CCNSink: Application-Layer Middleware for TCP/IP and Content Centric Network Interoperability

Christopher A. Wood
woodc1@uci.edu

June 12, 2014

Abstract

With the growing presence of data streaming services and applications in today's Internet, content-centric networks (CCNs) are an increasingly attractive design alternative to the traditional IP-based host-oriented architecture. CCNs emphasize content by making it directly addressable and routable within the network, in contrast to addressable hosts and interfaces. This fundamental difference leads to vastly different mechanisms to publish and retrieve content and enable peer-to-peer communication. Named Data Networking (NDN) and its sibling implementation - CCNx - is one particular CCN design that has received considerable academic and industry attention. Despite the many promising benefits, there has been little research into the NDN deployment strategy. Clearly, incremental deployment is the only viable solution. During this integration phase, however, there will undoubtedly be a need for IP-based (NDN-based) hosts to communicate with and retrieve content from NDN-based (IP-based) hosts. To this end, we present CCNSink, a middleware application to support interoperability between these different networking architectures with minimal application and transport layer modifications via semantic translation between the communication mechanisms used in both networks. We discuss the implementation details at length and study the performance overhead induced by this gateway and the message RTT is studied in various communication settings.

1 Introduction

At its core, the design and architecture of today's Internet is a communication-based packet switching network. Since its inception, it has been retrofitted with a variety of transport and application layer protocols and middleware to support a growing set of consumer applications, such as the Web, email, and perhaps most importantly in recent times, media streaming services. The latter type of applications are bandwidth-intensive content distribution entities which leverage the underlying communication-based network as a distribution network, leading to a massive consumption of vital, and sometimes scarce, networking resources.

Content-centric networks (CCNs) are a new class of network architecture designs that aim to address this increasingly popular class of applications by decoupling data from its source and shifting the emphasis of addressable hosts and interfaces to content [3]. By directly addressing content instead of hosts, content dissemination and security can be "distributed" throughout the network in the sense that consumers requests for content are satisfied by *any* resource in the network (i.e., not necessarily the original producer). For example, routers close to consumers may cache content with a particular name and then satisfy all content requests that match the content's name. In-network caching and data-centric security measures are two of the defining characteristics of these new network designs.

As of today there are several content-centric networking proposals being explored as alternative designs to today's Internet; Named Data Networking (NDN) [6] is one of the more promising designs that is still an active area of active research (see www.named-data.net for more information). As a replacement for IP-based networks, the complete adoption of NDN, or any one of these designs, will realistically need to be done by slow and continual integration and replacement of IP-based networking resources with NDN-based resources. Currently, however, there is no engineering plan to support the IP/CCN integration without significant software modification and application, transport, or network layer source code modifications (e.g., integrating and using CCNx to communicate with CCN-based applications from TCP/IP hosts).

Consequently, the primary objective of CCNSink is to aid the integration of future content-centric networking resources into the existing IP-centric Internet by providing a middleware to support IP and NDN interoperability[1]. Application-layer traffic corresponding to protocols such as HTTP will be translated by middleware to correctly interface with NDN resources, thereby serving as a semantic gateway between these two fundamentally different networking architectures. Similarly, using a custom NDN-to-IP interest naming convention, NDN *interests* for content will be translated to messages adhering to application-layer protocols to traverse the IP network. CCNSink also serves to bridge isolated NDN resources. In this use case, two CCNSink bridges will leverage the features of the IP network to forward interests and the respective content across separated NDN "islands." The primary benefit of this semi-transparent middleware is that existing IP-centric applications need not be modified at any layer in the network stack to interoperate with NDN resources. Furthermore, NDN-based applications can communicate across physically partitioned networks so long as there exists CCNSink bridges between them.

The rest of this report is outlined as follows. Section 2 provides an overview of NDN and CCNx as they pertain to this work, and Section 3 highlights some of the motivations for the gateway and bridge. The design of CCNSink is presented in Section 4, followed by implementation and performance details in Section 5 and 6, respectively. Finally, we conclude with a discussion of unfinished work and avenues for further development in Section 7.

2 NDN/CCNx Overview

NDN is one of the three currently active NSF FIA (Future Internet Architecture) projects [6]. CCNx is a closely related and more commercially-oriented design backed by PARC [5][2]. The defining characteristic of both designs is that they decouple location of content from its original publisher. To obtain content, a consumer issues a request (called an *interest*) referencing the name of the desired content. An interest is routed based on the specified name, rather than a destination address, over a sequence of routers, each of which keeps state of the forwarded interest. Requested content might be found either in a router that cached it based on a prior interest, or at the producer. Regardless, each router is expected (though not mandated) to cache each content it forwards. In essence, router caches and addressable content enable NDN/CCNx to reduce congestion and latency by keeping content closer to consumers.

An *interest*, though intended to carry a meaningful (human-readable) URI-like name, can in fact carry arbitrary strings corresponding to any type of data, such as encoded binary. Upon receiving an interest, a router looks up the content by name in its local cache, deemed the *content store* (CS). A match in the CS causes the associated content to be forwarded downstream over the same interface upon which the interest arrived. Interests that do not match any cached content are stored in a *Pending Interest Table* (PIT) together with the incoming and the outgoing interfaces. The interest is forwarded based on the longest-prefix match in the local *Forwarding Information Base* (FIB) table. Multiple interests matching the same name are collapsed into a single PIT entry to prevent redundant interests being sent upstream. Once a content matching a PIT entry is received by a router, it is cached (unless the interest explicitly asks not to cache this content) and forwarded to all incoming interfaces associated with the PIT entry. Finally, the PIT entry is flushed.

Beyond the pull-model that guarantees symmetric interest and content flow, content-centric traffic in NDN and CCNx has strong security implications. Most importantly, security is coupled to content rather than the channel of distribution. All sensitive content must therefore be encrypted in a meaningful way so as to ensure confidentiality. Content integrity and origin authenticity are ensured by mandating that all content be digitally signed by its producers. As we will discuss in the following sections, this requirement plays a crucial role in the bridge component of CCNSink. Contrary to generating digital signatures, routers need not verify signatures as content flows through the network due to the obvious computational overhead. On the contrary, consumers are assumed to verify all content signatures and re-issue interests in the event that signature verification fails. Issues regarding signature verification and public key distribution are elaborated upon in [2].

