Application-Layer Gateway for IP and Content Centric Network Interoperability

Glenn Scott* and Christopher Wood*†
*Palo Alto Research Center, Palo Alto, CA 94304
†Department of Computer Science, University of California, Irvine, CA 92617
Email: Glenn.Scott@parc.com, woodc1@uci.edu

***Abstract*—With the growing presence of data streaming services and applications in today's Internet, information-centric networks (ICNs) are an attractive alternative to the traditional IP-based, point-to-point architectures. In contrast with the host-based nature of addressable interfaces, ICNs make data directly addressable and routable within the network. This difference leads to different mechanisms to publish and retrieve data and enable peer-to-peer communication. Interoperability between the IP-based, point-to-point networks and CCNs are an important step toward deploying the benefits of ICNs without disrupting existing IP networks. We present CCNSink, an application layer middleware gateway providing interoperability between CCNs and IP. CCNSink provides 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 round-trip-time is studied in various communication settings.**

I. Introduction

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

Content-centric networks (CCNs) are a class of network architecture designs that decouple the content from its source and shifting the emphasis of addressable hosts and interfaces to the actual data itself [10]. By directly addressing content instead of hosts, content dissemination and security is distributed throughout the network in the sense that consumer requests for content may be satisfied by *any* participant in the network (i.e., not necessarily the original producer). For example, network routers close to consumers may cache content and then satisfy all content requests that match that 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; Named Data Networking (CCN) [13] is one of the more promising designs that is still an active area of active research (see www.ccnx.org for more information).

The adoption of CCN, or any one of these designs, will likely be done by incremental integration or even replacement of IP-based networking resources with CCN-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 CCN interoperability. Application-layer traffic corresponding to protocols such as HTTP will be translated by middleware to correctly interface with CCN resources, thereby serving as a semantic gateway between these two fundamentally different networking architectures. Similarly, using a custom CCN-to-IP interest naming convention, CCN *interests* for content will be translated to messages adhering to application-layer protocols to traverse the IP network. CCNSink also serves to bridge isolated CCN resources. In this use case, two CCNSink bridges will leverage the features of the IP network to forward interests and the respective content across physically separated CCN "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 CCN resources. Furthermore, CCN-based applications can communicate across physically partitioned networks so long as there exists CCNSink bridges between them.

The rest of this paper is outlined as follows: Section 2 provides an overview of CCN 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.

II. CCN Overview

CCN is a commercially-oriented information-centric network architecture design backed by PARC [17]. As an ICN, its defining characteristics are that it decouples the location of content from its original publisher and the security of content from the channel through which it is delivered.

To obtain content, a consumer issues a request, called an *Interest*, specifying the name of the desired content. The name is a structured, hierarchical path-name, much like a URI.

An Interest is routed, based on the specified name, toward an authoritative producer of content for that name, rather than a destination address. As the Interest traverses the network, each router examines the name to determine if it has a copy of the content specified stored in its Content Store (CS). If it does, the router transmits that content in reply to the Interest. Otherwise, the router records some state derived from the forwarded Interest in a Pending Interest Table (PIT) in order to provide a backward path for the reply to the requester. Finally the router transmits the Interest to the next hop specified in its Forwarding Information Base (FIB). Looking up the next hop in the FIB table takes advantage of the hierarchical structure of the content name and locates the longest possible prefix of the name in the FIB table to choose the next hop.

If a router receives multiple Interests in the same named content and it cannot reply from its Content Store, and it has already forwarded a previous Interest upstream, it may simply update the PIT entry recording the multiple reverse paths and await the single response from upstream and reply to all reverse paths for the named content.

As the Interest traverses the network toward the producer, a router that has cached the specified content may transmit that content in reply to the Interest thereby truncating the routing path of the Interest. Ultimately, if no router can reply to the Interest, the Interest arrives at the producer as the last routable hop of the Interest's path.

