software, or such a storage service hosted by a provider chosen by the user, which is independent of whoever provides the application.

Second, **the application should not rely on a user directory maintained by *a single entity***, which contains all users' information, even if the directory is stored in a distributed manner. Having such a directory at the core of the application means that, when the directory is unavailable, the application will not function properly. Moreover, the entity that maintains the directory can easily infer users' activities as it is queried by users to find new friends and the directory may contain additional user-specific information, e.g., location and time when a user is online.

Third, **the application should not depend on specific infrastructure support, other than the basic NDN forwarding functionality, for transferring its messages**. For example, there should be no specialized nodes or application-specific state in the network to forward the application's messages. Furthermore, the application should be able to operate in an ad hoc environment, e.g., over WiFi Direct, as long as the devices support such communication modes. The less our application depends on specialized routers, peers, or other infrastructure components, the more robust it becomes in face of network and user dynamics.

Fourth, **our design should allow users to establish trust relationships and authenticate each other's data without relying *solely* on a pre-defined trust anchor**. In the real world, we often derive our trust from a variety of sources, such as authorities defined by existing management structures, reputation reported by a trusted agency based on past observed behavior (e.g., a credit score), and judgement of people we trust. Some applications can work very well using a single trust source, while others may benefit from multiple sources especially when the operating environment may change dynamically over time.

Finally, **access to a user's content should be controlled by the user, not another entity**. The user should be able to control who among his/her friends can access a piece of his/her content without relying on a third party to enforce this access restriction.

2.2 Overview

Our application supports the following functionality. **First**, an npChat user can make a new friend by scanning another user's identity information (encoded in a QR code). He/she can also send a Friend Request to a remote user who can either accept or reject the request. **Second**, one can discover new users to befriend by finding other npChat application instances on the local area network and collecting each friend's Friend List. **Third**, a user can share content, such as photos and videos, with his/her friends. This may be called a story, feed, or some other name depending on the specific application. **Fourth**, such content may be restricted to specific friends, and it may be accessible only for a certain time period.

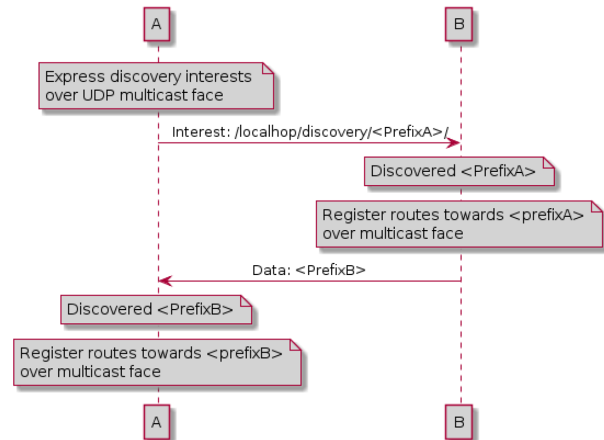

Figure 1: Discovering npChat Users via Multicast Face

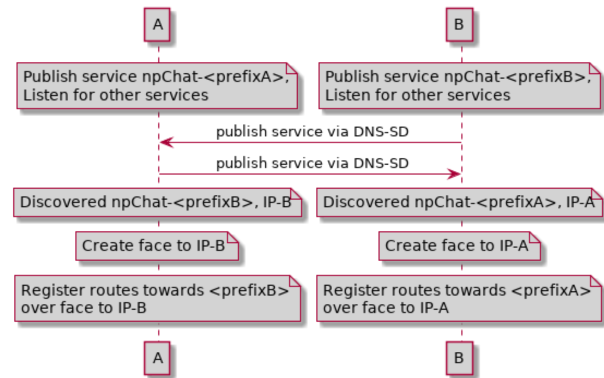

Figure 2: Discovering npChat Users via DNS-SD

2.3 Local Active User Discovery

Our application discovers other npChat instances running in the same local network. This discovery process serves three purposes: (1) after discovering any active friends, npChat subscribes to those friends' feeds so that it can retrieve their published data and receive any future data; (2) if the local network does not support multicast, then the discovery process creates unicast links and routes toward the active friends; (3) if any discovered user is not a friend yet, that user is added to a list of potential friends (see Section 2.6 about making new friends). As we will show in this section, our design avoids relying on a central server for discovering and storing all users to the extent possible.

Typically a wireless interface can be associated with either an Access Point or Wifi Direct but not both at the same time[1], so we outline the discovery process in each case below.

WiFi Direct In the WiFi Direct mode, NFD Android uses the “NDN Over WiFi Direct Protocol" [10] to discover all the name prefixes registered at the other NFD instances in the same WiFi Direct network and establishes appropriate unicast routes to them. This means that NFD will be able to route to any other npChat user’s data in the WiFi Direct network via unicast routes. However, as the discovery protocol in NFD Android is opaque to the application, the application still needs to identify the specific npChat users and their name prefixes, so we still run the same user discovery process as outlined below.

Infrastructure-based Local Area Network NDN currently does not have a standard local discovery mechanisms for applications, so we decided to develop our own mechanisms. There are three possible cases. ***First***, in networks that support multicast traffic to NFD’s port, npChat first tries to discover other users through NFD’s multicast face directly and use it for subsequent communication. As seen in Figure 1, the app sends Interests with the user’s prefix over this face at startup in the namespace /localhop/discovery. Upon receiving this Interest from another user, npChat responds with Data containing its user’s prefix. The probing user’s prefix is stored and any routes to that user are created over the multicast face. ***Second***, many networks block multicast traffic except for a few important ports so the above mechanism will not work. However, some of them allow multicast for network service discovery based on the DNS-SD mechanism [6], as printers and other devices depend on DNS-SD to publish their services. Therefore, we have incorporated a discovery mechanism based on DNS-SD in our app, which is illustrated in Figure 2. ***Third***, for networks that block all multicast traffic, we plan to use NDN Neighbor Discovery Service (NDND) [23] which allows NDN nodes to discover each other through a rendezvous server.

One obvious question remains for discovering users not in the local network. As we will present in Section 2.6, each npChat user **optionally** shares a Friend List containing the latest information (e.g., name prefix, key/certificate, and online/offline status) about that user’s friends. This is one way for that user’s friends who subscribe to this Friend List to learn about other users that may not be in their local network (note that only this user’s friends can subscribe to the Friend List). Alternatively, it is possible to use the NDND service to collect all the npChat user information globally, but this will lead to a centralized user list that we want to avoid (Section 2.1), so we will use NDND only in a local area network for discovering local users. We will offer more justification for our decisions in Section 2.6.