[1] We assume that NDN *will not* be deployed over the IP network, as such a deployment scheme would nullify the need for this type of middleware between the two networks.

[2] Due to the similarity between NDN and CCN with respect to this work, we use the terms NDN and CCN interchangeably

3 Motvating Interoperable Heterogeneous Networks

Consider the typical hourglass network stack in IP-based networks as shown in left-hand image of Figure 1. This layered design with a thin-waist infrastructure (IP packets for traffic flow) is what enabled the Internet to grow and expand at such a rapid rate; higher layers in the protocol stack extend this communication medium with support for a variety of applications and networking features (e.g., reliable message traversal via TCP). While the NDN architecture introduces a fundamental paradigm shift in the way information is published and retrieved on a network, its design, shown at a high level in the right-hand image of Figure 1, borrows the same hourglass design as IP networks. Observe that upper layers of the network stack still promote the development of robust applications based on the underlying communication layers. The difference, however, is that network traffic flow management (i.e., to enable reliable and stable communication) and security are *built into* the network stack. These architectural differences mean that application, transport, and network layer protocol semantics in IP-based networks are distinct from protocol semantics in NDN networks. CCNSink is intended to bridge between IP and NDN networks by performing this semantic translation between protocols.

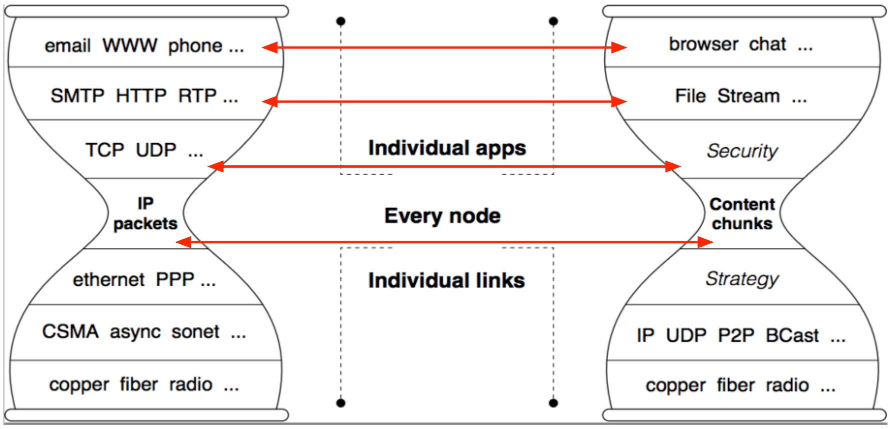

Figure 1: A visual comparison of the network stacks for the IP and NDN network architectures (figure from [6]).

Assuming that NDN is not to be deployed over IP, but instead as a separate network stack entirely, the need for such a gateway cannot be understated. Consider two instances of an application, A_1 and A_2, that wish to send data back and forth to each other. Application A_1 is running on a host with only an IP interface, and application A_2 is running on a host with only an NDN interface. What does it mean for application A_1 to establish a TCP connection stream with application A_2 and what does it mean for application A_2 to issue an interest to application A_1 when neither application speaks the network language of the other? In order for these two applications to communicate, the semantics of a TCP stream-oriented connection must be translated to a stream of contiguous interests, and vice versa. Now consider the alternative scenario in which applications on two or more physically disjoint NDN networks need to communicate to distribute content, but such networks can only be connected via the IP-based Internet. Strategically placed CCNSink gateways at the edges of these NDN networks can establish bridges across the Internet to enable cross-network interest issuance and content retrieval. Given these two seemingly unavoidable scenarios in an incremental deployment of NDN networks, CCNSink will enable continual application operation with minimal application and transport layer software changes. In the following sections we describe the gateway design that facilitates such interoperability.

4 Network Semantic Translations at the Gateway

As already emphasized, traditional IP-based applications and existing NDN-based applications treat both the network and content in significantly different ways. In IP-based settings, application and transport layer mechanisms and protocols leverage the underlying IP network layer to send packets to specific hosts. In contrast, in NDN-based settings, there are no straightforward application or transport layer analogs; the network layer, responsible for the issuance of interests and retrieval of content, is abstracted to authenticated (i.e., digitally signed) content objects or streams of data that are consumed by applications. From this perspective, there is no clear bijection between application and transport layer communication in IP-based settings and content and stream-centric data retrieval in NDN-settings. Therefore, to support the interoperability of these two networking paradigms, we need a mechanism to translate the semantics of IP-based communication to and from NDN-based content retrieval, as shown in Figure 4.

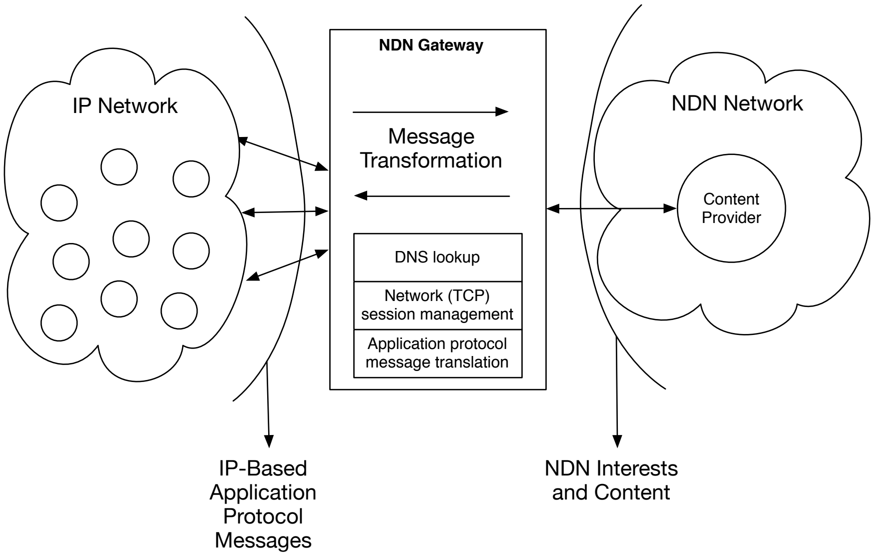
NDN Gateway
IP Network
NDN Network
Message Transformation
Content Provider
DNS lookup
Network (TCP) session management
Application protocol message translation
IP-Based Application Protocol Messages
NDN Interests and Content

Figure 2: CCNSink system context diagram.