In turn, as the reply to an Interest traverses the reverse-path and the producer indicates in the reply that the content may be cached, routers may choose to cache the reply in anticipation of a future Interest in the same named content. Router caches and addressable content enable CCN to reduce congestion and latency by keeping content closer to consumers.

Beyond the pull-model that guarantees symmetric Interest and content flow, content-centric traffic in CCN has strong security implications. Notably, security is coupled to content rather than its distribution channel. All sensitive content must therefore be encrypted in a meaningful way so as to ensure confidentiality. Content integrity and origin authenticity are ensured via digitally signatures generated by the content producers[1]. 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. Consumers, however, 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 [9].

[1]This is not a requirement, as the authenticity of content may be ascertained by specifying the cryptographic hash digest of the content expected to be returned. CCN leverages Manifest content objects to publish hash digests so that Interest messages with self-certifying names can be issued and extensive signature verification can be avoided. See [17] for more details about this content retrieval strategy.

III. 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 CCN 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/transport 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 differences mean that application, transport, and network layer protocol semantics in IP-based networks are vastly different from protocol semantics in CCN networks. CCNSink is intended to enable interoperability between IP and CCN networks by performing this semantic translation between protocols.

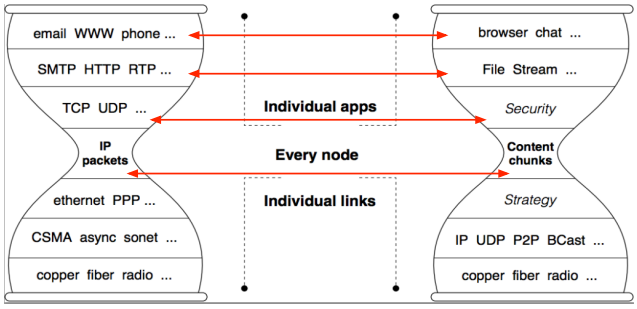

Fig. 1. A visual comparison of the network stacks for the IP and CCN network architectures (figure from [13]).

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 a CCN interface. When neither application speaks the network language of the other, what does it mean for application A_1 to post data to A_2 via HTTP, and what does it mean for application A_2 to issue an Interest to application A_1? In order for these two applications to communicate, the semantics of HTTP must be translated to appropriately named interests, and vice versa.

Now consider an alternative scenario in which applications on two or more physically disjoint CCN networks need to communicate to publish and retrieve content, but such networks can only be connected via the IP-based Internet. Strategically placed CCNSink gateways at the edges of these CCN networks can be used to establish bridges across the

Internet to enable cross-network Interest issuance and content retrieval.

Given these two seemingly highly plausible scenarios in an incremental deployment of CCN networks, CCNSink will enable continual application operation with minimal application and transport layer software changes. Furthermore, it will enable modern CCN applications to communicate with and retrieve content from legacy IP-based applications. In the following sections we describe the gateway design that facilitates such interoperability.

IV. The Application-Layer Gateway

Traditional IP-based applications and existing CCN-based applications treat both the network and content in significantly different ways. In IP-based settings, application and transport layer mechanisms and protocols use the underlying IP network layer to send packets to specific hosts. In contrast, in CCN-based settings, there are no straightforward IP-based application or transport layer analogs; the transport layer, responsible for the issuance of Interest messages and content retrieval, is abstracted via network-agnostic APIs to expose individual 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-oriented data retrieval in CCN-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 CCN-based content retrieval.

The direction of traffic across the gateway has a strong influence on how this semantic translation is done: IP-to-CCN application and transport layer protocols will be mapped to corresponding CCN-interests, and CCN-to-IP traffic (in the form of interests) will be encoded so as to map to the appropriate IP-based application layer protocol. We discuss these translations in the following subsections.