Note that for this work, we assume that if a user wants to communicate with users not currently on the local network, the user’s device needs to connect to an (overlay) NDN wide area network, e.g., the NDN testbed. As such, npChat does not create tunnels to those users directly, but relies on NFD to establish the necessary routes toward the wide area network to reach them.

[1]WiFi Ad Hoc mode requires rooting an Android phone, so it is not supported by NFD Android.

2.4 Naming

A straight-forward design for naming user data is to put a common application name prefix, e.g., `/com/npChat`, before the user name. However, such a design requires some entity to apply for and manage this application namespace, which goes against our goal of decentralization. Therefore, our design lets each user publish his/her npChat data under a namespace owned by the user. For example, suppose Jane Doe on AT&T’s network owns the namespace `/net/att/JaneDoe`, and she chose the user name `janedoe123` when she first started npChat, then her content will be published under `/net/att/JaneDoe/npChat/janedoe123`. As another example, John Smith at U. Memphis (`/edu/memphis/JohnSmith/`) with the npChat user name `jsmith` will have the name prefix `/edu/memphis/JohnSmith/npChat/jsmith` for his data.

Every user needs a public/private key pair for signing his/her npChat data, and the key name is under the user’s npChat namespace. If we use `<UserPrefix>` to represent a user’s namespace for npChat (e.g., `/net/att/JaneDoe/npChat/janedoe123`), then the user’s npChat key will be named `/<UserPrefix>/KEY/<KeyId>`. Section 2.5 describes how the users authenticate each other’s key to establish a trust relationship.

In addition to the above namespaces, we also need to have namespaces for the PSync library [34] to sync users’ data. We will present the naming scheme for PSync in Section 2.7.

2.5 Trust Model

In order to verify the authenticity of received data, an npChat user must be able to verify the authenticity and signing privilege of the key that signed the data, i.e., the key indeed belongs to the user that claimed to have generated the named data and the key is authorized to sign the data. The “trust” in this key’s authenticity and signing privilege is typically derived using a trust model specified in the application [30]. Our application requires npChat data to have the name format `/<prefix>/npChat/<user>/...` and it must be signed by the user’s npChat key named `/<prefix>/npChat/<user>/KEY/<KeyId>`. The question is how to authenticate this npChat key. A common trust model is a hierarchical one with a pre-defined trust anchor (e.g., NLSR’s trust model for verifying routing data [29]). This type of hierarchical trust model can be used by AT&T npChat users to verify Jane Doe’s key. More specifically, Jane’s npChat key should be signed by her AT&T namespace key, which in turn should be (recursively) signed by the key belonging to AT&T’s namespace management authority (i.e., the trust anchor). However, John Smith with a namespace from U. Memphis may not be able to authenticate Jane’s AT&T key if he does not have or trust the AT&T trust anchor. In this case, a web-of-trust-like model such as [31] may work better by allowing the users to endorse other members with various trust levels.

We believe that for a social networking application with many users from different communities, rather than letting the application choose a specific trust model, it is better to allow users to establish trust relationships using any trust models and trust sources they have available, as different models/sources may work in different situations. For example, Jane Doe and another npChat user with an AT&T namespace can use their AT&T trust model to establish trust, while Jane Doe and John Smith may be able to

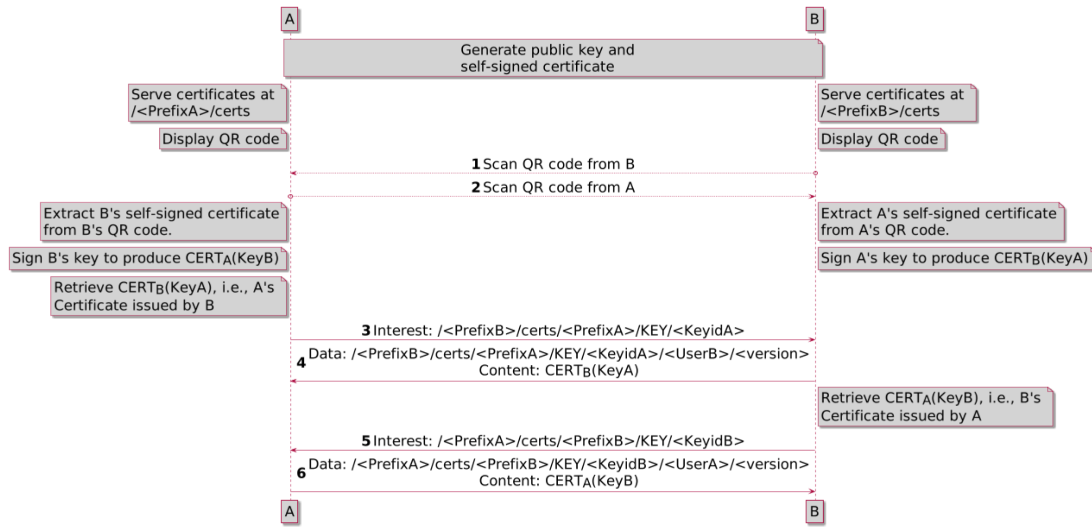

Figure 3: Procedure for Establishing Friendship Relationship in Person

establish trust through a web-of-trust-like model. This multi-modal approach closely resembles the trust establishment in a human world, which helps to grow a social network and makes it resilient. For this reason, ***we study how to establish trust through multiple means in this work***. More specifically, we incorporate the following mechanisms in our design.

(1) **personal verification**: two users meet in person to verify each other's identity and obtain each other's key after the verification;
(2) **hierarchical trust**: each user can supply one or more hierarchical trust schemas with fixed trust anchor(s) to the application. For example, John Smith can provide U. Memphis's trust schema through configuration to the application, which can be used to verify the keys from npChat users at U. Memphis;
(3) **mutual-friend based trust**: if two users have a mutual friend in npChat, then they can establish trust. For example, if Jane Doe and John Smith are both friends of Richard Roe, then they can trust each other's key after verifying the corresponding certificates issued by Richard to Jane and John (Section 2.6).

Note that the objective of this paper is not to propose new trust models, but rather to investigate the feasibility of supporting multiple trust models and the associated complexities. Therefore, the **novelty** of this work lies in the *process* for the application to establish trust with the uncertainty of multiple trust models and certificates per user (Section 2.6). In our future work, we will consider more complex trust models such as *partial* trust based on the number of mutual friends.

