Interoperability of ICNs and IP

(Technical Report)

***Abstract*—Information-Centric Networks (ICN) enable access to content, services and objects based on identity, independent of location. The different ICN architectures, Named Data Networking (NDN), MobilityFirst (MF), etc., are likely to begin as islands of distinct networks, while they evolve and gain acceptance. Information currently predominantly accessible over IP infrastructures, will also begin residing in the other domains. Thus, there is a strong need to interoperate across these network architectures and access content residing in any of them. We present a framework for interoperability across three very different (NDN, MobilityFirst and IP) network architectures. Our framework uses Object Resolution Services to map end-user keyword-based search queries to object names across domains, and uses gateways that retain essential interoperability state. The framework and gateway design supports interoperability in a general way, allowing "adapter plugins" to accommodate different ICN architectures (including others beyond NDN and MF) or IP on each gateway interface. Additionally, we provide a formal approach for modeling and verifying essential properties such as reachability, returnability and no-conflict via exhaustive search in configurations based on our framework. Measurement results from an implementation of the gateways between IP, NDN and MF show that the added latency is small and manageable.**

I. Introduction

Users primarily seek information over the network without necessarily wanting to focus on its location or the underlying mechanisms used to retrieve that information. However, the current way of using "location-based" access in IP results in a less convenient and less efficient means for information dissemination and retrieval. Information-Centric Networks (ICNs) separate content identity from location. ICN enables access of content based on its name, from wherever it resides, supporting mobility as well as accessing the named content from the "best" source. It also allows for ubiquitous network wide caching to reduce access latency. There are a number of ICN architectures – Named Data Networking (NDN) [1], [2] and MobilityFirst (MF) [3], XIA [4], PURSUIT [5], to mention a few – we primaily focus on NDN and MF here.

NDN [2] (a follow-on to Content Centric Networking (CCN) [1]), uses hierarchical, human readable content names to access information. NDN is based on a request/response, pull-based data retrieval model for all applications. The user-readable hierarchical names in NDN capture relationships between related content items. Mobility in NDN is supported by clients re-issuing Interest (request) packets. NDN routers cache content, thus benefiting subsequent requests for the same content. MobilityFirst (MF) [3], on the other hand, uses 20-byte flat Globally Unique Identifiers (GUIDs) to identify each content, user, object or service. A comparison between the two naming schemas, *i.e.*, hierarchical and flat labels is provided in [6]. The fact that each object can have a single GUID that can be mapped to different Network Addresses (NAs) as the object moves makes MF suitable for mobility. MF also supports in-network caching on routers. MF uses a reliable transport and store-and-forward capability in routers to support mobile nodes, disconnected operation and poor quality links. NDN propagates content names from publishers to the entire network, thus populating the Forwarding Information Base (FIB) of NDN routers. Thus, the mapping from content names to the location of the content is done implicitly, "on path", when a request for the content name is routed along the shortest path to the nearest publisher. MF on the other hand has a logically centralized Global Name Resolution System (GNRS) to perform the content/object name to location (*i.e.*, network address) mapping.

The fundamental architectural differences between the Future Internet Architectures (FIA) NDN and MF, and with the current IP architecture, requires a framework for interoperability to allow producers and consumers to access information freely across these distinct architectures. Then, content can be anywhere, with the interoperability framework bridging these islands that have distinct architectures and creating an overall connected environment for seamless information sharing.

The differences between each of these architectures, however, makes interoperability challenging: they have different service interfaces, different routing policies (RPF in NDN, unidirectional shortest-path forwarding in IP and MF), different naming schemas (hierarchical name space in NDN and flat-ids in MF) and different semantics (destination in IP packets denote nodes, in MF packets it denotes either objects or content, and content names in NDN). Name-to-location resolution is different (on-path for NDN *vs.* off-path for MF with the GNRS or for IP with DNS) and different types of packets (dual interest/data packets in NDN, one type of packet in IP and MF) and packet formats (different header fields in each).

We seek to overcome these challenges for interoperability across these ICN architectures and IP in this paper. There are some key properties that our interoperability framework supports. Our foremost goal is an obvious one: not require the individual architectures (IP, NDN or MF) to change to accommodate interoperability. Routers and end-systems should operate as is, without change. The naming schema in each architecture should not have to change, and be allowed to evolve independently. As far as possible, all the services offered in each of the domains should be available to be used across domains. We also seek to avoid creating yet another canonical network layer that spans all the domains.

In this paper, we present an ICN interoperability framework and protocol that supports three main services across NDN, MF and IP architectures: 1) queries for dynamic content, exemplified by the task of object resolution to obtain names (search for a keyword and get a list of content names associated with it), 2) static content retrieval (getting the content

associated with a requested name) and 3) publish/subscribe (getting all future publications associated with a subscription keyword). The interoperability between domains with different architectures is achieved by having an appropriate translating gateway at the intersection of the domains. The gateway maintains state for each session, so as to map and demultiplex packet flows in each direction (match requests to responses). The protocols and mechanisms in each of the domains remains unchanged. We implement the gateway with a common core and having an "adapter" on its interface to be able to communicate with the architecture of the domain that it connects to on that interface. While the framework and protocols we design address interoperability between IP, NDN and MF, the approach allows for interoperability with other network architectures as well, as long as they have a set of information elements to support the common communication patterns. Then, an adapter can be built for the new domain's architecture. One aspect that is implicit with all the architectures is the assumption that the name of an object or content desired by the user/client application is known and is used for constructing a request. We incorporate an "Object Resolution Service" (ORS) [7] to provide client applications with the object/content name and the domain type it resides in, so that the client can incorporate the domain type in the content request.

To demonstrate that our framework is correct, we formally model the proposed specification and verify essential properties using the Alloy tool [8]. We also seek to have a gateway that is efficient and scalable. Our measurements on a testbed implementing the gateway between NDN, IP and MF, indicate that the overhead is reasonable.

The contributions of this paper are the following: 1) a generic framework among ICN domains for interoperability, including static/dynamic content retrieval and publish/subscribe; 2) an implementation of the framework which provides interoperability among IP (HTTP), NDN and MF; 3) a formal model for interoperability to verify properties in all possible configurations; and 4) measurements from an implementation of the framework across three different domains to show the performance of our approach.

