

# CCNSink: An Application-Layer Gateway for TCP/IP and Content Centric Network Interoperability

Christopher A. Wood  
University of California Irvine  
Irvine, CA, USA  
woodc1@uci.edu

## 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, an incremental deployment strategy 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 an IP/NDN gateway 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. The performance overhead induced by this gateway is studied in unidirectional and bidirectional communication settings.

## Categories and Subject Descriptors

H.4 [TODO]: TODO; D.2.8 [TODO]: TODO—*TODO*

## General Terms

NDN/CCN; CCN Deployment; Network Gateway

## 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 applications which leverage the underlying communication-based network as a distribution network, leading to a massive consumption of vital networking resources.

Information-centric networks (ICNs) are a new class of network architecture designs that aim to address this increasingly popular type of network traffic by decoupling data from its source and shifting the emphasis of addressable content from 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 information-centric networking proposals being explored as alternative designs to today’s Internet; Named Data Networking (NDN) [4] is one of the more promising designs that is still an active area of active research (see [www.named-data.net](http://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-to-ICN integration without significant software modification and application, transport, or network layer source code modifications.

Consequently, the primary objective of this work is to aid the integration of future content-centric networking resources into the existing IP-centric Internet by providing an application and transport layer gateway between IP and NDN resources. We assume that NDN *will not* be deployed over the IP network, as such a deployment scheme would nullify the need for a gate-

way between the two networks. Application-layer traffic corresponding to protocols such as HTTP, FTP, SMTP, IMAP, etc. will be translated by middleware running in such gateways to correctly interface with the NDN resources, thereby serving as a semantic bridge between these two fundamentally different networking architectures. Similarly, using a custom NDN-to-IP protocol, NDN *interests* for content will be translated to the messages adhering to the corresponding application-layer protocol to traverse the IP network. When serving as a bridge between islands of NDN resources, two gateways will leverage the features of the IP network to move messages among two separate NDN beds. Specifically, NDN interests may be encapsulated in UDP datagrams composed of IP packets traversing an IP network between two NDN islands. The primary benefit of this semi-transparent gateway is that existing IP-centric applications need not be modified at any layer in the network stack to interoperate with NDN resources.

The rest of this paper 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. CCNx is a closely related and more commercially-oriented design backed by PARC<sup>1</sup>. 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; see below. 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.

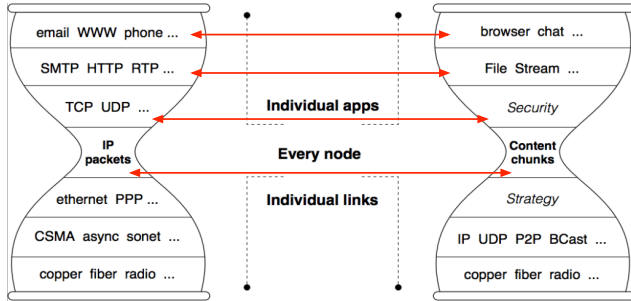
Beyond the pull-model that guarantees symmetric interest and content flow, content-centric traffic in NDN and CCNx has strong security implications. Most importantly, security is coupled to content rather than the channel of distribution. All sensitive content must therefore be encrypted in a meaningful 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 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 should signature verification fail. Issues regarding signature verification and public key distribution are elaborated upon in [2].

## 3. MOTIVATING INTEROPERABLE HETEROGENEOUS NETWORKS

Consider the typical hourglass network stack in IP-based networks as shown in left-hand image of Figure 3. 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 3, 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*

<sup>1</sup>Due to the similarity between NDN and CCN with respect to this work, we use the terms NDN and CCN interchangeably

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. The NDN gateway 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 [4]).**

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* and *B*, that wish to send data back and forth to each other. Application *A* is running on a host with only an IP interface, and application *B* is running on a host with only an NDN interface. What does it mean for application *A* to establish a TCP connection stream with application *B* and what does it mean for application *B* to issue an interest to application *A* 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 two or more physically disjoint NDN networks need to communicate to distribute content, but such networks can only be connected via the Internet. Strategically placed border gateways at the edges of these NDN networks can be used to establish a bridge 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, our IP/NDN bridge 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

Traditional IP-based applications and upcoming 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.

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 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 stateful and stateless 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 applications or systems could be quite significant. The gateway is designed to minimize the effect of modifying the network components of such applications to still communicate with NDN hosts. In particular, it enables 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, which are the two most common levels used by applications for point-to-point communication in the OSI network stack.

#### 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 IP address X.X.X.X of the gateway 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 (name, (source – IP, source – port)) in the 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 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.

