Implementing Instant Messaging Using Named Data

Jiangzhe Wang[*]
lucas@cs.ucla.edu

Chunyi Peng[*]
chunyip@cs.ucla.edu

Chiyu Li[*]
lichiyu@cs.ucla.edu

Eric Osterweil[†]
eosterweil@verisign.com

Ryuji Wakikawa[‡]
ryuji@us.toyota-itc.com

Pei-chun Cheng[*]
pccheng@cs.ucla.edu

Lixia Zhang[*]
lixia@cs.ucla.edu

ABSTRACT

The Internet has been a huge success, but it is showing signs of age. Among multiple proposed directions for the Internet's future design is a promising architecture called Named Data Networking (NDN). NDN casts data as a first class element of the network's architecture in an effort to greatly facilitate new application development. However, as with any new architecture, one important deployment issues is being able to evolve existing applications. In this paper, we use a library for Instant Messaging (IM) applications called *libpurple* as a case study to demonstrate both the advantages of implementing IM as a serverless application in NDN and to explore promising approaches to porting applications to NDN. Our new serverless design enables IM clients to chat with each other without infrastructure support. Since libpurple is widely used as the transport layer of several IM applications (including Pidgin, Adium, and Apolio IM) our new library *NDNPurple* is able to seamlessly support these applications without modification to higher-layer code. In this work, we propose that our serverless design serves as a template for porting applications, and using it requires only trivial changes existing applications' state machines in order to facilitate interactions with NDN through. We do this by embedding a local pseudo-proxy in the application itself, and we are therefore able to leave the legacy code's state machine alone.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications

[*]Computer Science Department, UCLA
[†]VeriSign Labs
[‡]Toyota Infotechnology Center, Mountain View

AINTEC'10, November 15–17, 2010, Bangkok, Thailand.

General Terms

Design, Experimentation

Keywords

Instant Messaging, Named Data Network

1. INTRODUCTION

The original design goal of today's TCP/IP Internet architecture is an effective interconnection of all existing networks and hosts [11]. Since interconnectivity is one of the Internet's most fundamental goals, the Internet's hourglass architecture is centered around endpoints, or IP addresses. Through much of the Internet's history (especially the in the earliest years) the majority of hosts were stationary and they had little online content, and there was a default notion that the *address* of content needed to be ascertained in order to access it. In this setting, the host-to-host communication model worked out quite well. However, as the Internet grows, its usage model and requirements are evolving. The number of network entities (hosts) is exploding, many are becoming increasingly mobile, and the way they access data is evolving too. In this changing environment, new applications face three basic challenges: server scalability, end-host mobility, and data security.

As one of the most popular IM and VoIP application, Skype acts as a prime example of one way that applications can enhance their server scalability [7]. By using a Peer-to-Peer (P2P) architecture, Skype reduces the resource requirements that a centralized architecture would have to pay. Rather, Skype relays data traffic through online peers (users' hosts). This approach highlights that some users and applications, such as IM and VoIP, may care more about the *data* that they can get from the network than they do about *how* it was delivered. Many users and applications may not care whether their text messages or voice packets are forwarded from a central server or another nearby Skype peer, as long as the data source can be authenticated and the data secured. If we look at network communications from a data-oriented perspective, it becomes straightforward to solve the aforementioned three challenges: i) one can directly name the

data instead of naming its container, ii) secure the data itself instead of the communication channels, and iii) deliver data to interested users rather than specific locations, removing the need for centralized servers.

In this paper, we use IM as an example to demonstrate the implementation of serverless applications using the Named Data Networking approach (aka CCN) [13]. We chose to implement an NDN-based IM middleware by adapting a well-known IM library called *libpurple* [4]. We called this port *NDNpurple*. By porting a commonly used library, rather than building a brand new one from scratch, we were able to implicitly port all of the IM clients that already use libpurple for their substrate. Our contributions in this paper can be summarized as follows.

1. We describe the implementation of a library *NDNPurple* that supports existing IM applications, and a demo application Pidgin built with *NDNPurple*.
2. We show a working example of how to construct name space for IM applications, as well as mechanisms of name discovery, membership management and multi user access control.
3. Our experience suggests that porting existing applications into NDN should favor the "keep-and-add" approach over "erase-and-rewrite" when software code base is large.