II. Design Rationale

In this section, we address what we believe are the important design goals considering the communication patterns and services supported across the ICNs and how application layers have used the underlying IP communication fabric. We then examine a number of alternatives that have been proposed in the literature and discuss possible alternative interoperability solutions.

A. ICN Architectures

NDN [1], [2] is a network architecture made up of human-readable, hierarchical names for content, special packets Interest/Data that carry request/response for named content, and routers that are capable of forwarding and caching content. NDN uses Reverse Path Forwarding (RPF) to deliver requested content back to the client based on request state maintained in the network. For static content, *i.e.*, content that does not change frequently or is independent of time of request, router caches can respond, thus reducing the response time for content access.

MobilityFirst(MF [3]) uses flat Globally Unique Identifiers (GUIDs) to identify each content, user, device, *etc.* A key component of MF, is a distributed Global Name Resolution Service (GNRS [9], [10], [11]) for name-to-address resolution. GNRS keeps a mapping between GUIDs and Network Addresses (NAs). MF helps when nodes are mobile, since the GUIDs are fixed, despite frequently varying NAs. MF allows for late binding, so that packets are forwarded towards the destination, but the router close to the destination can perform another lookup to reach a mobile destination with a new NA.

B. Communication Patterns

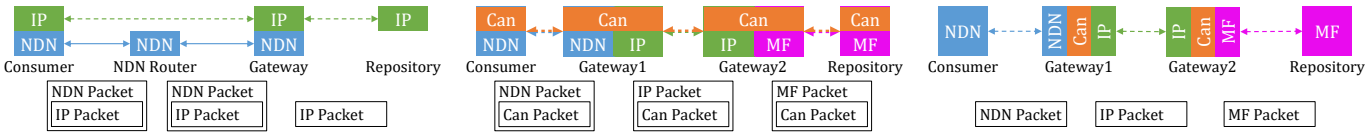
We recognize that there are a number of different services that are provided by the different ICN architectures and IP. The interoperability design needs to support all of these. Content retrieval is key. This is supported across all of the architectures. We differentiate between "static" and "dynamic" content. Static content does not change (at least not frequently) and is independent of the particular request instance. Examples of these are multimedia content or static web pages. Static content can be cached in the network. Dynamic content is such that the response data depends on the information in the request, as for example keywords in a search query. Dynamic content can based on a current server-generated response and be based on the current state of the information layer. Examples may be a query for the current weather or keyword search. Thus, the retrieved response should generally not be from a cache.

We also have additional service interfaces that are widely used for information access. *E.g.* publish/subscribe, where a user subscribes to an information item (or a group of items based on a topic hierarchy). The information may be generated at any point after the subscription is made (the usual case) or it may be generated even before the subscription (in which case respositories or information brokers of past publications have to be set up to respond).

The underlying communication capability in each domain supports both unicast and multicast. At the naming layer, content items may be assigned individual names. In addition, architectures such as NDN and IP support name relationships such as hierarchies to be used for identifying information aggregates.

C. Existing Interoperability Solutions

Interoperability between different network architectures has been studied in the past [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]. Interoperability approaches can generally be classified into tunneling/overlay, translation at the intersection between two domains, or conversion to a single canonical form (essentially introducing a new common layer). Many of the current approaches use *tunneling* to go across other



(a) Tunneling in other domains (overlay) (b) New thin-waist across all domains (c) Translation on the gateways

Fig. 1: Alternative solutions for interoperability across different domains.

domains, and typically consider the pairwise interoperability between domains (often between an ICN and an IP network carrying with HTTP/TCP). Some approaches propose the use of *a new layer* (serving as a new "thin waist") to enable the interoperability across multiple domains. The *translation* approaches so far have sought to translate between a specific ICN architecture and IP (HTTP) traffic.

1) Tunneling: To support interoperability between ICN and IP, tunneling solutions have been proposed. One protocol is an overlay and tunnels are created to traverse routers of another protocol (see Fig. 1a). They can be generally classified into NDN-overlay and IP-overlay methods. The basic design of NDN [1], [12] and its pub/sub extension [13], [14] adopt an NDN-overlay – NDN packets are encapsulated into UDP, TCP or native IP packets while going through IP routers. This design enables the incremental deployment of ICN over IP – starting from having key nodes (*e.g.*, end hosts, RPs) in the network support basic ICN functionality. Then, deploying more ICN-capable nodes provides better scalability.

In contrast to NDN-over-IP overlay solutions, IP-over-NDN overlay solutions (*e.g.*, VoCCN [15], TCP/ICN [16], IP-over-ICN [17]) allow legacy (TCP- or HTTP-based) applications function across an ICN infrastructure. Fig. 1a shows an example of an IP (or HTTP/TCP) network on top of NDN, where IP packets are encapsulated in NDN headers. These headers are decapsulated when leaving the NDN domain. VoCCN encapsulates SIP [22] and (S)RTP [23], [24] in the NDN Data packets as part of the overlay approach. Since vanilla NDN does not support a sender pushing content to the receiver, VoCCN makes the receiver initiate a request for each next possible segment (identified by "/prefix/*call-id*/rtp/*seq-no*"). To enable efficient data push (similar to TCP) in NDN, TCP/ICN adopts two NDN proxies (forward and reverse) in the process: the forward proxy caches the data (encapsulated TCP packet from the sender) to be pushed and sends an Interest (with TCP/IP headers as notification) to the reverse proxy; the reverse proxy then requests the cached data, decapsulates it and forwards the TCP packet to the receiver. Each data chunk is identified by "/prefix/*conn-id*/*seq #*".

While these solutions made significant contributions in the early stage of ICN (enabling rapid deployment of ICN, and supporting legacy functionality in ICN), there are still several challenges while designing a general platform for interoperability among all these domains. The first is that the provider and consumer have to be in the same domain (the overlay domain). This prevents an NDN client from requesting content provided by an IP repository unless the IP repository is willing to be an NDN node. While such an approach may be feasible when we only consider two architectures such as NDN and IP, it can be difficult when there are more domains like MF, XIA [4], PURSUIT [5], NetInf [25], *etc.* To ensure the global reachability, it would require *all* the consumers (or providers) to have access to services across *all* domains. This needs to be done in an efficient manner. Further, it becomes increasingly inflexible as we add new domains.