The direction of traffic across the gateway has a strong influence on how this semantic translation is done: IP-to-NDN application and transport layer protocols will be mapped to corresponding NDN-interests, and NDN-to-IP traffic (in the form of interests) will be encoded so as to map to the appropriate IP-based application or transport layer protocol. Stateless protocols such as HTTP greatly simplify the job of the gateway because it need not maintain any state to support communication across both networks. However, stateful protocols, such as TCP, naturally require the CCNSink gateway to maintain state so as to emulate the behavior of an endpoint or host that implements such protocols. For instance, if an IP-based application wishes to establish a TCP connection with an application running on an NDN network to retrieve data, then the gateway must maintain stateful information needed to transform streams of data retrieved over a TCP socket to (packed) discrete and contiguous interests in the NDN network. In what follows we describe the semantic translation details for application and transport layer protocols in both directions across the gateway.

4.1 IP-to-NDN Semantic Translation

The current CCNx libraries enable IP-based hosts to communicate with applications running on NDN-enabled hosts. However, the engineering effort to retrofit the entire networking layer or subsystem of an existing application or system could be quite significant. CCNSink gateways are designed to minimize the effect of modifying the network components of such applications to still communicate with NDN hosts. In particular, they enable existing IP-based protocols to be used as is or with slight variations (e.g., TCP connections require additional overhead to setup) to communicate with NDN hosts through the gateway. In this way, the semantic translation of IP-based messages to NDN-based interests and content is offloaded to the gateway, rather than done in the application itself. In the following sections we describe the steps necessary to interoperate with the gateway at both the application and transport layers.

4.1.1 Application Layer

Translating IP-based application layer messages to NDN interests is highly dependent on the particular application protocol in question. There is an intuitive NDN-friendly encoding of HTTP GET requests in which human-readable URIs are parsed as interest names. Stateless application-layer protocols enable such direct mappings. Therefore, the gateway supports IP-to-NDN application layer messaging by encoding interests in HTTP GET requests as follows. NDN content objects with names "ccnx:/name/of/content" are issued via the gateway by sending a HTTP GET request with the following format to the gateway (in this example, the gateway IP address X.X.X.X is known or can be obtained via DNS):

```
GET X.X.X.X:80/ndn/ccnx/name/of/content
```

Since HTTP requests and NDN interests are stateless, the gateway will parse the URI of the request to determine that (1) it corresponds to an NDN interest, and (2) the full interest name is explicitly "ccnx:/name/of/content". Upon reception, the gateway will store the key-value pair $(\mathsf{name}, (\mathsf{source}-\mathsf{IP}, \mathsf{source}-\mathsf{port}))$ in an IP pending message table and issue an interest with the given name to the NDN network. Upon receipt of a piece of content, a NDN content handler callback recovers the IP address and port from the IP pending message table using the content name as the index, and then writes the raw content back to the client over the same incoming HTTP TCP connection. Since it is not required that the HTTP request uses a persistent TCP connection, the HTTP message handler is a synchronous procedure that blocks while the NDN interest is satisfied so that the same TCP connection can be used to write the response. If an NDN interest times out, the gateway returns an appropriate error code to the consumer.

4.1.2 Transport Layer

Since interests can be overloaded to contain arbitrary data, CCNSink gateways exploit this characteristic to carry transport-layer streams of data from applications on IP hosts to supporting applications on NDN-hosts. IP is a host-based protocol, however, and so establishing a "virtual" TCP connection between two such applications must be done in two steps: (1) a TCP socket from the IP client to the gateway must be established, and (2) the first chunk of data sent from the client must be the NDN host prefix to which data will be sent. The gateway parses this path and stores it internally for each open TCP socket. Let C be the IP client generating data in the socket socket from source address and port (A, P), and let G be the CCNSink gateway forwarding data on behalf of C. The connection establishment phase proceeds as follows:

1. C opens a TCP socket connection socket to G. Let id be a unique identifier for the TCP socket in both C and G.
2. C sends prefix/EOM to G over socket, where EOM is an end-of-message indicator.
3. G reads and parses data from socket, and stores the key-value pair $(\mathsf{id}, \mathsf{prefix})$ in the TCP connection map.

With prefix and id generated during the connection establishment phase, all subsequent data chunks are then forwarded to this NDN host using the following steps:

1. C sends a chunk of b bytes to G over the socket socket.
2. G reads b bytes of data from b and encodes it in Base64 format. Let data be the resulting encoded content.
3. G generates a uniformly random string nonce of k bits from $\{0,1\}^k$ (k is usually small - 32 bits - for short-lived TCP connections).
4. G retrieves prefix from the TCP connection map using the TCP socket ID id, uses it to build an interest with the name prefix/nonce/data, and issues the interest to the NDN network.

By incorporating a fresh random string in each new interest name, the probability that any issued interest will be satisfied from a network cache instead of the desired NDN host is negligible. Therefore, all data sent over the socket socket will reach the NDN host, and the receiving application can parse the last component of the interest as fresh data. Note that since this is strictly IP-to-NDN communication, there is no state information persisted in the session table. This is because the NDN host application may not respond with data since the gateway is forwarding *transport* layer data, unlike the case where application-layer queries are

⟨*ip-interest*⟩→ ’/. . . /ip/’⟨*protocol*⟩

⟨*protocol*⟩→ ’http/’⟨*http-cmd*⟩[’/’⟨*http-path*⟩] | ’tcp/’⟨*tcp-ident*⟩’/’⟨*uri-encoded-string*⟩

⟨*http-cmd*⟩→ ’GET’ | ’PUT’ | ’POST’ | ’DELETE’

⟨*http-path*⟩→ ⟨*uri*⟩ | ⟨*ip-address*⟩[port][’/’⟨*uri-encoded-string*⟩]

⟨*tcp-ident*⟩→ ⟨*SHA256-hash*⟩’/’⟨*nonce*⟩’/’⟨*public-key*⟩

Figure 3: NDN-to-IP application-layer translation encoding grammar.

forwarded. If the target application wishes to respond, it may do so by retrieving the TCP session identifier from the interest name and responding with a properly formatted interest as specified below in Section 4.2. Also, this design assumes the NDN application receiving data will always treat the last component of an interest name as the data sent from the IP application.