The rest of this paper is organized as follows: Section 2 describes the basic NDN operations and architecture of libpurple based IM applications. Section 3 discusses the design of NDNPurple for name convention and discovery, and multi-user access control. Section 4 includes our implementations details. We discuss related work in Section 5 and conclude in Section 6.

2. BACKGROUND

In this section, we briefly introduce how NDN works and the architecture of *libpurple*-based IM application.

2.1 NDN Overview

Here, we briefly illustrate NDN's major procedures using a sample NDN topology shown in figure 1. H1-H3 are three end user hosts and R1-R4 are NDN routers. On top of the tree topology is a Youtube server Y that's connected with R1 and R2. R1 has three interfaces labeled as f0-f2. We show the advantages of NDN when three end hosts H1-H3 are interested in the same video clip stored on the Youtube server, say foo.mpg. NDN requires that every piece of content is named. For simplicity, we assume the video is named as *ccnx://youtube/foo.mpg*.

Every NDN router has three major components: Content Store, Pending Interest Table(PIT) and Forwarding Information Base(FIB) as shown in Figure 2. Similar with BGP[1], the whole system starts by Youtube server announcing name prefix *ccnx://youtube* to the Internet. Therefore in figure 2,

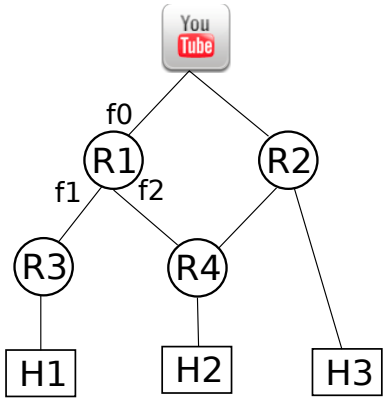

Figure 1: A sample topology of NDN

R1 configures its FIB so that the name prefix is associated with interface f0, which the prefix is announced from. In NDN's subscribe/publish model, end users express interest to pull data back from the network. For instance, H1 and H2 want to receive the same video almost simultaneously. Each one of the two hosts sends an interest message tagged with human-readable name *ccnx://youtube/foo.mpg*. After a NDN router receives an interest, say R3, it would first lookup its Content Store to see if there is already cached content that can satisfy the interest. If the router finds one, an response will be issued immediately with cached content. Otherwise, the router checks its PIT to see if it has already forwarded out an interest with the same name. In our example, when all Content Store and PIT tables are empty, the interest message I1 generated from H1 is forwarded along the path R3-R1, and then to the Youtube server Y. While R3 and R1 forward I1 toward Y, each of them puts the interest in its own PIT, as well as the corresponding incoming interface, i.e. f1 for R1. Suppose H2's interest I2 hits R1 after R4 forwards I1, R1 will not forward I2 to Y because I1, the name of which is the same as I2, is pending. R1 only needs to add f2 to the PIT entry as shown in figure 2.

NDN routes *interest* messages, but not *data*. *Data* packets are delivered back to their original requesters by going through the paths that their corresponding interest messages have traversed. When data is replied from Y, and arrives at R1, two separate pieces are sent out: one through f1 for H1, and the other through f2 for H2, thanks to the PIT entry's memory of f1 and f2. After data is sent out from R1, the PIT entry will be removed.

In a word, NDN proposes a content-centric paradigm which cares about *what* that users want rather than the network's *where*. NDN uses human-readable name as the primitive for end users and decouples data from its location. Its major differences from the conventional IP network can be summarized on three aspects.

- First, NDN applies a subscribe and publish model where receiver-side sends out interests to initiate communi-

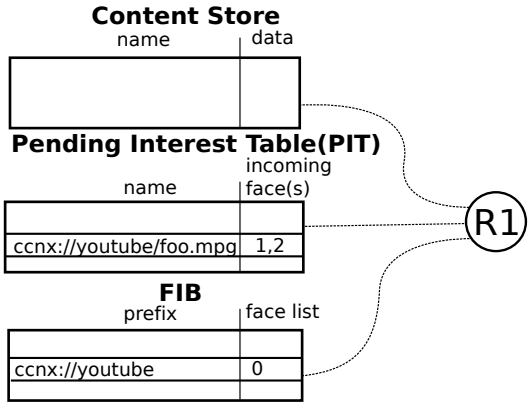

Figure 2: Components of NDN router

cation. Each interest message is tagged with a name representing what type of data the user would like to receive. Network routing is done by each NDN router forwarding interest messages toward publishers that can possibly satisfy the interest with data. A data packet satisfies an interest by matching the interest name as a prefix of its own name. Interest messages are stored in PIT on every router it traverses and used for data packets to flow back to the original requesters.