A. IP-to-CCN Semantic Translation

Translating IP-based application layer messages to CCN interests is simple. Due to the name-based routing mechanics of CCN, as well as the statelessness of HTTP, IP-to-CCN traffic is accomplished by encoding interests in HTTP GET commands. For example, a (legacy) CCNx content object with the name `ccnx:/name/payload` is retrieved *through the gateway* by sending a HTTP GET request with the following format to the gateway[2]:

```
GET X.X.X.X:80/protocol=ccnx/name/payload
```

Since HTTP requests and CCN interests are stateless (i.e., not dependent on previous or future requests or Interest messages), the gateway will parse the URI of the request to determine that (1) it corresponds to an CCN 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 CCN network. Upon receipt of a piece of content, a CCN 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 CCN interest is satisfied so that the same TCP connection can be used to write the response. If an CCN interest times out, the gateway returns an appropriate error code to the consumer.

[2]In this example, the gateway IP address X.X.X.X is known or can be obtained via DNS.

⟨*ccn-name-to-uri*⟩→ '/. . . /ip/'⟨*ip-protocol*⟩
⟨*ip-protocol*⟩→ 'http/'⟨*http-cmd*⟩[URI]
⟨*http-cmd*⟩→ 'GET' | 'PUT' | 'POST' | 'DELETE'

Fig. 2. CCN-to-IP application-layer translation encoding grammar.

The modern CCNx architecture – CCNx 1.x – provides an entirely separate field to carry an Interest payload. This precludes the need to embed such state in the name of an Interest. To use this particular field, the stateless HTTP POST command is used to carry Interest payload information. For example, if an IP-based application desires content with the name `ccnx:/name` and wishes to send the `payload` data in the Interest payload field, the corresponding HTTP POST command would formatted as follows:

```
POST X.X.X.X:80/protocol=ccnx/name --data
                payload
```

The gateway parses this request in the exact same manner as prescribed above.

B. CCN-to-IP Layer Semantic Translation

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

Interest messages encoded using this grammar are intercepted in the `CCNInputStage` of the gateway (see Figure VI-A). On reception, the gateway parses the message, stores a new entry in the CCN pending message table, and forwards the decoded message contents to the appropriate `IPOutputStage` pipeline stage. On reception, the IP response

is retrieved in the `IPInputStage` and forwarded inwards to the `CCNOutputStage`, where the corresponding entry in the CCN 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 CCN 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).

V. Bridging Isolated Networks

The second type of interoperability scenario that may arise during the deployment of CCN networks is when two or more physically disjoint CCN networks need to communicate with each other. Let I_i and I_j be two such disjoint CCN 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 V. 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 the transport mechanism[3].

In the latter case, one problem 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 [11]). We describe each of these three variants in more detail below; modifications to the content authentication procedure described above between bridges change in the obvious way (i.e., keyed MAC tags replace digital signatures for MAC-based variants). Figure V shows a visual depiction of variants (2) and (3).

[3]Since CCN stipulates that content is only signed as a means 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.

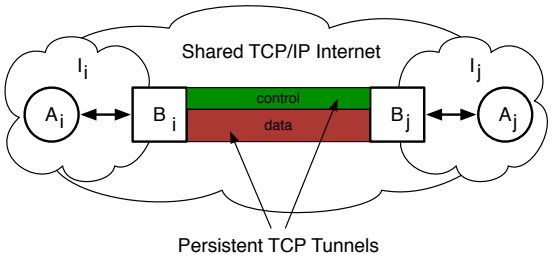

Fig. 3. Visual depiction of a bridge between CCN islands.

(1) 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.

(2) 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 $[n(n-1)]/2$ 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 "bridge directory" server, obtaining a list of all corresponding IP addresses, and then initiating control channel connections with them.