2.6 Making Friends

As introduced in Section 2.5, we experiment with three types of trust in npChat, and in this section, we describe how users make friends using these trust models. Before proceeding further, we want to clarify the distinction between the two terms "trust" and "friendship" in this work. Trust here refers to the acceptance of some key/data after applying the relevant authentication method, while friendship refers to two users' willingness to connect with each other to share data in the application. *Friendship is built on trust, but trust does not require friendship.* Once two users accept their friendship, they issue certificates to each other by signing the new friend's key. Note that issuing a certificate is a very basic operation in NDN, as every user signs every piece of data (including keys) that he/she produces.

We first present the procedure for two users to establish friendship in person (see Figure 3). In this case, users A and B can scan each other's QR codes containing their self-signed certificate to become friends. Upon scanning B's QR code, A signs B's key to issue B a certificate ($CERT_A(KeyB)$) which confirms their friendship. It then begins serving the certificate under the name /<prefixA>/certs/<PrefixB>/KEY/<KeyIdB> for B to fetch. In other words, A encapsulates B's certificate using A's name prefix so that B's Interest for the certificate will arrive at A instead of being consumed by B. Similarly, B issues A a certificate to confirm A as a friend. They then fetch the newly issued certificate from each other. This series of exchanges establish a friendship relationship between the two users so that each can subscribe to the other's data to receive all the data published in the other's feeds (Section 2.7). Each user can also select the other as a recipient when sharing access controlled content (Section 2.8).

Before we present the procedure for two users who cannot meet in person to become friends, we need to first explain how one can discover other users to befriend. In the simplest situation, one may get the other user's information (e.g., name prefix) through an out-of-band channel, e.g., a phone call. Otherwise, one can discover other users in the local network using the discovery mechanisms described in Section 2.3. In addition, each user **optionally** publishes information about their own friends in a feed called "Friend List". This way, user A, who is a friend of user B, may be able to find user C who is a friend of B but not a friend of A yet. Each user maintains a list of potential friends using information collected by the local discovery mechanisms and that from his/her friends' Friend Lists.

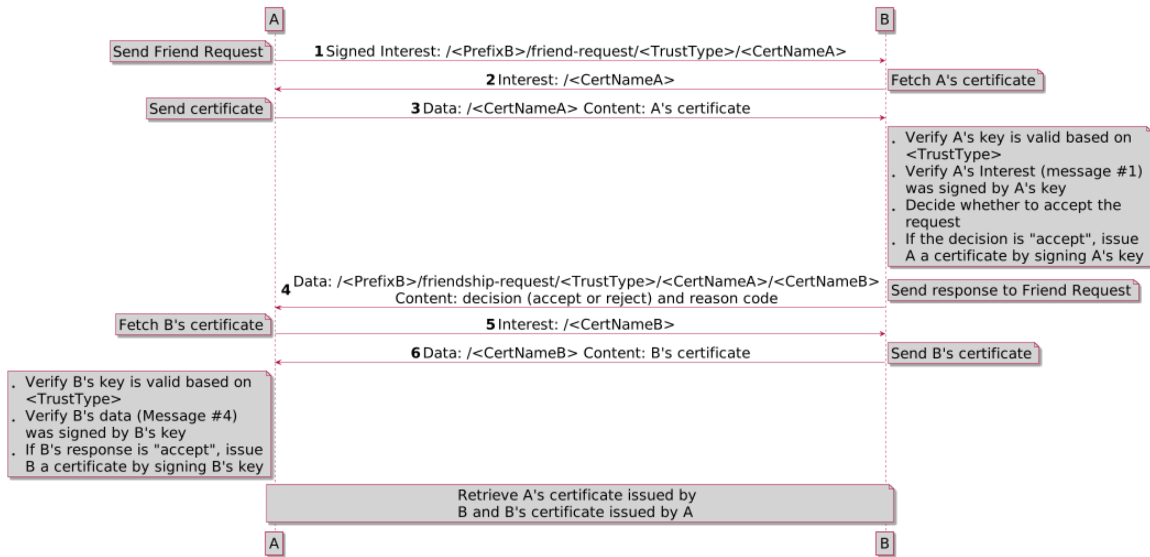

Figure 4: Procedure for Establishing Friendship Relationship over a Network

This list will increase over time as a user gains more friends and visits more places.

Although the above does not guarantee the discovery of every npChat user, this is unlikely to be a major problem because there are usually enough people to befriend within two social-network hops and within the physical local network (note that this set expands as one gains more friends). In fact, Mislove et. al. [19] studied the growth of the Flickr network and discovered that ***over 80% of new links in the network connect users that were only two hops apart***, meaning that friend-of-a-friend was the major source of new links in a social network. Moreover, one can meet new npChat friends in person and have out-of-band mechanisms to share potential friends' information. We believe that, compared to using a server to collect all user information centrally, our design decision is a reasonable trade-off for the various benefits gained from decentralization.

Now we present the procedure for two users A and B connected over a network to establish friendship remotely (see Figure 4). A first sends a Friend Request to B using the Interest /<PrefixB>/friend-request/<TrustType>/<CertNameA> signed by A's key. This signature prevents other users from pretending to be A. Note that because each user may have multiple certificates issued by different trust sources, there is a name component <TrustType> in the Friend Request to specify which type of trust source A wants to use and a name component <CertNameA> to include the name of the corresponding certificate. When B receives the Friend Request from A, he/she first retrieves A's certificate if it has not been retrieved before. Then B uses the specified TrustType to validate A's key using A's certificate and A's Interest using A's key, decides whether to accept the request, and then responds to A's request with a Data packet signed by B's key. Similar to A's Friend Request, B's response contains the TrustType and B's certificate name for A to verify the response. If B's response is valid and the decision is to accept A's request, then they will proceed to retrieve the new certificate issued by each other.

Below we explain how A may choose the TrustType for establishing trust with B. If A and B have the same namespace, e.g., AT&T namespace, then A can use "hierarchical trust" as the TrustType and include his/her AT&T issued certificate in the request. On the other hand, if A and B do not have a shared name space, but A learned of B from a mutual friend C, then A can use "mutual friend" as the TrustType and include the certificate C issued to A in the request.