- Second, each piece of content is named and cached on intermediate routers in NDN. When data packets are delivered back to the requesters, they can be cached on every intermediate router, with an explicit and unique name, e.g. *ccnx://youtube/foo.mpg*, independent of an end-to-end session. When a second interest message arrives at the NDN router whose cached content can match the interest's name, a response would be issued immediately with the cached data. Therefore, caching forms an automatic tree rooted at a single member and branched over other participants. It not only reduces duplicate data transmission, but also improves receiver side performance.

- Third, NDN enforces every packet to be tagged with its publisher's signature and secures data itself rather than the communication channel as in TCP/IP. Since security is not the major concern of our work in current phase, we will not go to in-depth discussion in this paper.

2.2 Libpurple based IM applications

Libpurple is used by several IM applications including Pidgin for Windows&Linux and Adium for Mac OS. It is notably known for its support of multiple chat protocols. Figure 3 illustrates the architecture of *libpurple* based applications. We put the aforementioned applications on the top layer and *libpurple* in the second layer as a common library. *Libpurple* package implements a flora of chat protocols such

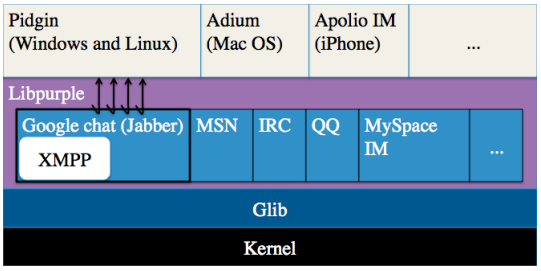

Figure 3: Architecture of libpurple based projects

as Jabber, MSN, Internet Relay Chat(IRC) etc. Among these protocols, Jabber encodes all of its messages in XML format [14], and supports more features than others, including asynchronous message relaying, transport layer security and audio/VoIP etc.[2] It serves as the underlying chat protocol of Google Talk [3]. Therefore we start with Jabber as a sample protocol and generalize the principles of porting existing protocols.

Depending on the protocols specified, *libpurple* runs the corresponding state machine by monitoring events injected from upper layer applications or triggered by lower layer network events. In order to send a message out, application layer uses the appropriate chat protocol library to construct a message and pass it to *libpurple*. And to handle received message, application layer registers callback functions with *libpurple*. When *libpurple* receives message from the network, and decides that application layer should be notified, the corresponding callback function will be triggered. For instance, before Jabber state machine runs, application layer would register a `buddy_status_change` function into *libpurple*, when *libpurple* receives a status change message from Jabber server, the `buddy_status_change` function would be triggered. What application layer does in the callback function is redraw the corresponding buddy's status icon.

3. NDNPURPLE DESIGN

This section describes our NDNPurple design that enables Instant Messaging over Named Data Network.

3.1 Design Issues

In chatting scenarios, each user may join and leave a chat room at any time, and each of them can independently send his/her messages. In this case, messages are dynamically generated and should be delivered in real time. The traditional way to implement a chatting or instant message application is to apply client-server model, where a central chatting server takes charge of all the signaling, membership management, communication coordination and finally message delivery among multiple parties. Our goal is to build a server-less chatting application over NDN, which can func-

tion when no central server is available to handle the above tasks. To develop NDN-based Instant Messaging applications, we must solve the following design issues:

- ***Name convention***. NDN requires that each content (here, text message and voice call) is identified by a unique name. How to assign a unique name for each piece of data? Unfortunately, the underlying NDN layer has no clue how to automatically generate content names. The fact that unique names are missing in application layer requires the middleware (i.e. NDNPurple) to provide a naming mechanism in order to assure message delivery over NDN.
- ***Name discovery***. In order to receive data over NDN, each party needs to get a content name and express the name in an interest message. However, in a serverless scenario, how to discover these names for NDN interests in a distributed way? Definitely, the discovery mechanism is closely related to the design of name convention. Besides, we will show how to bootstrap basic parameters that constitutes data names.
- ***Membership management***. Who joins or leaves a chat room? Who are still active in a chat room? Without a centralized server, each party has to learn and manage members in a distributed way. It requires NDNPurple to provide an extra mechanism to handle membership management over NDN.