(3) Group-Based Keyed MAC Tags: When all groups share a common key, the "director" server is repurposed as a distributed director for all registered bridges that is responsible for coordinating group key agreement protocols (i.e., TGDH [11]). 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 PrefixTable in each bridge B_i. This table is functionally

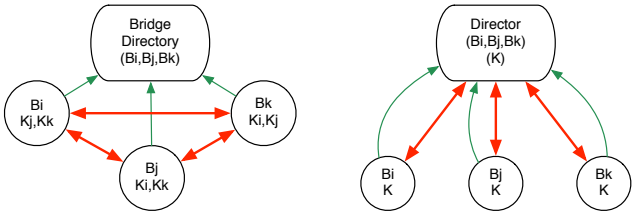

Fig. 4. The left figure shows the deployment strategy with a fixed global directory that exists merely for obtaining bridge addresses, and the right figure shows the deployment strategy in which a distributed directory coordinates group key agreement protocols between all bridges.

equivalent to a FIB in a CCN router; longest-prefix matching is used to guide incoming interests to the appropriate target bridge. If an incoming interest to bridge B_i has no key (entry) in PrefixTable, 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 PrefixTable. 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 PrefixTable entry is removed, and the broadcast procedure is retried.

VI. Design and Implementation Overview

CCNSink is implemented entirely in Python using the PyCCN [1] wrapper around CCNx 0.81 [14]. Currently, all functionality except transport layer semantic translation is implemented. The design of CCNSink can be viewed from two perspectives: 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.

A. Gateway Design

The core design of the gateway is the composition of two flexible pipelines that route traffic from CCN (resp. IP) networks to IP (resp. CCN) networks (see Figure VI-A). Each pipeline begins and ends with an `InputStage` and `OutputStage`, respectively, one for the IP network and one for the CCN 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-CCN interests, and the `CCNInputStage` registers a CCNx handle and configures an interest filter for incoming interests.

It is important to emphasize that the `CCNInputStage` operates independently of the underlying network implementation. In a true deployment, CCNx would not be used to interface with the CCN 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 CCN 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 CCN 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 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 CCN 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.

B. Bridge Design

As illustrated by Figure VI-A, 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 V). 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 Tree-Based Group Diffie-Hellman (TGDH) protocol that is also discussed

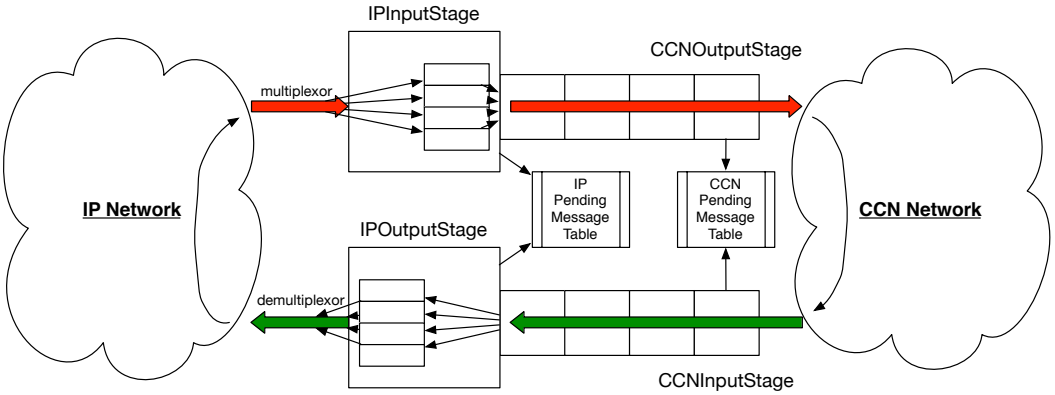

Fig. 5. Bidirectional message pipeline for IP-to-CCN and CCN-to-IP message traversal.

in Section V. 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 PrefixTable mapping.
- Controlling selective interest forwarding to specific bridges or broadcasting to all known bridges.