Note that, instead of a certificate name in the Friend Request and Response, A and B can use a certificate bundle [27] if the certificate validation requires the retrieval of multiple certificates all the way to the trust anchor. This may be useful when the nodes are operating in an environment where it is difficult to retrieve those other intermediate certificates. Another possible optimization is to include A's entire certificate in the Friend Request as an Interest name component, but this change will make the Interest and Data names much longer. Alternatively, the certificate can be an Interest parameter, which will not lengthen the Interest name.

2.7 Data Publication and Subscription

Our application uses a pub-sub model where a user publishes his/her content in multiple feeds, e.g., photo/video, location, Friend List, and each friend of the user can subscribe to one or more of the feeds. *We use the partial sync mode in PSync [34] to realize this pub-sub functionality.*

Most existing ICN pub-sub designs, e.g., COPSS [5] and HoPP [13], put subscription state in the network through routing or forwarding. In contrast, PSync pub-sub keeps state only in end nodes. More specifically, the PSync library uses name components in Interest/-Data messages to encode both subscribers' subscription information and producers' data set state, so that such information can be carried in PSync packets without using routing or forwarding state. More specifically, PSync uses an Invertible Bloom Filter [11] to represent the state of a data set, which allows efficient set difference determination. In the partial sync mode, it also uses a Bloom Filter

to represent a subscriber's subscription list. This approach suits edge applications especially well as the intermediate nodes may be mobile or transient, making it difficult to keep subscription state inside the network.

When a user chooses to publish a piece of data, e.g., a photo, npChat encrypts the data, segments it, signs the segments using the user's private key, and loads the signed segments into the application's content cache to satisfy incoming interests. npChat then uses the PSync library to publish the data name to all the subscribers of this particular feed. More specifically, suppose `<UserPrefix>` is a publisher's name prefix, then each subscriber sends a Sync Interest `/<UserPrefix>/sync/<SubscriptionList>/<IBF>` periodically to the publisher which stays pending at the publisher. Whenever the publisher produces new data for a particular feed, it will send a Sync Data packet in response to each pending Sync Interest, containing the name `/<UserPrefix>/<feed>/<seqno>`. Each subscriber then sends an Interest to fetch the data for `/<UserPrefix>/<feed>/<seqno>` which contains the actual data name, e.g., the photo's name, and any related access control information. The subscribers then fetch the data and decrypt it using the corresponding key (see Section 2.8 about access control).

2.8 Access Control

A user's feed can be accessed only by his/her friends. This control is enforced by encrypting all the feed data using a symmetric key created by this user and shared with all the friends. Whenever a friend is removed, npChat creates a new key and distributes the key to each friend by encrypting it with the friend's public key. In addition, the user may want to choose specific friends with whom to share a particular piece of content. As the set of friends who are granted access may differ for each data object in this case, we use a different symmetric key to encrypt each data object for such content, which also ensures forward secrecy.

The process for sharing data with selected friends is as follows. To prevent unauthorized friends from getting the data encryption key (let's call it "content key"), it is encrypted using each authorized recipient's public key. The set of authored friends is then distributed through PSync along with the name of the shared content (see Section 2.7). Upon receiving this information, each recipient in the set of authorized friends fetches both the filename and the encrypted content key, decrypts the content key and then decrypts the data using the content key. Even if an unauthorized friend retrieves the encrypted data, he/she will not be able to decrypt the content key and thus cannot decrypt the data.

We anticipate that this functionality is typically used to share content with a small set of selected friends. When the set of authorized friends is large, the overhead of encrypting and distributing the content key may be high if it changes for every shared data object. In this case, if the set of authorized friends is stable, we can use the same content key for all the data objects shared with them, so this key needs to be distributed only once - similar to the feed encryption. However, whenever the set membership changes, we need to change this key and redistribute it. This is one of our future research issues along with more sophisticated access control.

2.9 Privacy

One privacy concern related to NDN applications is the use of application names in routing, in-network caching, and security. Note that many incorrectly assume all NDN names are clear human-readable text, but this does not have to be the case. If names are sensitive, the producer can obscure them by encrypting or hashing them, as long as only the authorized consumers know how to apply the same operations to derive these encrypted/hashed names. There are more sophisticated approaches. For example, ANDANA is a Tor-like framework for NDN proposed by DiBenedetto et. al. to provide anonymity through onion routing [7]. Tourani et. al. also proposed an NDN name anonymization scheme to not only stop leakage of identity but also stop censorship by ISPs [28]. We would like to point out that npChat can take advantage of any name anonymity techniques developed for NDN.

A specific privacy concern for npChat is the sharing of Friend List, which helps discovering potential friends and growing the network. This is an optional feature in the application, so a user can turn it off to stop sharing the list with his/her friends. To encourage friends to share this information, the app may introduce some incentives, e.g., a user can earn some credit when the sharing of his/her Friend List results in another user discovering a new potential friend. What specific incentives to introduce is an open issue. Regardless of the specific incentive mechanims, our evaluation shows that even if only 20% of the users share their Friend List, the social network can still grow at a good rate (see Section 4.1).

3 Implementation

We have implemented the npChat application based on the design described in Section 2. It currently supports discovering local users, making friends using different trust models, and sharing of photos/files with all or selected friends.

3.1 GUI and Functions

npChat's main screen provides access to four primary activities available to users: Camera, Files, Friends, and See Photos. The Camera activity opens the Android device's camera and, upon taking a photo, allows the user to choose whether to save and publish or just save the photo. The Files activity lets the user share any files on the device and browse received non-photo files. The Friends activity is where the user can scan other user's QR codes, display their own code, send remote friend requests, and view a list of of their current friends. Received photos can be viewed in the See Photos activity.

3.2 npChat and NFD Android

npChat is supported by NFD Android which runs the C++ NFD implementation via Java Native Interface (JNI). The app uses the jNDN [20] and jNDN-management [22] libraries to connect to NFD Android via a TCP face and send management commands to NFD. In addition, NFD Android's GUI can be used to create faces, register routes, and check NFD's status.

NFD Android supports WiFi Direct and creates a TCP or UDP face towards the NFD running on every other node connected to the same WiFi Direct network. Moreover, it communicates with each peer NFD to obtain the set of prefixes registered in that NFD and automatically creates routes towards those prefixes using the

peer as the next hop. This eliminates the need for users to manually create these routes.