In the following sections, we elaborate our design solutions to name convention, name discovery and membership management in NDNPurple.

3.2 Name Convention

In this section, we propose a name convention solution combining sequence number and user namespace, which provides unique names for dynamically generated content.

IM application involves dynamic message generation and delivery. Suppose Alice and Bob are chatting in a chat room named *demo*. We assume the name of this chat room is different from the others, otherwise there exists another unique ID for each chat room. Obviously, NDN layer can inherit the name of a chat room to develop a unique NDN name scope, that is to say, all the messages in room *demo* can be named with a prefix *ccnx:/demo/*. However, this prefix itself is not sufficient to identify all the data generated in this chat room. Messages may dynamically grow, for example, Alice may share her trip photo, followed by her comments and trip tips, and they have to be named as different messages. To differentiate messages within the same chat room, we propose to construct a NDN name using sequence number as a suffix, e.g. *ccnx:/demo/v1*, *ccnx:/demo/v2*.

Sequencing is an effective measure to handle message dynamics in IM scenarios. With the help of sequence number information, each message can be uniquely identified. Moreover, it is convenient to derive the next sequence number (an incremental sequence number), which is desirable in name discovery. However, this simple sequence number-based solution may bring out a new design issue, i.e. name conflict, inherent in multi-user access channels. Take an example, Alice and Bob may send messages almost simultaneously and they may potentially use the same sequence number (e.g. *ccnx:/demo/v1*) for their new messages. In consequence, it may break the name uniqueness and lead to one message missing or unsynchronized message receiving at different parties. Even worse is that it is extremely hard for the application or NDN layer to detect and correct this mistake. Such conflict may become more severe under bad network connectivity. Consequently, we propose a user-based namespace to avoid name conflict among multiple parties.

The aforementioned name conflict comes from unorganized name space administration. All the active users in a chat room share the same namespace for all the messages and thus lead to name conflicts without appropriate coordination. We assign a user-specific namespace to each participant so as to avoid the above potential conflict. For example, the above messages sent by Alice are re-named as *ccnx:/demo/**alice**/v1*.

Combining sequence number and user-specific namespace, we come out a unique name convention for all the messages, i.e.,

$$[\mathbf{protocol}]\mathbf{:/}[\mathbf{roomID}]\mathbf{/}[\mathbf{UserID}]\mathbf{/}[\mathbf{SeqNum}].$$

Obviously, given user namespace and sequence number, it is easy to construct content name as a message publisher. On the other hand, it raises design challenges to name discovery in a distributed way. We develop our discovery mechanism in the following section.

3.3 Name Discovery

Based on the above name convention mechanism, the major job of name discovery is to learn who are active users and what is the latest sequence number. Given user and sequence number, it becomes simple to construct interest names and receive messages from other parties. The core technique of name discovery is to learn active users and latest sequence number. We propose the way of broadcasting each one's name through interests.

In our proposed solution, *NDNPurple* pre-defines a name space for queries, which is known to all participants, e.g. "*ccnx:/demo/**user***". Each user who wants to receive information (here, active user or latest sequence number) can announce it out to configure FIBs and attract interests. In Figure 4, Alice announces the well-known prefix in the network so that all routers and hosts know how to reach it. Alice's route propagation path is shown as solid arrows in the figure. This method does not follow the traditional interest-out-data-back model to learn active user. Instead, it takes a more aggressive way that embeds user name into an interest message and broadcasts to members that have announced the aforementioned prefix route. We think this mechanism is ok if the embedded information is reasonably short. Take