Secondly, tunneling cannot take advantage of the capabilities (*e.g.*, caching) in the underlying domain since the underlay usually does not understand the semantics of the overlay. While such solutions are acceptable for content that will only be consumed once (*e.g.*, VoCCN), they are inefficient when delivering popular content that is repeatedly accessed.Therefore, we believe there is a need to rethink interoperability – especially taking advantage of the capabilities (*e.g.*, caching) in all of the domains. We also need to recognize the need to treat *both* static and dynamic content as separate classes.

2) A common (canonical) name layer: We could provide the interoperability among all the ICN domains and IP using another unified/canonical name layer (*e.g.*, "Can" layer in Fig. 1b). This layer provides the service interface for the common communication patterns, §II-B. To send a canonical packet ("Can packet" in the Fig.), the end-hosts need to create an equivalent domain-specific encapsulation (NDN header on the consumer side and MF header on the repository side). At the border of each domain, the gateways (Gateway1 and Gateway2) need to recreate the domain-specific encapsulation similar to the end-hosts (*e.g.*, IP header through a set of IP routers).

The benefit of the solution is that it can take advantage of the domain capabilities since the canonical packet can be translated to something the domain nodes understand. *E.g.*, a content with canonical name "ICDCS18.Interoperability.pdf" can be mapped to NDN chunk names "ccnx://ICDCS18/Interoperability.pdf/20170508/_seg", HTTP URL "http://ICDCS18/Inter-operability.pdf", and MF GUID "FC 27…9483". Each domain can then leverage either the in-network cache (in NDN and MF) or the application-layer cache (CDN in HTTP/IP) to improve performance in content retrieval. Moreover, the end-hosts do not need to worry about supporting multiple domains as is required in tunneling solutions, since all the end-hosts communicate over the canonical layer.

While this solution addresses the issues with the tunneling solutions, it also poses several challenges. The most critical is that it requires another modification on the end-host logic and

gateway functionality, in addition to supporting the particular ICN.This would make the deployment of the new canonical layer to be even more difficult. Another concern with the canonical layer solution is the namespace size in each of the domains. Since each content, no matter where it is served, requires a name in the canonical domain, the namespace size in the canonical domain is the union of the total number of contents in all the domains. This makes it difficult for the domains/gateways to maintain the mapping between canonical names and domain-specific names (*e.g.*, MF GUIDs).

3) Translation:: A number of existing interoperability solutions [20], [18], [19] perform direct translation between HTTP and NDN/MF traffic. Work in [20] further optimizes the ability to cache in the network by adding heuristic rules (*e.g.*, treating two similar URLs as being same content item based on the content format of the HTTP server). Moiseenko *et al.* [21] also seek for solutions to modify NDN packets to better support HTTP-like communications (*e.g.*, uploading a large piece of data using POST).

While these solutions enjoy the benefits of interoperability without the penalty of creating another layer, there still exists several concerns. First, these solutions use a one-to-one mapping between NDN names (or GUIDs) and HTTP URLs, or vice versa. This could enlarge the namespace just like the canonical layer solution. Second, these proposals only consider pairwise translations, and primarily focused on content request/response, lacking a more general support for the range of ICN communication patterns. We believe it will be difficult for these solutions to add support for other ICN domains, or even non-ICN domains that support generic communication patterns (*e.g.*, P2P, FTP).

Beyond the challenges in providing functionality, it is important to consider the amount of state needed in each solution, as this will affect scalability. In the tunneling solution, the state of the tunnels are kept at the routers (especially at entry/exit) in the the underlying domain. In the translation solution, the gateways need to maintain more comprehensive state to keep the context of the flow on each side meaningful to the domain, and potentially added context to match them up. While we acknowledge that the amount of state is a challenge for translation-oriented solutions, we still believe this solution can better support the range of capabilities desired (services, rich name space, in-network caching). Moreover, we make the observation that maintaining a stateful forwarding plane as in NDN routers and HTTP proxies has become a common practice demonstrating its benefit in enhancing content retrieval and distribution as well as supporting mobility.

D. Summary of Requirements

Our interoperability framework seeks to achieve a set of important design goals to achieve a robust, efficient and deployable framework.

- It should add minimal or no architecture or protocol change to the individual domains.
- Be seamless to clients: a client in one domain uses the native mechanisms of the domain to exchange information with an entity in another domain.
- Support both static (*e.g.*, a movie) and dynamic content (*e.g.*, query for current weather information).
- Support in-network caching whenever the domain supports it.
- We do not want each domain's content name space to include all objects in the whole world; *i.e.*, no equivalent names need to be necessary in every domain. Rather, we wish the name space size to be limited to what the domain has.

E. Overview of Our Solution

The interoperability solution framework we adopt is to have gateways perform a minimal translation of requests and responses across domains. Our framework ensures that there are no changes to the clients or servers on each of the domains and the routers/forwarding engines within each domain are not altered. The framework itself enables the gateway implementation to be designed to be a "universal" one with the appropriate adapters for each of the interfaces that connect to different domains (see Fig. 1c). The gateway translates requests to an internal canonical form to provide extensibility and flexibility to support interoperability across a number of different ICN architectures and with IP. Our approach allows for interoperability that can span multiple intervening domains between a requesting client and a destination content repository.

Our approach does not seek to perform a one-to-one mapping of names from one domain to another. Rather, the client uses a form that is native to the client's domain by specifying a remote domain's name/identity as a domain prefix and an opaque string (as far as the client or intervening domains are concerned) that is the ultimate destination domain name. This allows the query to be in the client domain's native format. Similarly, the response is in the repository domain's native format. We enable caching in the domain's routers based directly on the opaque "name". Ensuring the delivery of the most current version of static content is by translating between HTTP's "if-modified-since" values and the "exclude" field in NDN. The interoperability gateway explicitly differentiates access to static content *vs.* dynamic content by examining the HTTP method (for IP and MF) or version number (NDN). This also ensures that routers in an individual domain (*e.g.*, NDN) do not cache dynamic data by having the client exploit the version number to create a different name for requests for dynamic data. By extending NDN to support pub/sub with an approach like [13] and treating subscriptions as "standing" queries (not consumed by a corresponding response), the interoperability framework also supports the varied service interfaces for information delivery (query/response and pub/sub).