3.3 Local User Discovery

We use the UDP multicast face created by NFD Android to send periodic NDN Interests for local user discovery (see Section 2.6). Other users on the same network can respond to such Interests with their user prefix information. User prefixes discovered in this manner are saved in persistent storage and can be viewed by the user as potential friends. If the network does not support multicast to NFD's port, we use Network Service Discovery (NSD) [3], which is Android's implementation of DNS-SD, to discover local users. We encapsulated NSD functionality in the NSDHelper class, which handles creating faces and routes upon discovery of other npChat instances. It also handles destroying a face upon notification that the corresponding service no longer exists.

3.4 Prefix Registration and Routes

npChat registers the prefix `/<UserPrefix>/npChat/<username>` and filters Interests under several sub prefixes. The sub-prefix `/metadata` is used to publish meta information for fetching files, e.g., file name and list of authorized users. The `/file` sub-prefix is for publishing data from files (note that photos are served as files too). In addition, the app serves certificates issued to friends under `/certs`, encryption keys under `/keys`, and user's friend list under `/friends`. Finally, it uses `/friend-request` to receive remote friend requests.

For outgoing Interests to another user, the app installs routes based on how the user is discovered. If the user is found by our NDN-based discovery mechanism, npChat registers that user's routes on the multicast face created by NFD. If the user is discovered by the DNS-SD discovery mechanism, npChat creates a unicast face to the user's IP address, and registers the corresponding routes on that face. Otherwise, i.e., the other user is not on the local network, we still need to implement an Identity Manager for NFD Android in order to connect to a wide area network (e.g., the NDN testbed).

3.5 Making Friends

As mentioned in Section 2.6, friends can be made in person by using a QR code. A QR code generator and a QR scanner, via the ZXing library [2], have been implemented for the purposes of facilitating in person friend registration. Upon logging in as a new user, the app generates an RSA public/private key pair, and creates a PIB (Public Information Base) and a TPM (Trusted Platform Module) to store the keys in persistent storage. It then encodes the user's username and self-signed certificate in the form of a QR code.

If the two users cannot meet in person, our app generates a signed Friend Request and Response, and uses the ValidatorConfig from jNDN to load the trust anchor and trust schema for verifying received messages.

3.6 Pub-Sub, Data Sharing, and Access Control

A file selection activity was created to allow users to publish files stored on their device. Publishing a file also creates a QR code which encodes the data name. Other phones are able to fetch files by its name or by scanning the corresponding QR code. Users are also able to use the camera to take pictures, save them, and use a similar file selection activity to publish photos. For access control, we use Java's cipher library for encrypting the file and the jNDN library for encrypting the symmetric key. We then use the jNDN library to segment the file into 8,000-byte packets and sign those packets.

The PSync library provides the functions to create full sync producer, partial sync producer, and consumer objects. Similar to NFD Android's approach, we cross-compiled PSync for Android and wrote a JNI interface to make PSync available on Android. npChat creates a partial sync producer to publish the user's new data information and uses multiple consumer objects to subscribe to the data feeds of the user's friends.

Whenever npChat receives a notification of new data published by a friend, it uses jNDN's SegmentFetcher to fetch all the data segments. By using the public key that was exchanged upon friendship acquisition, npChat can verify that the data segments are actually from the friend. If the stored friend's public key does not match the signature, the data packet is dropped and the user is notified within the app. The current publicly available SegmentFetcher implementation waits to retrieve each segment until it has received and verified the previous one, which is quite inefficient. As such, it is being updated to use a dynamic window based on the additive-increase/multiplicative-decrease (AIMD) algorithm so that the consumer can pipeline its Interest packets. As we will show in the next Section, this pipelining implementation significantly improves the data transfer performance.

The content key for encrypting a user's published file is a 256-bit Advanced Encryption Standard (AES) symmetric key generated with the Java Cipher API. Encryption is performed using the AES algorithm in Cipher Block Chaining (CBC) with PKCS5 Padding. CBC requires a random initialization vector which is generated and prepended to the encrypted file before it is segmented into packets.

4 Evaluation

In this section, we evaluate important features in npChat including its ability to communicate under multiple network conditions, the trust model, access control, and use of NFD's Content Store. We simulated the growth of friends in the npChat network, compared file transfer speeds under all three supported network conditions, measured the time it takes users to become friends, demonstrated the retrieval of data from the content store of another user after the producer is disconnected from the network.

In the real experiments, we used three Android phones: two first-generation MotoX with a Dual-core 1.7 GHz CPU running Android 6.0.1 and one Nexus 5X with a Hexa-core 1.4 GHz CPU running Android 8.0. For the access point, we used a Tenda AC1900 wireless router to connect all devices using 802.11n. We ran each experiment 5 times and present the average results.

4.1 Simulating npChat network growth

We wrote a simple network simulation where each node is a potential npChat user and each bidirectional link represents friendship. In [19], authors studied the growth of the Flickr social network and observed that a user creates one out-degree link per day for every 227 out-degree links it already has. As mentioned in Section 2.6, they also found that more than 80 percent of the new links are within 2 hops of the user (friends of friends). In our simulation, we randomly assign 200 users to 10 organizations and bootstrap

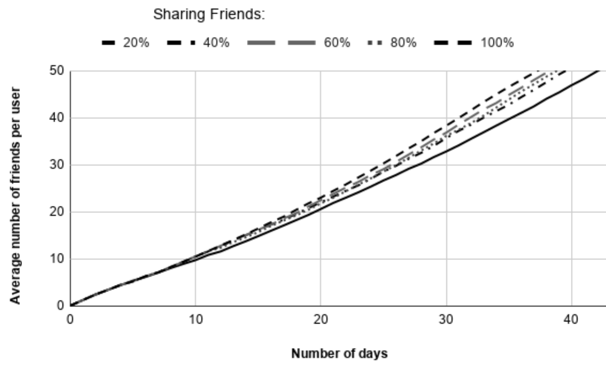

Figure 5: Average number of friends per user for 200 nodes under various levels of Friend List sharing