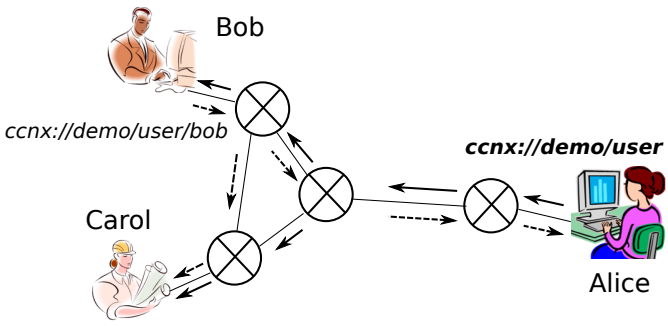

Figure 4: Prefix announcement and periodic interests

an example of an active user query. The requester does not send his/her query interest out, instead, he/she just listens to information broadcasted from others. On the other side, active users do not wait for an incoming query, instead, they periodically claim "I am here". For instance in Figure 4, Bob sends an interest named "*ccnx:/demo/**user**/bob*" out and notifies other users that he is still alive in the chat room *demo*. Since all the users are configured to announce the name "*ccnx:/demo/**user***", they are able to hear his alive claim. Upon receiving an alive message, each receiver can immediately update his/her active user list. Similarly, Bob can broadcast his latest sequence number using the interest "*ccnx:/demo/**latest**/bob/[SeqNum]*".

Note that in the above mechanism, interest messages are used to disseminate users' names or sequence numbers, and therefore would not pull data back. However, in the case of a newly joined member, he/she needs to wait until its prefix announcement reaches every other member and also everyone else broadcasts their interest, before the new member can build a complete user list. This process can possibly introduce longer delay than expected and degrade user experience compared to the traditional server-based architecture. As an optimization, after a member, say Carol, who's already in the chat room hears a new member Bob's interest, i.e. "*ccnx:/demo/**user**/bob*", she can issue a response with her current user list. Hence, Bob is able to accelerate the user discovery process by utilizing Carol's response.

3.4 Membership Management

The aforementioned design components–name convention and name discovery–yield a solution to IM message delivery on top of NDN. In this section, we propose another design component to membership management. In any conventional IM applications, it is a basic functionality to discover other users within a chat room and maintain an up-to-date active user list. The current IM solution calls libpurple API to connect with a well-known central server who plays the role of membership management. On the NDN-based IM side, the upper application should call NDNPurple APIs to manage all the users in a server-less scenario enabled by NDN technology.

The above mechanism of name discovery provides a solution to learn active users in a distributed way. In this section, we only need to consider how to maintain and manage membership. We apply a soft-state membership management, similar to neighborhood management in RIP routing protocol [5]. Specifically, each user maintains a local active user table (AUT), where each entry is associated with an expiration timer. We reset the entry timer if we receive the corresponding active user message before timeout, otherwise we remove this entry (this user is regarded to leave) after timeout. The selection of timer is a classic design tradeoff between message workload and management latency. We use a small granularity (10s) to reduce response latency in our implementation.

4. IMPLEMENTATION

To get *NDNPurple* running over NDN network for server-less IM applications, there are two major challenges we have to overcome. The first issue is how to modify libpurple, which is based on socket programming, to support NDN that is essentially an abstract layer above sockets. NDN uses a data structure as an indicator for a data retrieval process instead of a socket, so most of the functions bound with sockets cannot be used. We developed a set of new NDN functions for *libpurple* so that they fit the NDN network. An important thing we need to claim is that we didn't change the interfaces of *libpurple* exposed to application layer and therefore code of upper layers does not need to be changed.

The set of our developed NDN functions acts as a shim layer between application and transport layers, as shown in Figure 5. The other issue is how to make the IM message protocol (XMPP [14]) designed for server-based IM applications operate in a server-less scenario. To make XMPP backward compatible, we are not trying to disfigure majority of *libpurple* code, which is quite prone to bugs given its complicated state machine. Instead we propose a proxy-based solution that allows XMPP to work as usual. The proxy works as a virtual server to carefully interact with XMPP client and masks the server-less situation from it. With both new NDN APIs and the proxy, all the IM applications supported by *libpurple* can work in a server-less scenario without any modification.