The key to our approach is to have a content/object name provided in a form that enables the client to construct a query in its native form without having to translate it. The destination domain is also provided to the client as part of the name, if the

content resides in a remote domain. This enables routing of the request appropriately towards the destination domain. The name is provided to the client either by out-of-band means (*e.g.*, a user typing in the name), or, more likely by an object resolution server (ORS) [7]. The client tells the ORS the domain it is residing in (similar to the user agent field in HTTP with the web browser type, OS type). The ORS forms the response in a format that can be used by the client.

translates Alloy expressions to conjunctive normal form statements and applies satisfiability algorithms to them

III. Architectural Design

In this section, we present the high-level architectural view of the interoperability framework and its essential components, explain the details of the protocol exchange for various services and scenarios, and describe the gateway implementation.

A. Common Information Elements

We first list the set of common information elements shared (and needed) by all the domains. We use HTTP, NDN and MF as examples but the architecture supports all protocols that have these common elements (*e.g.*, XIA, NetInf, or even FTP).

Request type: In our current gateway implementation, we treat HTTP with "method=GET" as a request for static data (assuming the content is RESTful [26]) while POST is interpreted as a request for dynamic data since the client can provide extra information (*e.g.*, key words) to get different results even when using the same server/content name. We can do the same thing for MF (since it also uses HTTP at the application layer). HTTP CONNECTION is interpreted as a subscription, overloading this method due to the lack of pub/sub support in HTTP. In NDN, we observe that while querying for static content, common practice is to query for a name prefix without a version and segmen id. A version (current time) is added in the query for dynamic content, to avoid routers caching the response. Our implementation treats requests with a version as dynamic content requests; otherwise as static content requests. COPSS subscription packets are treated as subscription messages.

Destination domain & content name: To avoid an equivalent name in each domain, we choose to use "DstDomain/ContentName" to identify content in queries. Note that this is not "another name" in the consumer (or intervening) domains as the name itself does not create another entry for resolution (*e.g.*, DNS in IP, FIB in NDN and GNRS in MF) therefore it is scalable. We only assume that the ContentName is understandable by the destination domain while the other domains (including the consumer) just see it as a binary string. We need a resolution entry for each destination domain which points to the proper gateway. *E.g.*, in DNS, "DOM_NDN"$\mapsto$IP$_{GW}$, in MF, GUID$_{DOM_IP}\mapsto$NA$_{GW}$, and in NDN FIB of "/DOM_MF" points to the interface towards the gateway.

Content version: We allow several versions under the same name (prefix), similar to NDN. Therefore, each response should have a version element. It is equivalent to the "Last-modified" field in HTTP header, and the version component in NDN names.

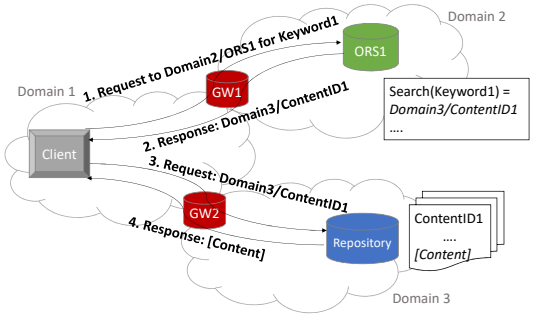


Fig. 2: Protocol exchange to obtain the name and retrieve the content.

Exclude for static content request: Consumers can get content from caches. In order to get the latest version of a content, the consumer can specify "exclude the versions earlier than the one I already have". This prevents redundant downloads of the same content. It is equivalent to the "If-modified-since" field in HTTP and the exclude field in NDN Interest.

Input for dynamic data request: To allow a consumer to pass parameters to a dynamic data provider, we include the element"input". It can take the POST body of HTTP (and can include header fields like Cookie, browser type, *etc.*). In NDN, since the Interest usually does not contain a body, we encode the input field into the name in the Interest. We can also use the solution described in [21] to provide efficient large parameter (*e.g.*, file) uploading.

Demultiplexing key is used to identify a corresponding request when the data comes back onto a gateway. For static data, we use $< DstDomain, ContentName, Exclude >$ tuple as it can uniquely identify a content response. We treat two requests with the same demultiplexing key as the same request. They can be aggregated at the gateway, similar to PIT in NDN. For dynamic data, we need a unique ID for each request. It can be consumer $< IP : port >$ in HTTP, consumer $< GUID, ReqID >$ in MF, and a cryptographic hash of the request in NDN.

B. Interoperability Architectural Components

The network environment we consider can have a number of interconnected domains (IP, NDN and MF) and gateways that connect these domains. Clients, publishers/repositories can reside in any one of the domains. A gateway could have multiple "adapters", each for an interface to a domain of a particular ICN architecture the gateway is connected to.

Naming: We envisage an architecture that supports increasingly complex and flexible structures for identifying and naming information and objects, allowing for multiple naming schemas, including hierarchies as well as graph structures (that have more complex linkages between entities, allowing loops).

Object Resolution Servers: When a client seeks a content in another domain, it requires the content ID (*i.e.*, name, URL

or GUID) for it in the other domain. To obtain that foreign content ID in a format understandable by the client as well as the remote server/repository we propose the use of one or more application-layer search-engine like entities called Object Resolution Servers (ORS) [7]. The primary role of the ORS is to provide the name of an object or content that a client is searching for based on keywords. ORSs may reside in one or more of the domains we consider in our environment. Each content ID is sent back to the client together with an identifier of the domain type that the content belongs to.

A schematic of a protocol exchange to obtain the name from the ORS and retrieve the content is shown in Fig. 2.

Service Interface: The interoperability framework supports a number of functions at the service interface, across multiple domains: query/response for both static and dynamic contents, and publish/ subscribe. We assume IP and MF domains have multicast mechanisms on top of which a pub/sub service can be deployed. For NDN, we assume the network is equipped with a pub/sub framework like COPSS [13]. COPSS utilizes an additional Subscription Table (ST) for subscribers on a "downstream" interface. In MF multicast [27], the group topology is maintained inside the GNRS and multicast GUIDs need to be mapped to actual node GUIDs along the path .