The bridge component in B_i uses the `CCNInputStage` and `CCNOutputStage` in the following way: If the name of an arriving interest does not have the default gateway prefix as specified in Section IV, the interest is forwarded to the queue of the bridge thread. This bridge thread then extracts the interest from the queue and checks the PrefixTable map for the interest prefix. If a match is found for prefix prefix, the interest name is sent to the mapped bridge $B_j = \mathsf{PrefixTable}[\mathsf{prefix}]$ over a persistent TCP stream. If a prefix match is not found in the PrefixTable, 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 `CCNOutputStage` queue. When content is eventually returned, the `CCNInputStage` 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 PrefixTable table along with the address of B_j. If the tag is invalid, the content is ignored and the PrefixTable table is not updated.

VII. 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-CCN messages), and (3) the latency of messages traversing CCN bridges. Since bridge key generation and directory updates happen infrequently and asynchronously, we did not measure the time to perform these tasks; 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. Cryptographic libraries such as OpenSSL were used for MAC tag generation and verification.

Measurements (1) and (2) were tested via scripts that issue a series of IP-to-CCN and CCN-to-IP requests and record the time to retrieve a response. The overhead of translating IP to CCN messages was approximately 0.00078s for interest names composed of one (1) to five (5) components. Similarly, the overhead of translating CCN to IP messages was slightly higher with an average time of 0.005s. Clearly, this overhead is negligible for sequential requests generated by small loads. Due to a lack of resources, large-scale tests were not conducted to see how this overhead increased with request load. Measurement (3) was tested by two remote clients connected to the central bridge directory. A script running on one client issued a series of CCN interests (via `ccnpeek`) to be forwarded to the other client and satisfied by its set of connected hosts, and the RTT to retrieve content for each invocation of `ccnpeek` was recorded.

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.

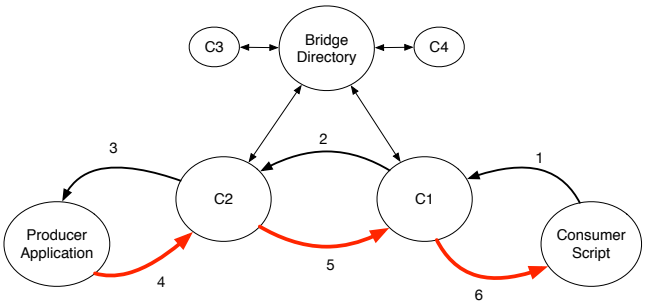

Fig. 6. Experimental setup for measuring the bridge component RTT.

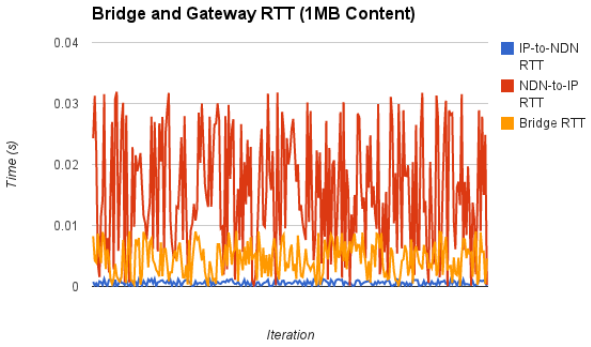

Fig. 7. Average RTT times for IP-to-CCN, CCN-to-IP, and bridge messages when requesting content of approximately 1MB in size.

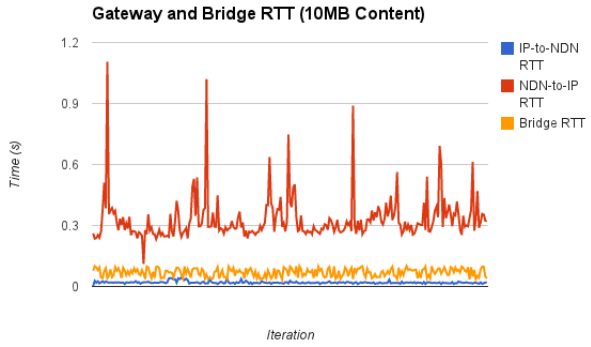

Fig. 8. Average RTT times for IP-to-CCN, CCN-to-IP, and bridge messages when requesting content of 10MB in size.