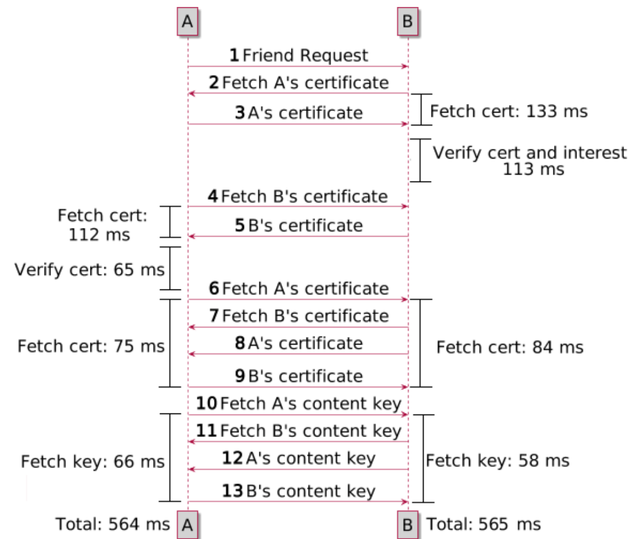

Figure 6: Remote Friendship Request Time between Devices on Two Different Networks

the social network by letting each user send a Friend Request (out-degree link) to another randomly chosen user (in-degree link) in the same organization. Users who sent requests to each other became friends. Then for each day, each user decides if it will establish a new out-degree link with a probability of 1/227 multiplied by the number of out-degree links that user has. If a user decides to establish a new out-degree link, with 80% probability, the link will connect to a user who is in the same organization (local discovery) or a friend of friend and, with 20% probability, the link will connect to a user who is not in the potential friend list, i.e., not in the same organization and not a friend of friend, in order to simulate making friends with such a user in person.

We varied the percentage of users sharing their Friend Lists from 20% to 100% and let the simulation run until the average number of friends per user reaches 50 (1/4 of the total user population). Figure 5 shows that, the average of 50 friends per user was achieved in 38 days with everyone sharing their Friend Lists, and it took only 5 more days to reach the same number with only 20 percent users sharing. This is because there is sufficient overlap between the Friend Lists so that the union of those shared Friend List will cover a sufficient percentage of the union of the private Friend Lists. In other words, the social network will grow well even if only a small percentage of users are willing to share their Friend Lists.

4.2 User Discovery

We evaluated our NDN-based local user discovery scheme using the two MotoX's and the Nexus 5X connected to a single access point. After one user sent its Discovery Interest, the first response was received in an average of 270ms. This means the existing users discovered this new user in half of that time. This user continues to discover the other users (besides the one that sent the first response) through their periodic Discovery Interests sent every 3 seconds, so it takes up to 4 seconds to discover all local users.

4.3 Making Friends

In order for users to become friends in person, they must establish trust by exchanging and signing each other's certificates. Our experiment demonstrates how long it takes this exchange to occur by recording the amount of time that passes from the users scanning each other's QR codes until both users have received their certificates back via each NDN face type. Table 1 shows our experiment results. As MotoX has a much slower processor than Nexus 5X, it takes more than twice the time to issue a certificate compared to the Nexus 5X. The transfer time for the certificate is similar for all the three network connections (from 223ms to 228ms).

If two users cannot meet in person, trust and friendship must be established remotely. For our evaluation, we connected each MotoX to a Raspberry Pi running as an access point with NFD and connected the Pi's over Ethernet to simulate the activity for when two users are on separate networks. Figure 6 shows that network operation of fetching certificates takes the most time. We can eliminate this time by putting the certificate in the Friend Request as mentioned in Section 2.6. Nevertheless, the overall process is quite fast (much less than 1s).

4.4 Data Transfer in Different Network Environments

To test file transfer times in different network environments, we replicated each environment with two MotoX phones and a Wifi router that supports DNS-SD and UDP multicast traffic. For the TCP unicast connection, we disabled the multicast face in our app. For the UDP multicast experiment, we disabled DNS-SD within our app. Finally, to test Wifi Direct, we connected the two phones directly using NFD's Wifi Direct before launching our app.

Each test consisted of transferring a 1.1MB, 2.1MB, and 5.2MB photo from one device to the other and recording the time from which the producer updated the sync data until the consumer had fetched the last segment. All tests were run first with the non-pipelined version of the jNDN Segment Fetcher and then with the AIMD pipelining version. As shown in Table 2, the transfer speeds over an access point are similar for both unicast and multicast faces through the AP, but WiFi Direct is noticeably slower, especially

Table 1: Making Friends in Person (scan each other's QR code and fetch new certificate from each other over a local network).

Nexus 5X Certificate Extraction	Nexus 5X Certificate Signing	MotoX Certificate Extraction	MotoX Certificate Signing	TCP Unicast Transfer	UDP Multicast Transfer	Wifi Direct Transfer
2ms	30ms	3ms	76ms	223ms	226ms	228ms

Table 2: Data Transfer Time (seconds) in Different Network Environments

Data Transfer	Transfer Mode	No Pipelining			Pipelining		
	File Size	1.1MB	2.1MB	5.2MB	1.1MB	2.1MB	5.2MB
NDN Face Type	TCP Link via AP	10.4	23.3	67.8	2.3	4.9	10.8
	UDP Multicast via AP	10.2	22.6	70.0	2.9	4.7	10.2
	TCP Link via WiFi Direct	12.3	115.8	203.4	3.3	5.2	14.8

Table 3: Access Control Cost: Data Encryption and Decryption Time (milliseconds) with Different Devices

	Encryption		Decryption	
	MotoX	Nexus 5X	MotoX	Nexus 5X
1.1MB	53.9	10.1	10.2	4.2
2.1MB	80.1	10.4	11.7	4.6
5.2MB	144.3	10.9	19.2	5.7

with the non-pipelined Segment Fetcher which suffered from many timeouts during the fetching process. Pipelining the Interests reduced the data transfer time significantly (by a factor of 21 in the case of WiFi Direct and 2.1MB file size).

4.5 Access Control

For access control, the main overheard comes from encrypting and decrypting the file. We recorded the time it took a MotoX and a Nexus 5X to encrypt and decrypt 1.1MB, 2.1MB, and 5.2MB photos. As can be seen in the results shown in Table 3, this adds an insignificant amount of time to the process. If we assume the MotoX is the producer and Nexus is the consumer, then the total encryption and decryption time for a 5MB photo is 150ms, while the data transfer time is between 10.2 and 14.8 seconds depending on the network type (see Table 2). This means the access control overhead in this case is no more than 1.5%.

4.6 Disconnections

The ability to retrieve data from other NDN nodes' content stores is an important part of the NDN architecture. To demonstrate how our app takes advantage of this, we connected three phones to a single access point and transferred a 509KB photo. We will refer to the device (MotoX) that publishes the file as the Producer, the device (Nexus 5X) that fetches the file from the Producer as Consumer 1, and the device (MotoX) that fetches the file from Consumer 1's content store as Consumer 2. As illustrated in Figure 7, in the first test, the Producer publishes the file and both consumers receive the updated sync data. Consumer 1 immediately fetches the file and the Producer goes offline. Consumer 2 then begins to fetch the file. Figure 8 illustrates the second test, where both consumers immediately begin to fetch the file, but Consumer 2 disconnects after fetching 5 packets and reconnects 4 seconds from initially receiving the sync update, by which time Consumer 1 has finished fetching the file and the Producer has disconnected.