Interoperation Gateways: To be able to go across domains (inter-domain routing), we use gateways to interface between each pair of domains. A gateway that connects two distinct domains understands both protocols. As some of the necessary information required for interoperability may be part of the application payload in one domain, rather than just the network layer header, a gateway can processing data appropriately all the way up to the application layer header. This makes these gateways a mixture of application-layer proxies and network-layer routers/forwarders. Also, gateways retain state in order to match a response with its associated request.

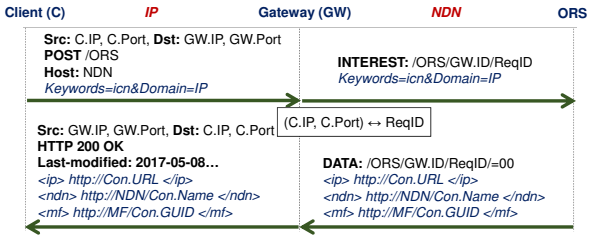
Routing: For simplicity, we assume single logical gateway between each pair of domains. Gateways connecting the same two domains may exchange state to achieve a single logical gateway. Reverse Path Forwarding (RPF) is a key policy in NDN and our interoperability framework reflects that. When there are multiple gateways between an NDN domain and another domain, the response needs to be sent to NDN domain to the same gateway as the one that received the Interest (*i.e.*, entry and exit gateways for NDN must be the same). MF and IP do not have such a constraint.

C. Protocol Exchange in Interoperation

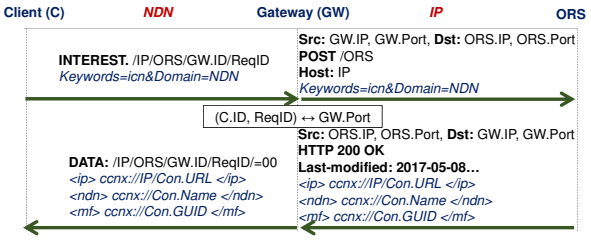
We now describe the protocols for interoperability for three services, namely, dynamic content retrieval (DCR), static content retrieval (SCR) and publish/subscribe (Pub/Sub) across multiple domains.

1) Dynamic Content Retrieval:

The response of a dynamic content might depend not only on the input from the consumer, but also on the current state of the server (*e.g.*, time or a random number on the server). Therefore, the retrieved response cannot be from a cache as the current server-generated response is desired. This requires



(a) IP-to-NDN



(b) NDN-to-IP

Fig. 3: Dynamic content retrieval (object resolution) between IP and NDN.

the requests to be distinguishable (globally unique), to have the correct response-to-request mapping on the servers and gateways, including those made by the same client. In TCP/IP, client IP and port numbers provide this demux capability. For NDN and MF, we introduce the use of unique Request ID (ReqID) generated by the consumer or the gateway. ReqID can be a component of the DCR Interest name in NDN and part of the request payload in MF. To ensure the global uniqueness of ReqID in NDN and MF, we can either use cryptographic hash or combine it with a unique client ID (*e.g.*, GUID in MF). The gateways need to maintain a state for each request to subsequently associate the response. In our architecture, we maintain a mapping between the demultiplexing entity of the incoming request (from the consumer) and the demultiplexing entity of the outgoing request (from the gateway). *E.g.*, in Fig. 3a, the mapping on an IP-to-NDN gateway is a 3 tuple of $< ClientIP, ClientPort, ReqID >$. When the Data comes back from NDN, the gateway can find the corresponding request based on ReqID in the ContentName.

Fig. 3a shows an example of dynamic (ORS) content retrieval where the consumer C is in IP and the server ORS is in NDN. The two domains are connected via gateway GW. C forms an HTTP POST request including the destination domain and ORS name in the URL. His own IP and port are placed in the IP and TCP headers respectively. The GW's IP will be placed into the IP header via a DNS lookup for domain name "NDN". The name of ORS is an opaque name that the client does not have to understand. C places keywords and his own domain type inside payload of the POST request. It is important that all HTTP requests carry the "Host" field so the destination domain name (in this case NDN) does not get lost when traversing multiple domains. In all of the figures in this section, only the relevant parts of packet headers are shown. Information carried in payloads is italicized, and in

blue.

Once GW receives the request, it processes the packet and generates an Interest using its own ID, a new request ID (as version), ORS as a prefix and sends it with the request input towards the ORS. The input can be encoded into the Content-Name or placed in the packet payload [21]. GW also retains state for the outstanding request $< clientIP, Port, GW - generatedReqID >$, shown as a box below in the Fig.

After receiving the new request from GW, ORS would look into its index and find candidate names based on the keywords. For illustration, we consider the general case that for the keyword, there is a matching content ID in each of the three domains, and the example response has a name returned for each domain. Note that based on different consumer domain (parameter in the input "Domain=IP/NDN"), ORS would form different name formats that are understandable by the clients (difference in response between Fig. 3a and 3b). Generally, a response can contain arbitrary number of candidate names; and possibly include descriptive snippets.

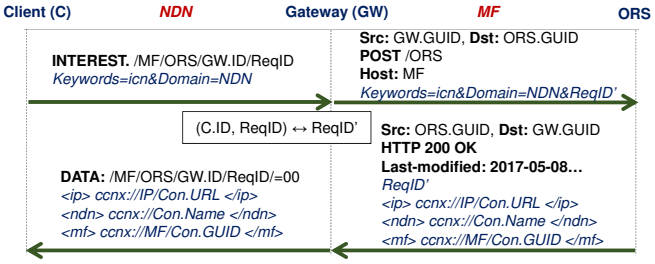
Upon receiving the response, GW locates the ReqID in the name and perform a lookup in the state table. It can find the IP and port of C and send the response in the form of HTTP response. Fig. 3b shows the reverse scenario, *i.e.*, an NDN client querying an IP-resident ORS. Here, the gateway's job is to convert an Interest packet to an HTTP request format and keep the corresponding essential information as state for the request (very much like flipping the left and right side in Fig. 3a).

Dynamic content request (and object resolution) in NDN/MF (Fig. 4) and IP/MF (Fig. 5) domains follow a similar pattern as IP/NDN except that on the MF side, we have GUIDs as source and destination. For a cross-domain request, MF uses $\text{GUID}_{\text{DstDomain}}$ as dstGUID and it will be mapped to NA_{GW} according to GNRS. Since there is no default transport layer in MF for demultiplexing, we need the reqID carried in the response for detail).