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 tt 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 VII.

The RTT results for measurements (2) and (3) when retrieving content of roughly 1MB and 10MB in size is shown in Figures VII and VII. Our results indicate that IP-to-CCN and bridge message latencies were fairly consistent, whereas CCN-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 with elements of size 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 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.

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

VIII. Related Work

The NDNPurple middleware presented in [18] is one attempt to deal with the issue of TCP/IP application evolution for CCN networks. This NDN-based middleware is designed to allow existing TCP/IP instant messaging applications based on the XMPP protocol, which is implemented on top of the libpurple library, to operate over an NDN network. To accomplish this task, NDNPurple re-implemented the networking layer of the libpurple library to interface with an NDN proxy that communicates with other proxies for other parties in a single "chat room." The proxy is responsible for marshalling buddy-list update requests and overloaded message content interests to all communication proxies. Additionally, the proxy is responsible for intercepting XMPP control messages (e.g., session keep-alive message) issued from the libpurple library and responding with the appropriate response. This proxy-based design enables chatrooms to be created in an ad-hoc manner, i.e., without a centralized server that manages chatroom session information.

Wang et al. [19] proposed a similar HTTP-based gateway to facilitate IP-to-CCN traffic *only* (CCN-to-IP traffic was not handled). Their middleware – composed of an ingress and egress gateway – uses a similar protocol conversion technique (e.g., HTTP URI to CCN naming conversion)[5]. The primary use case is to support seamless transmission of CCN messages (interest and content objects) across IP/CCN network boundaries. For example, if a CCN-based consumer was to issue an interest for content generated by an IP-based producer, the gateway would facilitate the translation of the interest into a corresponding HTTP GET command and also wrap the response in a content object. Bursztynowski et al. [2] proposed a similar gateway. Both works, however, neglect the issue of content object signature generation to extend the authenticity guarantees from the IP-based producer to the CCN-based consumer. Furthermore, there is no support for (securely) bridging isolated CCN networks over IP.

CCNxTomcat [15] attempted to solve the problem of application evolution without the use of a translational proxy. In this work, the CCNx 0.8x code was integrated into the Tomcat web server to seamlessly provide a basis for CCN- and HTTP-based web applications. The utility of their design was tested by deploying a real CCN-based web application on top of CCNxTomcat in an experimental CCN network. Performance-wise, their experimental results indicate that CCNxTomcat outperforms related proxies by 58% in a single request. While efficient, the server-based solution is limited in its ability to bridge CCN islands; it only provides application-level support for CCN functionality. Additional provisions would be needed to extend this application-level support to communicate with external CCN resources.

Another client-based solution for solving the problem of application evolution was proposed in [16]. Shang et al. presented a scheme based on the NDN Javascript library (NDN.js) and WebSocket protocol [8]. Their design communicates with a WebSocket proxy via NDN.js to access CCN network. This solution is limited, however, as it only permits Interest messages to be issued from a web application. It does not permit producer applications to be developed using Javascript.

The primary features of CCNSink - performing IP-to-CCN and CCN-to-IP translation, and bridging isolated CCN networks – are not unique to the development and deployment of information-centric networking architectures. The problem of inter-operating isolated networks is quite prominent with the continued deployment of IPv6. Clercq et al. [6] describe an approach to connect IPv6 islands over a Multiprotocol Label Switching (MPLS) enabled IPv4 cloud. The main idea is that core routers host two stacks – one for IPv6 and one for IPv4 traffic – and use a MPLS IPv4 core to translate IP packets between each of the respective protocols. This idea has been extensively used in practice [7], but is not the only option. IPv4 VPNs have also been leveraged to transmit IPv6 traffic [5], [7], [4]. In particular, VPN tunnels carry IPv6 traffic between islands in a manner similar to how interest and content objects are encapsulated in CCNSink.