4.2 NDN-to-IP Layer Semantic Translation

Due to their architectural differences, there does not exist a native correspondence between NDN interests and IP-based application layer protocol messages. For example, there is no standard way for a client to represent an HTTP GET request in the format of an NDN interest. To make this type of semantic translation possible, the NDN-to-IP application layer bridge in CCNSink leverages the human-readable names of content to encode IP-based application layer protocol specifics. The grammar for encoding for HTTP and FTP semantic translations is specified in EBNF form in Figure 2; other application-layer protocols can easily be supported by extending this grammar in the natural way.

Interests encoded using this grammar are intercepted in the NDNInputStage of the gateway (see Figure 6.1). Upon reception, the gateway parses the message, stores a new entry in the NDN pending message table, and forwards the decoded message contents to the appropriate IPOutputStage pipeline stage. Upon reception, the IP response is retrieved in the IPInputStage and forwarded inwards to the NDNOutputStage, where the corresponding entry in the NDN pending message table is indexed using the contents of the arriving message to retrieve the original incoming interest name. Once fetched, the gateway creates and signs a new content object with the IP network response, and then forwards the content downstream to its intended consumer.

Unlike traditional NDN routing, the gateway pending message table does not collapse interests by default. The reason for this is that application-protocol interests are often issued when *new* state needs to change (i.e., cached responses not generated on demand from the intended IP recipient are not acceptable).

It is important to note that stateful protocols such as TCP must explicitly embed identifying information about the consumer in order to operate correctly. This is because one TCP stream must be associated with at most one consumer, and since NDN does not have any notion of host addresses or identifiers, the consumer application must explicitly encode its identity in the interest. Our grammar enforces identities based on consumer public keys and a corresponding digital signature of the entire interest for such protocols so that consumers can be explicitly identified and their sessions cannot be hijacked by other consumers (doing so would require compromising a consumer and its private key).

5 Bridging Isolated Networks

The second type of interoperability scenario that may arise during the deployment of NDN networks is when two or more physically disjoint NDN networks need to communicate with each other. Let I_i and I_j be two such disjoint NDN networks, and let A_i and A_j be two applications running on hosts in I_i and I_j, respectively. Without any additional mechanisms, A_i and A_j would not be able to communicate. However, if there existed two bridges B_i and B_j at the edges of I_i and I_j, each of which connected to the same IP-based network (i.e., the Internet), then interests from A_i (A_j) could be sent to A_j (A_i) as follows:

1. An interest from A_i is intercepted a bridge B_i.
2. B_i encapsulates the interest in an IP packet sent to bridge B_j.

3. B_j unwraps and re-issues the interest and waits for the content to be retrieved from A_j. Upon reception, the content's signature is verified and the content is signed and sent to G_i.
4. G_i verifies the signature of the packet, creates and signs a new content object, and sends the content object back downstream to A_i.

A visual depiction of this scenario is shown in Figure 5. To increase interest throughput across the bridge, each bridge uses persistent content-oriented TCP connections between other adjacent bridges. In order to authenticate content sent between bridge, we have the option of establishing a secure connection via SSL/TLS so that all content messages will be authenticated below the application-layer of the bridge, or we can manually sign and verify content separately from transport mechanism. Since NDN/CCN stipulates that content is only signed (as a form of authentication), we do not need the additional overhead of encrypting content as it moves between bridges. This is why we provide support for secure and insecure persistent connections.

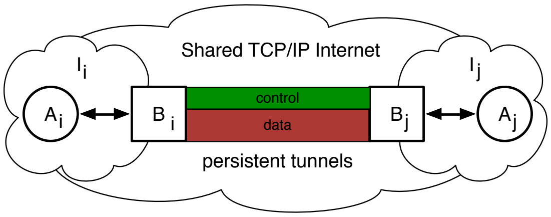

Figure 4: Visual depiction of a bridge between CCN islands.

In the latter case, one issue is the method for locating and retrieving cryptographic keys used for digital signatures. Our design permits three variations for authenticating content between bridges: (1) full PKI-based based digital signatures, (2) keyed MAC tags sourced by keys generated from authenticated symmetric key agreement protocols (i.e., Diffie Hellman key agreement), and (3) keyed MAC tags sourced by a *shared* symmetric key bridge group key generated using a group key agreement protocol (i.e., Tree-Based Group Diffie Hellman [4]). In what follows we describe each of these three variants in more detail; modifications to the content authentication procedure described above between bridges change in the obvious way (i.e., keyed MAC tags are replace digital signatures for MAC-based variants) Figure 5 shows a visual depiction of the variants (1) and (2).

Full PKI-based Digital Signatures: The only modification required for this variant is that bridges must be given the certificate for each bridge with which it will communicate. Certificates are exchanged and stored via the control channel between adjacent bridges.

Pair-Wise Keyed MAC Tags: In this variant, each pair of bridges will run the DH protocol to establish a shared common symmetric key to use for generating MAC tags. The DH protocol is run over the control channel between adjacent bridges. With n bridge routers, this variant requires $\frac{n(n-1)}{2}$ key pairs. MAC keys need to only be refreshed when they expire out or the bridge connectivity changes. New bridges can discover all other bridges by querying a central "gateway directory" server, obtaining a list of all corresponding IP addresses, and then initiating control channel connections with them.