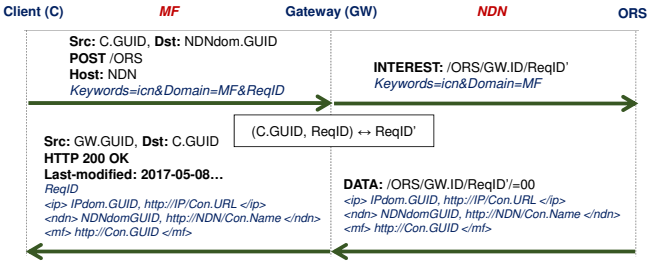
Our interoperability framework also supports going across any number of domains, as seen in Fig. 6 which is the case of an IP client wishing to query an NDN-resident ORS while there is an intervening MF domain in the path. Gateways GW1 and GW2 store the required state, which is limited to the request and response between the two domains they interface with; *i.e.*, in figure 6, GW2 is relaying a request from GW1 to ORS and only retains state associated with the request on the MF and NDN sides. The client domain (IP) and ORS domain (NDN) are kept unchanged in the packet over the entire path.

2) Static Content Retrieval:

Once the client has acquired the content ID from the ORS, content retrieval service can be initiated to request for a piece of content. Here, we focus on static content, *i.e.*, content that can be cached and retrieved from router content stores or CDNs. In this case, there is no need for a globally unique ReqID since the response can be uniquely matched by the content name and the exclude. Therefore, the gateway keeps state based on the content name and exclude.

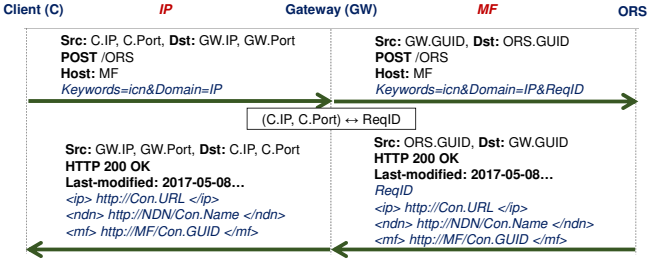


(a) NDN-to-MF

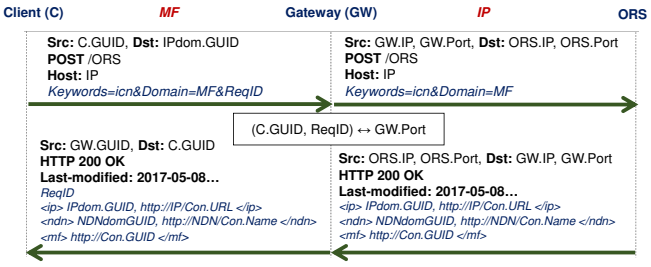


(b) MF-to-NDN

Fig. 4: Dynamic content retrieval (object resolution) between NDN and MF.



(a) IP-to-MF



(b) MF-to-IP

Fig. 5: Dynamic content retrieval (object resolution) between IP and MF.

Fig. 7a shows NDN-to-MF static content retrieval scenarios. Here, an MF-residing content named "Con.GUID" is treated as just another NDN hierarchical name in the form of "/MF/-Con.GUID" in the interest/data packets. Conversely, an NDN-residing content named "Con.Name is" used inside an MF HTTP GET request packet as part of the URL. Note that the gateway is performing the translation between NDN exclude field and the "If-modified-since" field in HTTP. On receiving the request, the repository can respond with "200 OK" when there is a new version, or otherwise "304 Not Modified"

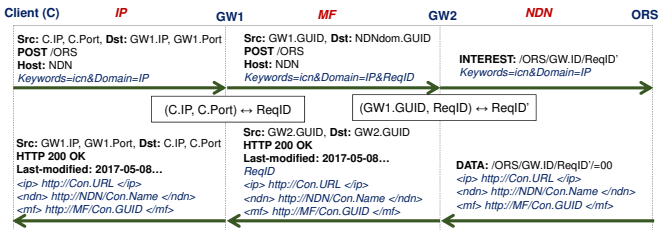
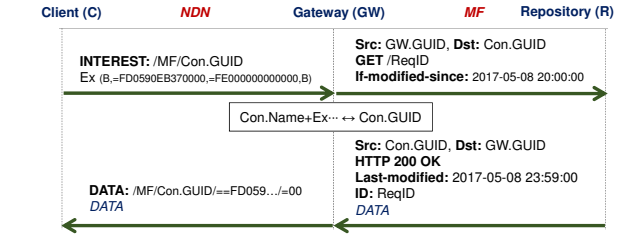
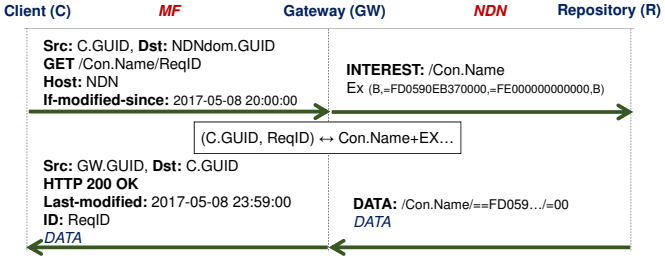


Fig. 6: Dynamic content retrieval across 3 domains.

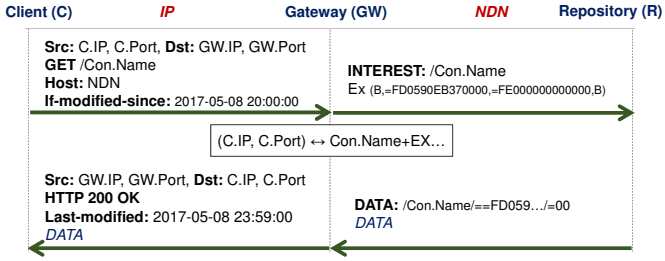


(a) NDN-to-MF

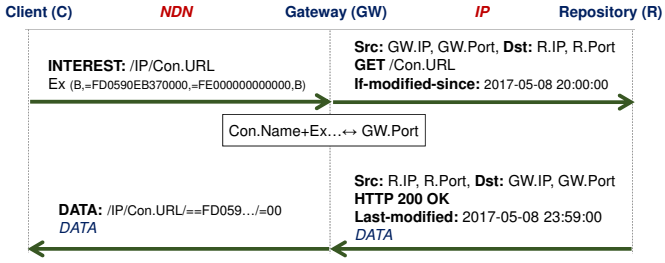


(b) MF-to-NDN

Fig. 7: Static content retrieval between NDN and MF.