#### 4.1.2 Transport Layer

Since interests can be overloaded to contain arbitrary data, our gateway design exploits 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  $S$  from source address and port  $(A, P)$ , let  $(pk, sk)$  be a public and private key pair owned by  $C$ , and let  $G$  be the gateway forwarding data on behalf of  $C$ . The connection establishment phase proceeds as follows:

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

With **prefix** and  $id$  generated during the 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  $S$ .
2.  $G$  reads  $B$  bytes of data from  $S$  and encodes it in Base64 format, denoted as **data**.
3.  $G$  generates a uniformly random string **nonce** of  $k$  bits from  $\{0, 1\}^k$ .
4.  $G$  retrieves **prefix** from the TCP connection map using the TCP socket ID  $id$ , uses it to build the interest **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  $S$  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 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.

## 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 leverages the human-readable names of content to encode IP-

based application layer protocol specifics. The encoding for HTTP and FTP semantic translations is specified in EBNF form below; 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 IOutputStage 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. The reason is that 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).

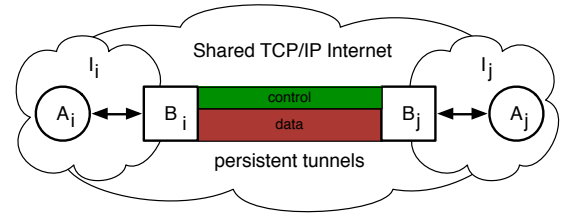
## 5. BRIDGING ISOLATED NETWORKS

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 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.  $G_i$  encapsulates the interest in an IP packet sent to bridge  $B_j$ .

3.  $G_j$  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 3: Visual depiction of a bridge between NDN islands.**

In the latter case, one issue is the method for locating and retrieving cryptographic keys used for digital signatures. If the content TCP socket is not Our design permits three variations for authenticating content between gateways: (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 gateway group key generated using a group key agreement protocol (i.e., Tree-Based Group Diffie Hellman). In what follows we describe each of these three variants in more detail; modifications to the content authentication procedure described above between gateways 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 gateways must be given the certificate for each gateway with

```

<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>
<tcp-param> → <uri-encoded-string>

```

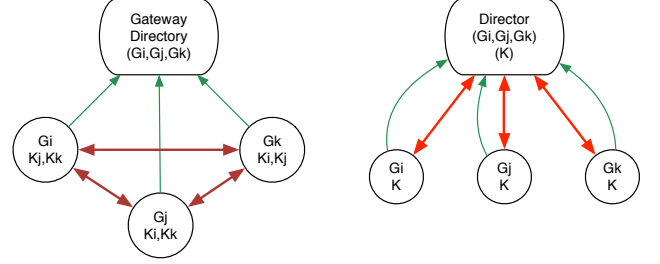
**Figure 2: NDN-to-IP application-layer encoding grammar.**

which it will communicate. Certificates are exchanged and stored via the control channel between adjacent gateways.

**Pair-Wise Keyed MAC Tags:** In this variant, each pair of gateways 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 gateways. With  $n$  gateway routers, this variant requires  $\frac{n(n-1)}{2}$  key pairs. MAC keys need to only be refreshed when they expire out or the gateway connectivity changes. New gateways can discover all other gateways 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 “gateway directory” server is repurposed as a distributed director for all registered gateways that is responsible for coordinating group key agreement protocols (i.e., TGDH). After a single invocation of TGDH, each gateway will possess the same MAC key  $k$  used for tagging and verifying content. Each individual gateway is also required to periodically send heartbeat messages to the director in order to maintain an active list of available gateways. The director will initiate new instances of the TGDH protocol to establish a new shared group key whenever gateway connectivity changes, i.e., when new gateways are added or time out.

Another issue is that incoming interests need to be directed to gateways connected to the desired endpoint network. We achieve this by maintaining a prefix-gateway mapping **PTGMap** in each gateway  $G_i$  about interest prefixes and the associated gateways from which they can be satisfied. If an incoming interest to gateway  $G_i$  has no key in **PTGMap**, i.e., a previous interest with the prefix  $p$  was never received by  $G_i$ , then the interest is broadcast to all known gateways. Whenever a response is retrieved from a source gateway  $G_j$ , the key-value pair  $(p, G_j)$  is inserted into **PTGMap**. All subsequent



**Figure 4: The top figure shows the deployment strategy with a fixed global directory that exists merely for obtaining gateway addresses, and the bottom figure shows the deployment strategy in which a distributed directory coordinates group key agreement protocols between all gateways.**

interests with the prefix  $p$  received by gateway  $G_i$  will be sent directly to gateway  $G_j$ . 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. Its design 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 the respective designs 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 NDN network interface controller. 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 allocate an entry in the *pending message table* for the network interface, decompose the components of the message into its “raw form” and save them in common message object wrapper, and 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 set it as the “destination” field in the outgoing message object, and forward this 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 enables any number of intermediate stages to be configured between the input and output stages. For simplicity, the `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 network responds with 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 lat-

ter 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 prefix match is found, the interest name is sent to the mapped bridge  $B_j$  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 upon receipt of a response  $r = (c, t)$  from the target bridge  $B_k$ .