4.1 NDNPurple Functions

We have developed a set of new NDN functions in order to shift *libpurple* to a new NDN-based library. They are wrapped in the APIs of *libpurple* so that its supported IM applications can run over NDN without any modification. Within each host working over NDN network, a daemon, as a software router, should be running to handle NDN events. Then, we use the APIs that NDN provides to interact with the daemon. There are three major tasks in our provided functions.

First, *libpurple* would set up a connection to the NDN

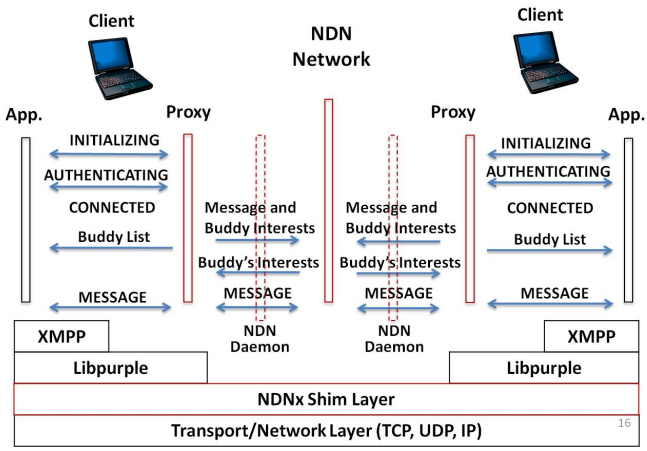

Figure 5: Implementation Architecture

daemon, express initial interests and register required interest filters while the application is initialized. Then *Purple_ccn_connect function* is called. Initialization includes announcing the user identity which informs others of its arrival and announcing prefix routes, i.e. registering interest filers. The registered interest filters are used to tell the daemon what kinds of interests the user wants to receive. Each filter is hooked with a callback function, which will be called once an interest arrives. We name the callback functiona *Purple_ccn_incoming_interest*.

Second, *libpurple* has to deal with outgoing interests and incoming data. When an interest needs to be expressed, the callback function *Purple_ccn_incoming_content* has to be registered, which will be called once the data for the interest is available. After the callback function is triggered, the *Purple_ccn_read* function will be invoked to pull available data from the daemon.

Last, *libpurple* needs to cope with incoming interests and outgoing data. The incoming interests are handled in the registered function, *Purple_ccn_incoming_interest*, where other users' alive information can be learned and some requested information would be sent out to reply to the interests through *Purple_ccn_write*. This write function is used to deliver data back to the NDN network. It is registered by the *Purple_ccn_input_add* function, and invoked when application layer has a piece of data ready. These main NDN functions we designed for *libpurple* and their functionality are listed as the following:

1. *Purple_ccn_connect*: It sets up a connection to the NDN daemon, expresses initial interests and registers required interest filters.

2. *Purple_ccn_write*: It writes data into the NDN daemon with a specified data name.

3. *Purple_ccn_read*: It reads data from the NDN daemon with a specified data name.

4. *Purple_ccn_input_add*: It is used to register a callback function for receiving data from the application.

5. *Purple_ccn_incoming_interest*: It is a callback function for receiving interests, which will be triggered once an interest arrives in the NDN daemon.

6. *Purple_ccn_incoming_content*: It is a callback function for receiving content, which will be triggered once there is available data in the NDN daemon.

4.2 The proxy-based solution

The proxy works as a virtual server for the application running over NDN network so that the client still considers that at the opposite side it's communicating with a server. In order to keep the state machine of XMPP operating as usual, the proxy should work as a server to initialize and authenticate the client to lead it to the connected state, as shown in Figure 5. After the virtual connection state establishes, the proxy learns buddies list by collecting other clients' interests and produces a XML format packet for application layer. Not only does the proxy generate correct responses to react to the XMPP protocol but also it needs to communicate with the proxies of other clients over NDN network on behalf of its served client.

4.2.1 Interaction between the proxy and XMPP

Many control messages are required to initialize the application and then keep it stay alive. So we observe what messages are used between client and server to do initialization and status maintenance. Then, we make the proxy to recognize these messages and give appropriate responses to them.