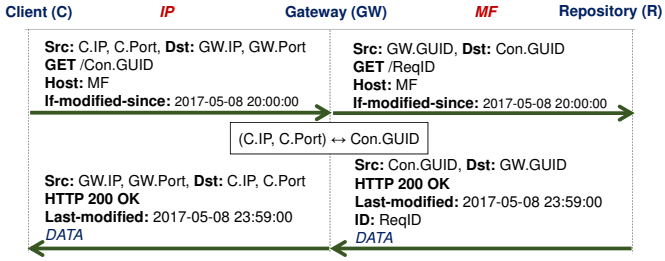


(a) IP-to-NDN

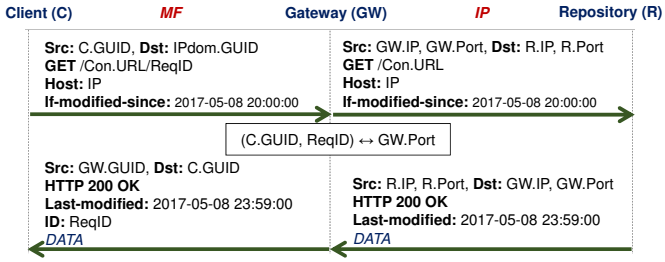


(b) NDN-to-IP

Fig. 8: Static content retrieval between IP and NDN.



(a) IP-to-MF



(b) MF-to-IP

Fig. 9: Static content retrieval between IP and MF.

without a response body. NDN repository can simply discard the request and the consumer will wait until a timeout (just as the normal NDN file get logic). Fig. 7b shows the reverse scenario, *i.e.* one where a MF-residing client requests static content from an NDN repository. Other scenarios, namely IP/NDN and IP/MF also follow similar patterns and are displayed in Fig. 8 and 9 respectively.

Fig. 10 shows a scenario for three domains which is a case of IP/MF/NDN. We have gateways that are only aware of the two domains they are interfacing with, so as long as the state identifies each request/response mapping without conflict, it can be used to return the response data in the form the requesting client can use.

When the content traverses through the NDN and IP domains, they can be cached in content stores or CDNs identified by their unique name (no matter if the domain can understand the opaque name) and version. However, when MF is not the target domain, content traveling through MF will not be cached at routers (but it can still be cached at application-level CDNs). This is due to the fact that we use the content ID as opaque strings within the HTTP header, and the destination GUID is a node GUID. Thus, the non-existence of a content GUID for a non-MF residing content precludes MF routers from caching that content. One way to overcome this would be to register a new content GUID for every new content entering MF, but this may be expensive in terms of number of lookups. We allow different MF domains to have different choices based on the policy of the ISP.

3) Publish/Subscribe:

A Publish/Subscribe service enables clients to subscribe to a content name/prefix. We assume every domain supports a form of group communication. IP multicast, MF multicast, and COPSS in NDN, enable "pushing" published information to subscribers.

Pub/sub service with a publisher on the NDN side is of interest; subscriptions can be for a hierarchy of objects,

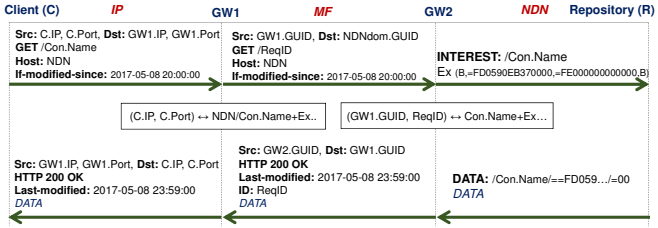
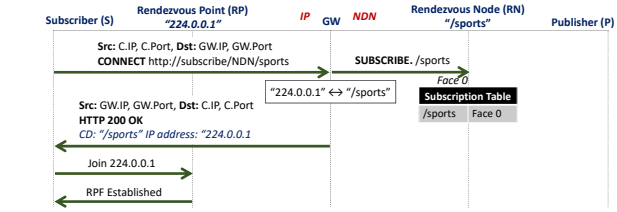
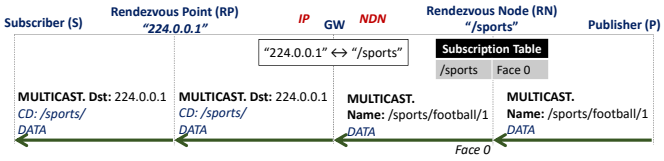


Fig. 10: Static content retrieval across 3 domains.



(a) Subscribe



(b) Publish

Fig. 11: Pub/sub between IP and NDN.

leveraging aggregation. To illustrate the protocol exchange, we use a scenario with a subscriber S in IP and a publisher P in NDN (Fig. 11). S subscribes to "/sports" and wishes to receive everything below it in a topic based hierarchy (*e.g.*, including content "/sports/football/1"). Here, the gateway GW looks like a publisher to S via an IP multicast group and a subscriber of "/sports" on the NDN side. GW maintains a mapping between subscribed name and the IP multicast address.

In Fig. 11a, S sends a subscription request to GW for an NDN prefix "/sports". Assuming the NDN side supports COPSS, GW subscribes to "/sports" and the interface associated with it will be added to the subscription table at the rendezvous node (RN) associated with "/sports" (and downstream of that RN). GW assigns an IP multicast group address, (*e.g.*, "224.0.0.1" in Fig. 11), to "/sports", retains the mapping and then sends this IP address back to the interested subscriber S. S joins the associated multicast group via a rendezvous point (assuming PIM-SM).

Fig. 11b shows what happens when a new publication under "/sports", viz., content named "/sports/football/1" is generated by P. RN forwards this to its subscriber GW. Using the prefix-to-IP address map, GW sends the new content to the multicast group 224.0.0.1, reaching S via the RP in the IP domain.