Group-Based Keyed MAC Tags: When all groups share a common key, the "bridge directory" server is repurposed as a distributed director for all registered bridges that is responsible for coordinating group key agreement protocols (i.e., TGDH [4]). After a single invocation of TGDH, each gateway will possess the same MAC key k used for tagging and verifying content. Each individual bridge is also required to periodically send heartbeat messages to the director in order to maintain an active list of available bridges. The director will initiate new instances of the TGDH protocol to establish a new shared group key whenever bridge connectivity changes, i.e., when new bridges are added or time out.

Another issue is that incoming interests need to be directed to bridges connected to the desired endpoint network. We achieve this by maintaining a prefix-bridge mapping PTGMap in each bridge B_i about interest

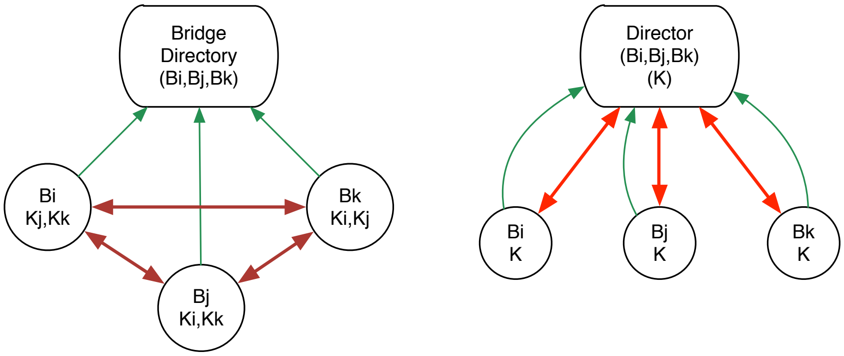

Figure 5: The left figure shows the deployment strategy with a fixed global directory that exists merely for obtaining bridge addresses, and the right figure shows the deployment strategy in which a distributed directory coordinates group key agreement protocols between all bridges.

prefixes and the associated bridges from which they can be satisfied. If an incoming interest to bridge B_i has no key in PTGMap, i.e., a previous interest with the prefix p was never received by B_i, then the interest is broadcast to all known bridges. Whenever a response is retrieved from a source bridge B_j, the key-value pair (p, G_j) is inserted into PTGMap. All subsequent interests with the prefix p received by bridge B_i will be sent directly to bridge B_j. If an interest times out, the PTGMap entry is removed, and the broadcast procedure is retried.

6 Implementation Overview

CCNSink is implemented entirely in Python using the PyCCN [1] wrapper around CCNx. Currently, all functionality except transport layer semantic translation is implemented. The design of CCNSink can be viewed from two perspective: the gateway component and the bridge component. Though the bridge component uses elements of the gateway to minimize redundant code, the designs of each are mostly independent. In this section we describe each of them in more detail.

6.1 Gateway Implementation

The core design of the gateway can be perceived as the composition of two flexible pipelines that route traffic from NDN (resp. IP) networks to IP (resp. NDN) networks (see Figure 6.1). Each pipeline begins and ends with an `InputStage` and `OutputStage`, respectively, one for the IP network and one for the NDN network. Each `InputStage` instance runs an appropriate interface to the network from which it receives traffic. Specifically, the `IPInputStage` runs an HTTP server to intercept IP-to-NDN interests, and the `NDNInputStage` registers a CCNx handle and configures an interest filter for incoming interests.

It is important to emphasize that the `NDNInputStage` operates independently of the underlying network implementation. In a true deployment, CCNx would not be used to interface with the NDN network. Rather, a CCN network stack would be implemented on top of the appropriate network interface controller (NIC). The mechanism for registering an interest filter on top of this network stack would change, but the functionality that happens after an interest is intercepted will remain the same. Using CCnx for the preliminary development of this gateway was necessary since there does not yet exist NDN NICs or CCN software stacks that are not built upon the TCP/IP stack.

After an input stage receives an incoming message (i.e., an IP packet or NDN interest), the asynchronous message handler will (1) allocate an entry in the *pending message table* for the network interface, (2) decompose the components of the message into its "raw form" and (3) save them in common message object wrapper, and (4) forward the resulting object to the next pipeline stage. For example, the `IPInputStage` will save the IP source host and port information in the `IPPendingMessageTable`, extract the URI path and

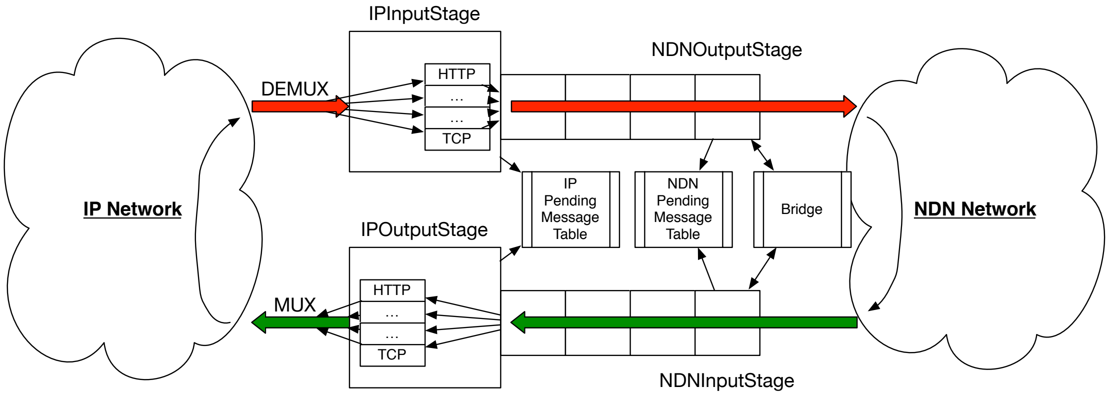

Figure 6: Bidirectional message pipeline for IP-to-NDN and NDN-to-IP message traversal.

set it as the "destination" field in the outgoing message object, and forward this message object to the next pipeline stage. In addition to the message source information for each entry in the pending message table, a binary semaphore is also stored. After forwarding the object, each handler will acquire a lock on this binary semaphore until the content associated with this message is retrieved from the target network.