Upon reception at the target bridge  $B_j$ , an entry (`name, 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 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  $\text{HMAC}(K_i, c)$  (i.e., the keyed MAC tag generated with key  $K_i$  associated with

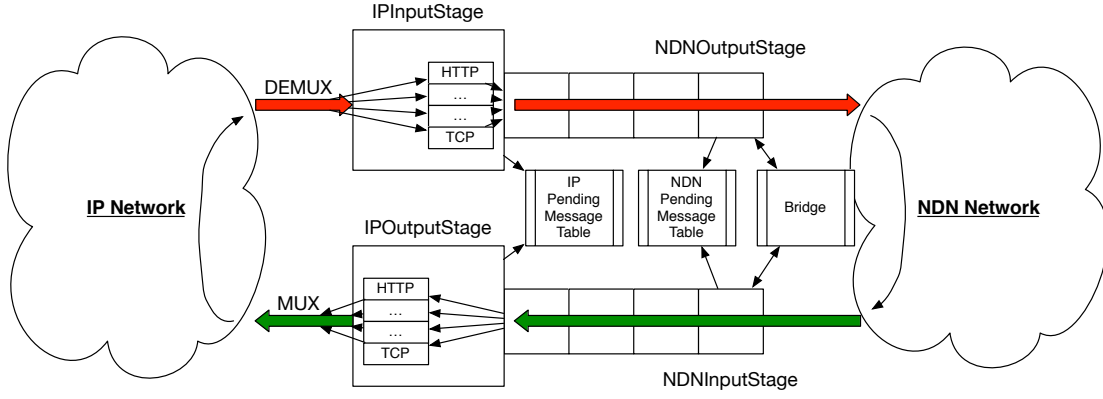
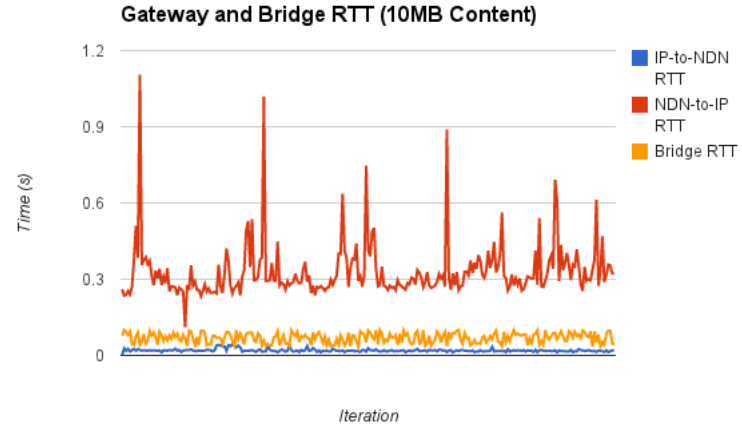
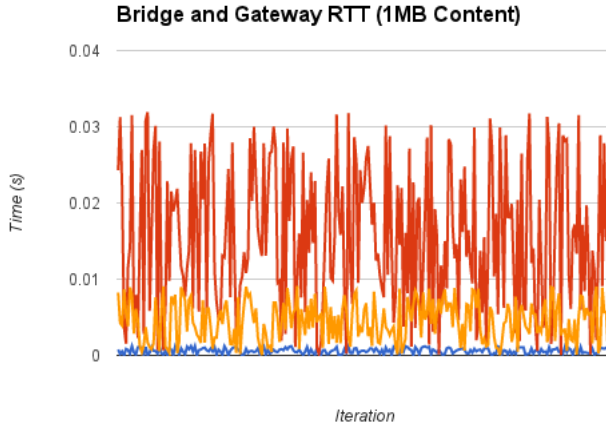


Figure 5: Bidirectional message pipeline for IP-to-NDN and NDN-to-IP message traversal.



the recipient bridge  $B_i$ ).

When  $B_i$  receives a response  $c = (c, t)$  from a socket connect 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.

## 7. PERFORMANCE EVALUATION

- test machine setup - experimental procedures and applications - link to source code - message translation overhead both ways - unidirectional and bidirectional RTT - bridge latency - say key generation and directory updates are asynchronous and one-time (don't happen a lot, so we didn't measure them)

## 8. CONCLUSION

TODO

## 9. 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] 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.