The problem of protocol translation is well studied. Lam et al. [12], [3] and many of the works since then have studied the problem formally. In particular, given two processes P and Q which communicate using two distinct protocols, the problem is to find a converted C that is capable of supporting interoperability between them. In the context of CCNSink, we solve a similar problem of converting between IP and CCN using a message-passing state machine.

IX. Conclusion

We presented CCNSink, an application that enables TCP/IP and CCN network interoperability. CCNSink is expected to aid incremental deployment of pure 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 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 CCN-to-IP translation encoding grammar can be expanded to support additional TCP/IP application layer protocols, such as FTP, DNS, etc. 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. Furthermore, pair-wise bridge keys can be replaced with shared group keys instantiated via group-key agreement protocols such as to TGDH.

[5]The ingress gateway is deployed at consumers and the egress is deployed at producers.

References

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

[2] D. Bursztynowski, M. Dzida, T. Janaszka, A. Dubiel, and M. Rowicki. Http/ccn gateway and cooperative caching demonstrator. *Technical Report on CCNx Community Meeting*, 2012.

[3] K. L. Calvert and S. S. Lam. Formal methods for protocol conversion. *Selected Areas in Communications, IEEE Journal on*, 8(1):127–142, 1990.

[4] Y. Cui, J. Wu, X. Li, M. Xu, and C. Metz. The transition to ipv6, part ii: The softwire mesh framework solution. *Internet Computing, IEEE*, 10(5):76–80, 2006.

[5] J. De Clercq, D. Ooms, M. Carugi, and F. Le Faucheur. Bgp-mpls ip virtual private network (vpn) extension for ipv6 vpn. *RFC4659, September*, 2006.

[6] J. De Clercq, D. Ooms, S. Prevost, and F. Le Faucheur. Connecting ipv6 islands over ipv4 mpls using ipv6 provider edge routers (6pe). *Internet Engineering Task Force RFC*, 4798, 2007.

[7] H. Esaki. Ipv6 integration and coexistence strategies for next-generation networks. *IEEE Communications Magazine*, page 89, 2004.

[8] I. Fette and A. Melnikov. The websocket protocol. 2011.

[9] Ghali, Cesar and Tsudik, Gene and Uzun, Ersin. Network-layer trust in named-data networking. *SIGCOMM Comput. Commun. Rev.*, 44(5):12–19, Oct. 2014.

[10] 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.

[11] 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.

[12] S. S. Lam. Protocol conversion. *IEEE Trans. Softw. Eng.*, 14(3):353–362, Mar. 1988.

[13] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. Named Data Networking. Technical report, April 2014.

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

[15] X. Qiao, G. Nan, W. Tan, L. Guo, J. Chen, W. Quan, and Y. Tu. Ccnxtomcat: An extended web server for content-centric networking. *Computer Networks*, 2014.

[16] W. Shang, J. Thompson, M. Cherkaoui, J. Burke, and L. Zhang. Ndn.js: a javascript client library for named data networking. In *Proceedings of IEEE INFOCOMM 2013 NOMEN Workshop*, 2013.

[17] I. Solis. CCN 1.0 (tutorial). In *ACM ICN 2014*, Sept. 2014.

[18] J. Wang, C. Peng, C. Li, E. Osterweil, R. Wakikawa, P.-c. Cheng, and L. Zhang. Implementing instant messaging using named data. In *Proceedings of the Sixth Asian Internet Engineering Conference*, AINTEC '10, pages 40–47, New York, NY, USA, 2010. ACM.

[19] S. Wang, J. Bi, J. Wu, X. Yang, and L. Fan. On adapting http protocol to content centric networking. In *Proceedings of the 7th International Conference on Future Internet Technologies*, CFI '12, pages 1–6, New York, NY, USA, 2012. ACM.