The pipeline is designed so that each stage (with the exception of the `InputStage`) has a thread-safe input queue and a reference to the next stage (with the exception of the `OutputStage`). This simple interface and design enables any number of intermediate stages to be configured between the input and output stages. For simplicity, the current CCNSink implementation only uses two stages - input and output stages.

Once a message reaches the output stage, a new message for the target network is created and issued to the network. For example, an NDN interest will be formatted based on the contents of the outgoing message object retrieved from the output stage queue. After the target network returns content, the input stage of the target network performs the following tasks: First, the pending message table is checked for the an entry corresponding to the input message (e.g., an interest name that matches name of a previously issued interest). If an entry is found, the content field of the matching entry is populated and the binary semaphore associated with the semaphore is released. The latter step unblocks the original asynchronous input stage message handler, which then retrieves the content from the pending message table entry and sends it to the original consumer. If a matching entry in the pending message table is not found, the incoming message is treated as a "new" message, and it is formatted and flows through the pipeline in the opposite direction.

6.2 Bridge Implementation

As illustrated by Figure 6.1, the bridge component of CCNSink is designed to interoperate with the input and output stages of the gateway. The current implementation uses a central directory to manage bridge locations and status updates (see Section 5). Additionally, due to the small scale at which CCNSink was tested, the pair-wise MAC key management scheme is implemented. The design and implementation does not impede the integration of the more efficient TGDH protocol that is also discussed in Section 5. Each instance of CCNSink runs a separate thread for the bridge component that manages the following tasks:

- Sending periodic heartbeat messages to the bridge.
- Establishing and storing pair-wise MAC keys with all known bridges.
- Managing the PTGMap table.
- Controlling selective interest forwarding to specific bridges or broadcasting to all known bridges.

The bridge component in B_i leverages the `NDNInputStage` and `NDNOutputStage` in the following way: If the name of an arriving interest does not have the default gateway prefix as specified in Section 4, the interest is forwarded to the queue of the bridge thread. This bridge thread then extracts the interest from the queue and checks the PTGMap map for the interest prefix. If a match is found for prefix prefix, the interest name is sent to the mapped bridge $B_j = \mathsf{PTGMap}[\mathsf{prefix}]$ over a persistent TCP stream. If a prefix match is not found in the PTGMap, then the interest name is broadcasted to all bridges over their respective TCP streams. After sending this interest, B_i will spawn a new thread that blocks until receipt of a response $r = (c, t)$ from the target bridge B_j.

Upon receiving an interest at the target bridge B_j, an entry $(\mathsf{name}, \mathsf{socket})$ (where name and sock are the interest name and socket connection from which the interest was received) is created. The bridge then inserts this entry in a separate pending interest table so that the content response can be streamed back to the appropriate socket when retrieved. Then, an `OutgoingMessage` with the interest is created and inserted into the local `NDNOutputStage` queue. When content is eventually returned, the `NDNInputStage` will examine the bridge pending interest table for the interest name to retrieve the socket object that will be used to send the response $r = (c, t)$, where c is the content and t is $\mathsf{HMAC}(K_i, c)$ (i.e., the keyed MAC tag generated with key K_i associated with the recipient bridge B_i).

When B_i receives a response $c = (c, t)$ from a socket connected to B_j, the MAC tag is verified. If the tag is valid, the content c is returned to the original consumer and the content name prefixes are inserted into the PTGMap table along with the address of B_j. If the tag is invalid, the content is ignored and the PTGMap table is not updated.

6.3 Implementation Difficulties

There were several implementation difficulties encountered during the course of this project. For brevity, we enumerate them below.

1. Since TCP streams were used to transmit key data, interests, and content between bridges, we needed a simple way to encode and isolate each of these respective messages in the stream. To accomplish this task, we prepend each type of message with a single byte identifying the type of message to be sent (e.g., key data messages are prepended with the byte "k") and end each message with a
`n` character. The newline character is escaped in all content messages so as to avoid premature escapes and corrupt TCP streams. TCP message formatting
2. Interfacing with the bridge directory needed to be a simple task that should be handled independently of the programming language used to implement the bridge clients. Therefore, we encode update messages in HTTP POST messages that can be formulated from any bridge client. The Python Flask web server enables us to easily parse and handle these HTTP messages, and the use of an application-layer protocol to send these messages allows us to re-implement the bridge client in any programming language should the need arive. For example, if we need to implement the bridge in C/C++ for increased performance, the bridge directory HTTP API does not permit us from doing so.
3. Since we use the HMAC algorithm to tag and verify content sent between adjacent bridges, we needed to implement or find existing code for generating keys and computing this function. In the end, we settled on a hybrid approach in which the keys are generated using custom-written Diffie Hellman procedures and the HMAC function is computed using the Python `hmac` library. Separating key establishment from the HMAC computation gave us added flexibility in how key shares are generated by the bridge components (i.e., we could generate key shares on demand in TCP servers in response to key messages from a TCP client). Additionally, using the existing HMAC implementation reduced the likelihood of bugs in the critical cryptographic code.

7 Performance Evaluation

To evaluate the performance of CCNSink, we are concerned with (1) the overhead incurred by the semantic translation methods discussed in Section 4, (2) the message latency when crossing between different networks (e.g., the RTT of IP-to-NDN messages), and (3) the latency of messages traversing NDN bridges. Since directory updates happen infrequently and asynchronously, we did not measure the time to perform this task; we leave such evaluation to future work. To assess each of these measurements, we deployed a single bridge directory server that managed four (geographically) remote clients. Each of these hosts were running Ubuntu 12.10 and ran CCNx 0.81 and Python 3.1. All necessary cryptographic functions were implemented using Python standard libraries (i.e., the `hmac` Python library).