Other than control messages, the proxy currently supports two kinds of messages, buddies list and data content. To get the buddies list, the proxy collects peers' names through their announced interest messages and builds a buddies list. When it receives buddies' interests, it will add new buddies into the table or renew the timer of the existing buddies. Each buddy entry is attached with a timer so that the off-line buddy can be removed from the list when its interests have not been heard for a certain time period (10s). The buddies list would be periodically updated to the application.

In our system, there is no server and each data message is prepared for all clients in the same chat room so that we replace the original server id with the chat room id. When a proxy replies data messages to NDN network, the receiver id of the XMPP message format should be replaced with the chat room id. In data messages that the proxy receives from NDN network, the sender id of the XMPP message format should be substituted by the chat room id, and the receiver id should be substituted by its served client id. Therefore, the application can handle data messages as usual and know that the incoming messages are sent from the chat room.

4.2.2 Interaction between the proxy and the NDN daemon

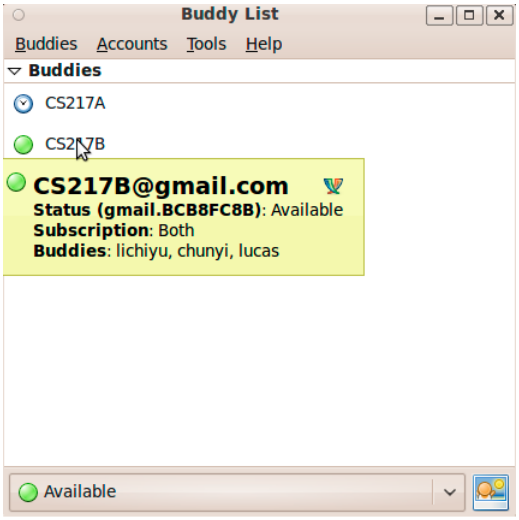

Figure 6: Buddy list

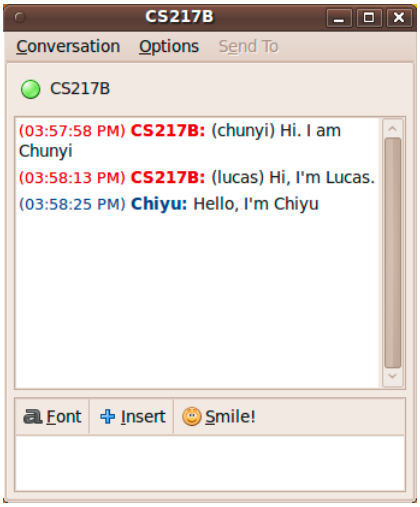

Figure 7: Chat window

With our new developed NDN functions, the proxy can communicate with others over NDN network through the NDN daemon. It sets up three connections with the NDN daemon using purple_ccn_connect: ccn_signal, ccn_read and ccn_write. Each client can periodically express its buddy interest to inform others of its existence through the ccn_signal connection. Expressing interests to pull data can be done through the ccn_read connection. The ccn_write connection can be used to put data to NDN network. Additionally, the proxy registers two callback functions for receiving interests and content: *Purple_ccn_ incoming_interest* and *Purple_ccn_incoming_content*.

The two interest prefixes that the proxy periodically expresses are *"ccnx://RoomID/AnotherUserID/SeqNum"* and *"ccnx://RoomID/SelfUserID"*. The first is used for pulling data message and controlled by the sequence number. The other one is sort of a keep-alive message used to reset timer on other members' buddy list. Thus the later interest will not pull data back. Each client would register an interest filter of the prefix *"ccnx://RoomID"*, so that it can receive existing users and newly joined ones' keep-alive interests.

4.3 Demonstration

We employ the Graphical User Interface (GUI) of Google Chat in pidgin to demonstrate our implementation of *NDNPurple* APIs and the proxy. Figure 6 shows the window of the chat room list. The original buddy list is replaced with the list of chatrooms, because a basic unit of the conference chat application is a chat room, rather than a buddy. The client can choose which chat room it wants to join and get each room's information from its tooltip. The tooltip of CS217B shows that Chiyu, Chunyi and Lucas are chatting in the room. Figure 7 shows the Chiyu's conversation window. With the aid of proxy, the Google Chat program still considers that it has a successful connection with a Google server and can accordingly work as usual without code-wise modification.