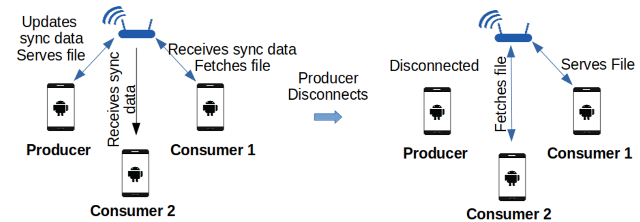

Figure 7: Producer disconnects before Consumer 2 fetches file.

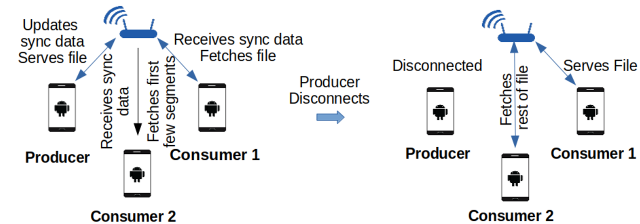

Figure 8: Consumer 2 disconnects before finishing file transfer.

Figure 9 shows that Consumer 2 was able to successfully fetch the files published by the Producer even after the Producer had gone offline. In the first graph, Consumer 1 fetches the last segment at around 1.5 seconds, the Producer disconnects at the 3-second mark, and at 4 seconds, Consumer 2 comes online and is able to fetch the file. In the second graph, Consumer 2 initially fetches some segments and the disconnects at .2 seconds, Consumer 1 finishes fetching at about .8 seconds, the Producer goes offline at 3 seconds, and Consumer 2 comes back online at 4 seconds and is able to fetch the remaining segments in the file.

5 Related Work

5.1 NDN Applications

A number of applications have been developed over NFD Android, including NDNFit [33], ChronoChat [26], and NDN-Whiteboard [12]. NDNFit collects health data on an Android device and publishes this data. ChronoChat [26] allows users to form chatrooms to enable group messaging. NDN-Whiteboard enables users to share a "whiteboard" on which they can draw together in real time. Another application Now@ [4] implements a similar design to Twitter by allowing users to subscribe to multiple namespaces under which users can write posts. Compared to these previous efforts, our work

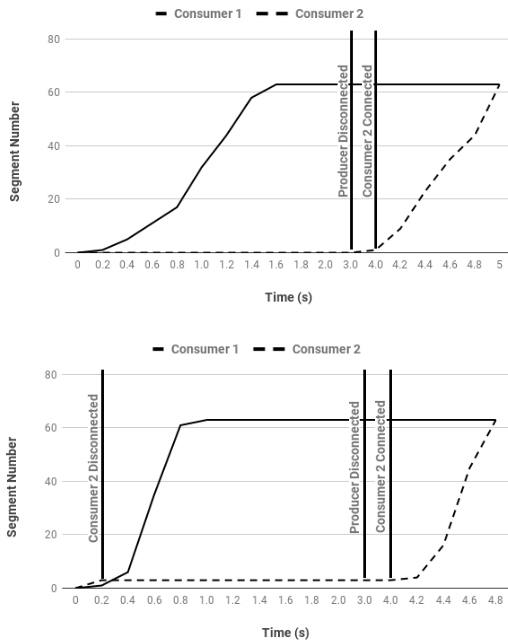

Figure 9: Consumer Retrieving Files after Producer Disconnects

puts more emphasis on decentralization in all aspects of the application design.

5.2 Federated Social Media Applications

Federated social media applications such as Mastodon [16], GNU Social [25], and Diaspora [8] have been gaining popularity. These applications have the model of federated servers communicating with each other to form a decentralized network. A user can create an account on any of the existing servers or on a self hosted server. Mastodon uses ActivityPub [1] for federation, so it can interact with other applications that also use ActivityPub. It also supports OStatus for compatibility with GNU Social. Diaspora uses its own protocol for federation but allows feeds from Twitter, Facebook, etc. to be integrated. Below are the major differences between npChat and federated applications represented by Mastodon and Diaspora.

First, the administrator of a server may shut down the server at any time, and none of the users on that server will be able to use their application. This is a general problem of any federated systems. npChat does not have this problem, as it is a fully decentralized.

Second, similar to Email, the ActivityPub protocol is based on a push model where one user's server (source server) posts media content to another user's server (destination server). This model requires servers to be stable (almost always on), otherwise they will not receive the content. Even if a user self hosts a server, this server still needs to be on all the time to receive potential content. In contrast, npChat users do not depend on stable servers. Whenever a user becomes active, that user's npChat will check whether there is any new content published through PSync and pulls the content. Even if the producer of the content is offline at that moment, npChat can get the content as long as the content has been received by another user or cached in the network.

Third, the administrator of a federated server in Mastodon has access to all its users' data (data is not encrypted on the server) and keys [17]. In Diaspora, private messages are encrypted and the symmetric key is encrypted using recipient's public key [9]. However, the keys for a user are still stored on the server the user chooses to use. In npChat, only authorized users can decrypt the data shared with them since no other users have the symmetric key used to encrypt the data.

Finally, in Mastodon, the trust on a user's data is implicitly derived from the trust of the server that hosts the user, as the user's data is signed with the user's public key (by the server) and the key is retrieved over SSL from the server. This trust model depends highly on having these stable servers as trust sources. In contrast, npChat allows the users to establish trust with each other directly and derive trust from multiple sources, including stable trust anchors and mutual friends. Diaspora allows multiple signers for each user's public key and defines a mechanism for discovering the signing keys. However, the specific authentication procedure is outside of Diaspora's scope. In npChat, the key authentication (and trust establishment) is done automatically by the application using the information provided in the Friend Request message, without relying on a separate authentication protocol or process.

5.3 Solid Platform

Solid [24] aims to decouple data storage from social media applications. Each user's data is stored in one or more PODs that can be self hosted or provided by a third party. Unlike Mastodon or Diaspora, a Solid POD contains only users' data, not application logic. Applications can get authorization from a user to access the user's data. Some applications have been developed on the Solid platform, such as Contacts [15] which allows users to manage contacts stored on their PODs.