Measurements (1) and (2) were tested via scripts that issue a series of IP-to-NDN and NDN-to-IP requests and record the time to retrieve a response. More specifically, IP-to-NDN messages were generated and timed

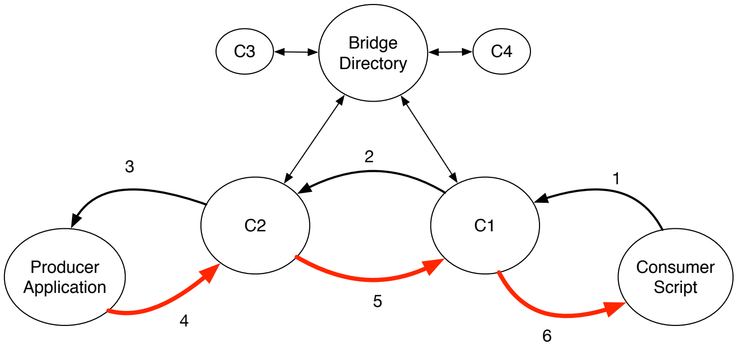

Figure 7: Experimental setup for measuring the bridge component RTT.

by a Python script using the `httplib` library, and NDN-to-IP messages were generated and timed by a script that issues interests using the PyCCN library. The semantic overhead time was recorded by the gateway component itself (i.e., not by the test scripts). The resulting overhead of translating IP to NDN messages was determined to be approximately 0.00078s for interest names composed of one to five components. Similarly, the overhead of translating NDN to IP messages was slightly lower with an average time of 0.005s. Clearly, the overhead of both translations is negligible for sequential requests generated by a small number of clients[3].

Since measurement (3) used the bridge component, we needed to test this using multiple machines. In this setup, two remote clients, C_1 and C_2, are connected to the central bridge directory and are used to issue and forward NDN interests. A script running on one client C_1 issued a series of NDN interests (via `ccnpeek`) to be forwarded to the other client C_2 and satisfied by its set of connected NDN hosts, and the RTT to retrieve content for each invocation of `ccnpeek` was recorded. This RTT includes the time to:

- Send the interest `ccnx:/test/`r_i from the consumer script to the bridge on C_1 (where r_i is the experiment iteration sequence number),
- Forward the interest from C_1 to C_2,
- Re-issue the interest from C_2 to the producing application,
- Retrieve the corresponding content from the producer at C_2,
- Sign and send the content from C_2 back to C_1, and
- Verify the content and send the result to the requesting consumer script.

The experimental setup and steps in this RTT calculation are shown in Figure 7.

The RTT results for measurements (2) and (3) when retrieving content of roughly 1MB and 10MB in size are shown in Figures 7 and 7, respectively, and the RTT result for measurement (3) when retrieving content of 100MB is shown in Figure 7, respectively. Our results indicate that IP-to-NDN and bridge message latencies were fairly consistent, whereas NDN-to-IP content retrieval incurred sporadic spikes in RTT. We attribute these anomalies to Python's thread synchronization primitives. We also note that the RTT times for the bridge are not worst-case in that they do include the preliminary TCP connection establishment and pair-wise key agreement overhead (using an appropriate Diffie Hellman group $\mathbb{Z}_p$, where p is 1024 bits). Establishing the shared key takes approximately 0.186s[4] and the time to establish a TCP connection is negligible. It is also interesting to note the flucuations in the brige RTT time for large (100MB) content. This is due to the nature in which content was generated by the producing application. In particular, the producing application responds to all interests with *random* bytes of data. In doing so, an arbitrary delay is also inserted so as to emulate real-world latency side-effects due to heavy loads, background application

[3] Due to a lack of resources, large-scale tests were not conducted to see how this overhead increased with request load.

[4] This average value was determined by timing the key establishment overhead for a small set of 10 experiments.

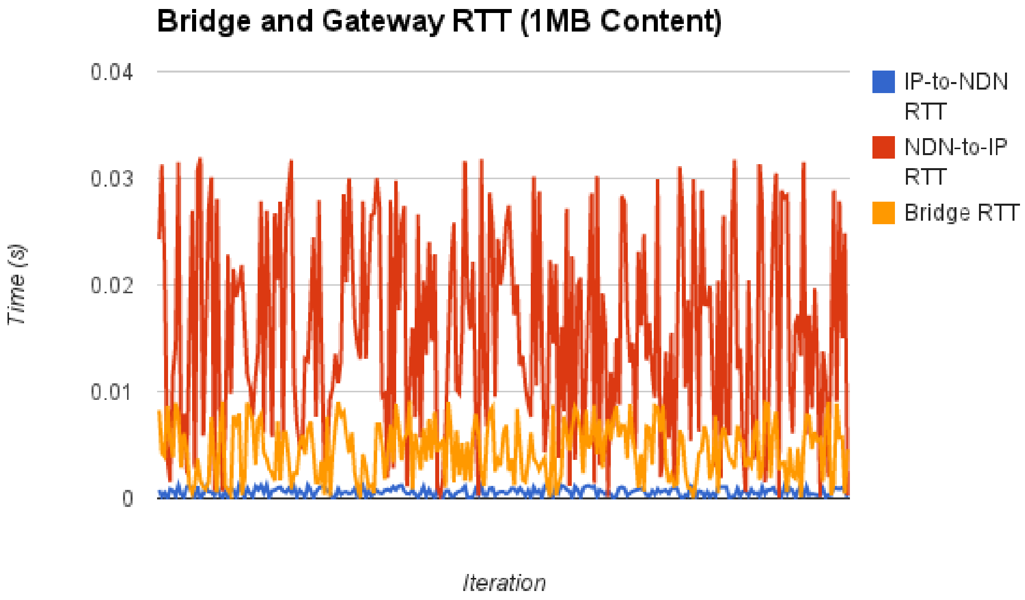

Figure 8: Average RTT times for IP-to-NDN, NDN-to-IP, and bridge messages when requesting content of approximately 1MB in size.

I/O, operating system context switches, etc. that could not be replicated in our small test environment. If this delay was removed, however, we would see much smoother RTT times for large pieces of content.

8 Conclusion

We presented CCNSink , a middleware application that enables TCP/IP and NDN/CCN network interoperability. CCNSink is expected to aid incremental deployment of pure NDN/CCN networking resources by allowing applications implemented on top of different networking stacks to communicate with each other almost transparently using existing protocols via the middleware; for example, the use of overlay network libraries such as CCNx does not need to be implemented in each TCP/IP application that wishes to communicate with applications running on NDN/CCN hosts. Furthermore, the design and implementation of CCNSink is simple and flexible enough to permit more meaningful semantic translations between different network protocols while introducing minimal performance overhead in message latency.