5. RELATED WORK

There is a rich literature regarding how to build a new data centric Internet architecture. A fundamental question in all the data-centric architecture designs is what to use as data identifiers. In [8] Adjie-Winoto *et al* propose to assign XML format meta-data to content and service. As a result, network routers perform content search through XML attributes match. While XML attributes based matching is flexible and expressive, CPU-intensive computation and routing scalability remain as open issues for global adoption. NetInf [9] and ROFL [10] propose to use cryptographic-based flat labels as content identifiers. Flat labels (often derived from hashing of content) remain constant during end-host mobility and are used to ensure content-level security. However, these designs require a mapping infrastructure to translate human-readable application names to flat labels, and how to secure the mapping system itself remains an open issue. Furthermore, flat labels cannot be aggregated as IP addresses or NDN hierarchical names in order to scale the routing system.

VoCCN [12] shows an example of how to construct NDN name space based on SIP [6] message headers and how to implement real time voice applications on NDN. VoCCN develops a new voice application implementation to utilize NDN, our work keeps the existing application's interface unchanged and only converted the transport substrate to use NDN.

6. CONCLUSION

While NDN is an exciting new architecture whose design

embraces the evolving requirements of the Internet, an important step in its deployment is the adaptation of existing applications to its architecture. In this work we have presented one of the first attempts to do this in a general way using libpurple. From this effort we have digested several key lessons that we feel will greatly inform future porting efforts: First, NDN requires each packet to have a unique name, yet existing applications, such as libpurple, often do not have unique names for every piece of data. Thus, deciding how applications must name their data is critical; a good name structure can greatly simplify the design. Second, in a multicast application, where more than one user may produce data simultaneously, each user should be assigned a unique name. This facilitates a design in which each packet can easily be identified by its user's name plus a sequence number. Finally, our experience was that embedding a pseudo-proxy within the existing libpurple state machine made the porting effort quite straightforward. We were, therefore, able to avoid modifying the existing application code by injecting a new NDN shim layer. This seemed to indicate that a general approach to porting network applications from host-to-host to NDN might be to embed pseudo-proxies.

This work represents a preliminary step towards the new NDN architecture, with an initial focus on designing the name space to meet the application's functional needs. Therefore we do not cover how content security is addressed in NDN. According to [13], NDN already associates a cryptographic key with each user name and uses that key to sign all data he or she produces. Besides, a large number of research issues remain to be addressed, not the least of which is the routing system scalability. We plan to address this issue, together with data security, in our future efforts.

7. REFERENCES

[1] A Border Gateway Protocol 4 (BGP-4). In *RFC 4271*.

[2] Comparison of instant messaging protocols. In *http://en.wikipedia.org/wiki/Comparison_of_instant_messaging_protocols*.

[3] Google talk. In *http://www.google.com/talk/*.

[4] Libpurple. In *http://developer.pidgin.im/wiki/WhatIsLibpurple*.

[5] RIP Version 2. In *RFC 2453*.

[6] SIP: Session Initiation Protocol. In *RFC 3261*.

[7] Skype P2P telephony explained. In *http://www.skype.com/intl/en-us/support/user-guides/p2pexplained/*.

[8] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system.

[9] B. Ahlgren, M. D'Ambrosio, C. Dannewitz, M. Marchisio, I. Marsh, and B. Ohlman. Design considerations for a network of information. *ReArch*, 2008.

[10] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, I. Stoica, and S. Shenker. ROFL: Routing on Flat Labels. *SIGCOMM*, 2006.

[11] D. D. Clark. The design philosophy of the darpa internet protocols. In *in Proceedings of ACM SIGCOMM (Computer Communications Review Vol 18, No*, 1988.

[12] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton, and R. L. Braynard. VoCCN: voice-over content-centric networks. In *ReArch '09: Proceedings of the 2009 workshop on Re-architecting the internet*, pages 1–6, New York, NY, USA, 2009. ACM.

[13] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *CONEXT*, Rome, Italy., 2009.

[14] E. P. Saint-Andre. Extensible messaging and presence protocol (XMPP): Core. In *RFC 3920*, 2004.