The objective of npChat is to develop a fully decentralized application, so we focus more on application logic, not on data storage. However, similar to Solid and unlike the existing federated applications, npChat does not couple storage with the application. In fact, an npChat user can utilize the data storage hosted by any provider – npChat can use the NDN Repo protocol to store the data on a repo hosted by the provider. Whether a user in another application can access the data depends on whether the npChat user shares the data decryption key with that application/user. We will address this issue in future application design.

6 Conclusion and Future Work

In this work, we explored the design and implementation of a fully decentralized application over NDN. Our experience demonstrates that it is feasible to develop such an application, but it requires new approaches to designing the application namespace, establishing trust, discovering potential friends, and performing pub-sub. In the next step, we will investigate access control methods to scale to a large and dynamic set of selected users to share content. We will also explore more complex trust models, such as partial trust based on the number of mutual friends. To increase usage, we will add the functionality to connect users to the NDN testbed by finishing the Identity Manager. Finally, we will improve the user interface and publish our application on the Android Play store.

References

[1] ActivityPub. 2018. ActivityPub W3C Recommendation. https://www.w3.org/TR/activitypub/

[2] Agustin Delgado, et al. 2011. ZXing (Zebra Crossing) barcode scanning library for Java, Android. https://github.com/zxing/zxing

[3] Android Developers. [n. d.]. Use network service discovery. https://developer.android.com/training/connect-devices-wirelessly/nsd

[4] Omar Aponte and Paulo Mendes. 2017. Now@ - Content Sharing Application over NDN. In *Proceedings of ACM ICN 2017*.

[5] Jiachen Chen, Mayutan Arumaithurai, Lei Jiao, Xiaoming Fu, and KK Ramakrishnan. 2011. COPSS: An efficient content oriented publish/subscribe system. In *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*. IEEE Computer Society, 99–110.

[6] Stuart Cheshire and Marc Krochmal. 2013. *DNS-based service discovery*. Technical Report.

[7] Steven DiBenedetto, Paolo Gasti, Gene Tsudik, and Ersin Uzun. 2011. ANDaNA: Anonymous named data networking application. *arXiv preprint arXiv:1112.2205* (2011).

[8] Diaspora Foundation. 2019. diaspora. https://diasporafoundation.org/

[9] Diaspora Foundation. 2019. diaspora federation protocol: Encryption. https://diaspora.github.io/diaspora_federation/federation/encryption.html

[10] Allen Gongm. [n. d.]. NDN Over WiFi Direct Protocol Specification.

[11] Michael T Goodrich and Michael Mitzenmacher. 2011. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 792–799.

[12] Sumit Gouthaman, Peter Huang, and Peter Bankole. 2015. NDN-Whiteboard. https://github.com/named-data-mobile/apps-NDN-Whiteboard

[13] Cenk Gündoğan, Peter Kietzmann, Thomas C Schmidt, and Matthias Wählisch. 2018. HoPP: Robust and Resilient Publish-Subscribe for an Information-Centric Internet of Things. *arXiv preprint arXiv:1801.03890* (2018).

[14] P. Jones, G. Salgueiro, M. Jones, and J. Smarr. 2013. *WebFinger*. RFC 7033. RFC Editor.

[15] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. 2016. A demonstration of the solid platform for social web applications. In *Proceedings of the 25th International Conference Companion on World Wide Web*. International World Wide Web Conferences Steering Committee, 223–226.

[16] Mastodon. 2019. Mastodon documentation. https://docs.joinmastodon.org/

[17] Mastodon. 2019. Mastodon Privacy. https://docs.joinmastodon.org/usage/privacy/

[18] Spyridon Mastorakis, Alexander Afanasyev, Yingdi Yu, and Lixia Zhang. 2018. nTorrent: Peer-to-Peer File Sharing in Named Data Networking. In *ICCCN*.

[19] Alan Mislove, Hema Swetha Koppula, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2008. Growth of the flickr social network. In *Proceedings of the first workshop on Online social networks*. ACM, 25–30.

[20] NDN Project Team. 2013. jNDN: A Named Data Networking client library for Java. https://github.com/named-data/jndn

[21] NDN Project Team. 2015. Android Implementation of NFD. https://github.com/named-data-mobile/NFD-android

[22] NDN Project Team. 2015. jndn-management: Tools for managing an NDN forwarding daemon. https://github.com/named-data/jndn-management

[23] Arthi Padmanabhan, Lan Wang, and Lixia Zhang. 2018. Automated Tunneling Over IP Land: Run NDN Anywhere. (2018).

[24] Andrei Vlad Sambra, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, Dmitri Zagidulin, Ashraf Aboulnaga, and Tim Berners-Lee. 2016. Solid: A Platform for Decentralized Social Applications Based on Linked Data.

[25] GNU Network Services. 2019. GNU Social. https://gnu.io/social/

[26] Tyler Vernon Smith, Alexander Afanasyev, and Lixia Zhang. 2017. *ChronoChat on Android*. Technical Report. NDN.

[27] NDN Project Team. 2015. Certificate Bundle Design. https://redmine.named-data.net/issues/2766

[28] Reza Tourani, Satyajayant Misra, Joerg Kliewer, Scott Ortegel, and Travis Mick. 2015. Catch me if you can: A practical framework to evade censorship in information-centric networks. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking*. ACM, 167–176.

[29] Lan Wang, Vince Lehman, AKM Mahmudul Hoque, Beichuan Zhang, Yingdi Yu, and Lixia Zhang. 2018. A secure link state routing protocol for NDN. *IEEE Access* 6 (2018), 10470–10482.

[30] 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 International Conference on Information-Centric Networking*.

[31] Yingdi Yu, Alexander Afanasyev, Zhenkai Zhu, and Lixia Zhang. 2014. *An Endorsement-base Key Management System for Decentralized NDN Chat Application*. Technical Report. UCLA.

[32] Haowei Yuan and Patrick Crowley. 2013. Experimental Evaluation of Content Distribution with NDN and HTTP. In *IEEE INFOCOM 2013 Mini-Conference*.

[33] Haitao Zhang, Zhehao Wang, Christopher Scherb, Claudio Marxer, Jeff Burke, and Lixia Zhang. 2016. Sharing mHealth Data via Named Data Networking. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*. 142–147.

[34] M. Zhang, V. Lehman, and L. Wang. 2017. Scalable name-based data synchronization for Named Data Networking. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057193