While the design is still rudimentary, there are several opportunities for improvement. From a design perspective, for example, the NDN-to-IP translation encoding grammar can be expanded to support additional TCP/IP application layer protocols, such as FTP, DNS, etc. The difficulty in encoding these types of application-layer protocols is that some of them require stored state and private client identification information. For example, if an anonymous FTP server is to be accessed via an NDN-based application, the CCNSink gateway must maintain a persistent FTP connection to the target server so that all issued commands can be executed. For example, the gateway must (1) establish a connection to the FTP server, (2) associate the connection with a *single* NDN consumer, which can be identified via a hash of their public key, and (3) use this connection whenever the NDN consumer issues new commands (e.g., “cd” commands). The current design does not include a storage for such application-layer protocol state, but this could easily be fixed by including a helper state management class that is referenced by the appropriate pipeline stage.

From an implementation perspective, the the bridge directory can be implemented as a set of distributed servers for increased performance under a large number of bridge clients generating heavy loads. Since we

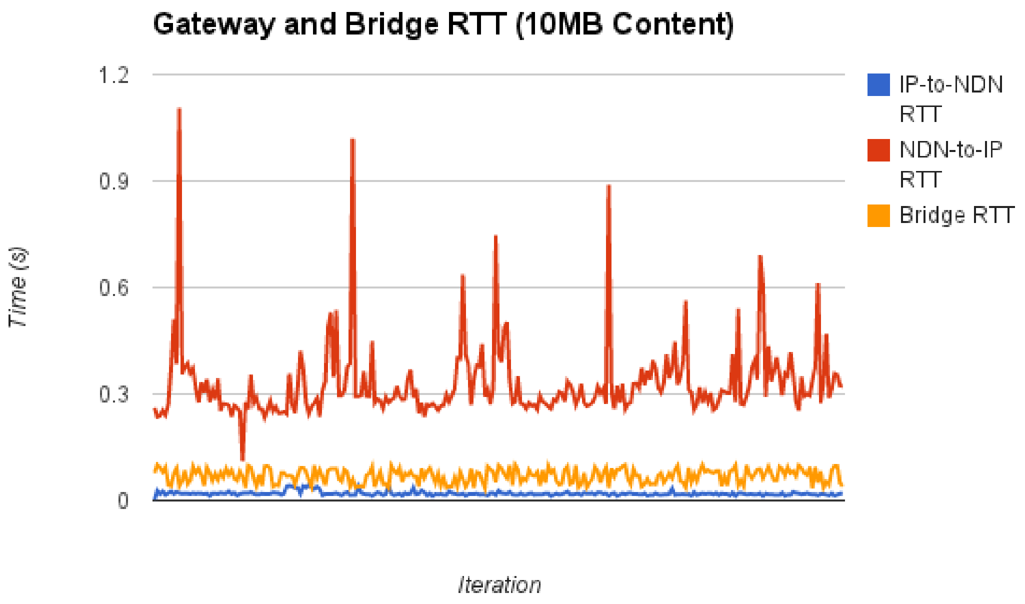

Figure 9: Average RTT times for IP-to-NDN, NDN-to-IP, and bridge messages when requesting content of 10MB in size.

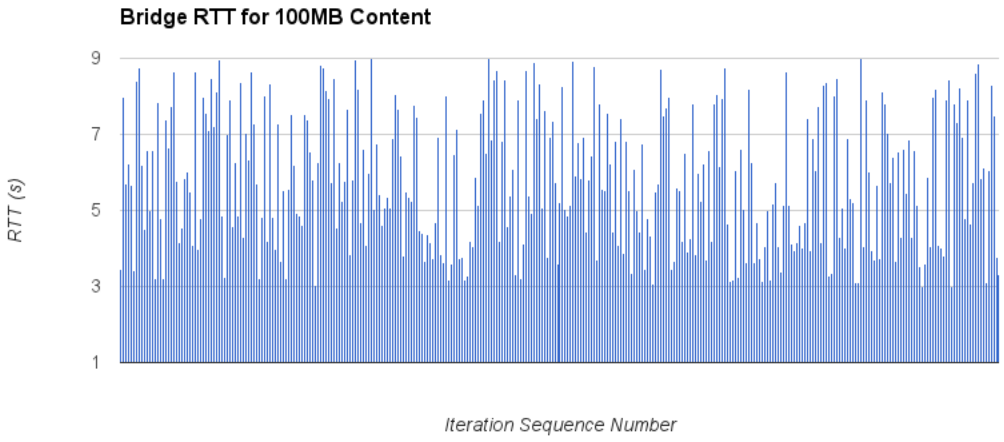

Figure 10: Average RTT time for bridge messages when requesting content of 100MB in size.

did not have access to a large set of clients to stress test the bridge directory, we are not sure how well the current implementation will scale. Such tests will also help us identify an appropriate frequency at which "heartbeat" messages are sent to the bridge directory. That is, if the design does not scale well under heavy loads, we can reduce the rate at which heartbeat update messages are sent to avoid self-induced denial-of-service attacks. Furthermore, pair-wise bridge keys can be replaced with shared group keys instantiated via group-key agreement protocols such as to TGDH. We are unaware of any open implementations of these cryptographic primitives, so we would need to construct them using the Python cryptographic libraries.

References

[1] J. Burke and D. Kulinski. PyCCN - Python CCNx Bindings. *University of California, Los Angeles*, https://github.com/named-data/PyCCN.

[2] C. Ghali, G. Tsudik, and E. Uzun. Elements of trust in named-data networking. *arXiv preprint arXiv:1402.3332*, 2014.

[3] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM.

[4] Y. Kim, A. Perrig, and G. Tsudik. Tree-based group key agreement. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):60–96, 2004.

[5] PARC. CCNx. Available online at: `https://github.com/ProjectCCNx/ccnx`, May 2014.

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