If a new subscriber T in the IP side subscribes to a different prefix "/sports/football", it will be assigned a new IP multicast group. However, this time since everything under "/sports/football" is also under "/sports", GW does not need to subscribe again on NDN side.

Now if a publication under "/sports/basketball/" arrives from the NDN side, GW needs to receive and send it to multicast groups associated with the name prefix. As for which multicast groups to send it to, "longest prefix match" in the prefix map applies. Thus, it is possible that the publication data needs to be sent by GW to multiple multicast groups. The Pub/Sub service in our framework also supports multiple domains. In addition to IP, MF multicast is very similar except for the only technical difference of using a multicast GUID instead of an IP multicast address.

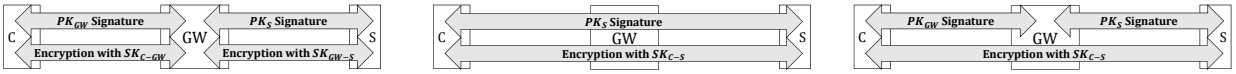
D. Security

Security is a major concern, when accessing online content. HTTPS over SSL/TLS is fast becoming the default over the Internet, to secure the channel. However, ICN secures objects by self-certification rather than securing the communication channel. Because of this conceptual difference in security, it is challenging to integrate security across different domains. While we are unable to describe the details of the security mechanisms due to space limitations, we outline three major possible patterns for secure interoperation, and justify the preferred pattern we adopt. We consider requirements of authentication, provenance, integrity and confidentiality. We assume Client (C) and Provider (P) are in two separate domains connected by Gateway (GW).

Pattern 1 (GW fully trusted): In this pattern (Fig. 12a), public key/certificate exchange/signature generation and verification are all done within a domain; *e.g.*, an IP client considers a message authentic if it is signed by GW after an SSL/TLS session setup. For encryption, symmetric session keys are exchanged within each domain, the GW being one end. This pattern requires no architectural change and no additional knowledge that spans more than one domain (*e.g.*, about cipher algorithms). Messages crossing domains require the GW to re-sign and performs a decrypt-then-encrypt of the message. If both domains are secure, an un-signed and/or un-encrypted message will not be carried on any channel, so there will be no chance for a network eavesdropper to forge/intercept the information. The provider and the consumer are not mandated to use the same kind of security mechanism. However, a major challenge remains: GW sees unencrypted content fully.

Pattern 2 (GW untrusted): In this pattern (Fig. 12b), C reveals nothing to GW, even what content is requested. It is equivalent to establishing and delivering encrypted traffic within a tunnel, in this case spanning different domains. Thus, none of the domain-provided security mechanisms will be used. The problem here is that it requires common application-layer protocols to help client/provider pairs to understand, establish and agree upon key exchange, encryption and verification and the supported algorithms, a major burden and requiring changes to end-system capabilities. This will also not use any of the rich content-based security mechanisms in each ICN domain, depending only on channel-based security.

Pattern 3 (GW minimally trusted): In this pattern (Fig. 12c) C trusts GW's signature and validation similar to pattern 1. However, for encryption, if secret keys are shared and agreed



(a) Pattern 1 (b) Pattern 2 (c) Pattern 3

Fig. 12: Possible security patterns for interoperation (pattern 3 preferred and adopted).

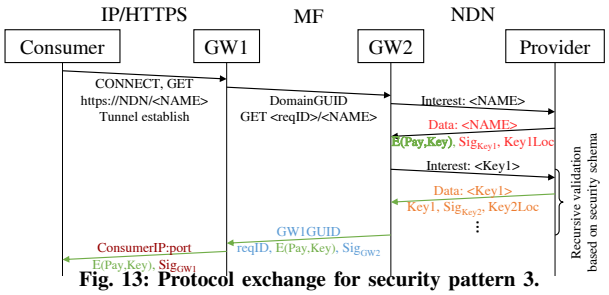


Fig. 13: Protocol exchange for security pattern 3.

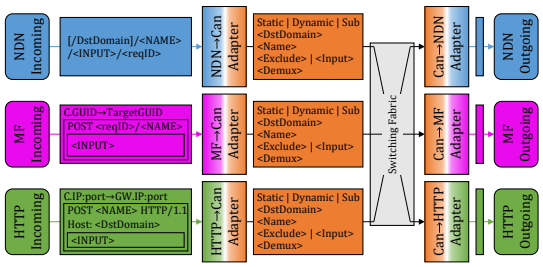


Fig. 14: Implementation: translating dynamic requests on the gateway.

by both ends (C and P), GW will not see or decrypt the content. This pattern does not require complete trust of GW, but requires a mechanism for sharing keys between C and P (*e.g.*, via an out-of-band or another in-band method).

We select pattern 3 for our interoperability framework, as it provides the appropriate trade-off between security and the cost of architectural change as well as end-user complexity.The reason for that is that it allows for domain-specific solutions for data integrity, provenance and authentication (signature) and relies on an agreement between consumer and provider for confidentiality (encryption). Thus, by fully leveraging domain-specific security mechanisms for authentication-related requirements and not revealing any sensitive information to a third-party, pattern 3 gets the best of both worlds.

We describe our security mechanism, *i.e.*, pattern 3, in more detail, with an example, in Fig. 13. A client in the IP domain retrieves content from an NDN repository, going through an intervening MF domain. The initial connection between the client and GW1 is via HTTP CONNECT using the URL associated with the content name; thus, GW1 sees the requested content name but nothing else, including subsequent data in the message payload. On the path of the content response, GW2 (Fig. 13) validates the NDN signature (by retrieving key chains, trust schema according to Certificate Authorities and cipher algorithms), re-signs it and hands to GW1. GW1 does something similar. Thus, only pair-wise trust within a domain is required, without the need for any entity to know about algorithms, schemas, certificates, *etc.*, used in a foreign domain.

E. Gateway Implementation

We implement the gateway that interconnects different domains of ICN as well as IP as shown in Fig. 14. The gateway translates requests for information received from one domain to a request meaningful in the adjacent domain and a similar translation of response headers in the reverse direction. We implement the interface to each distinct domain as a "pluggable adapter" on the gateway in each direction. We choose to translate the incoming request or the headers of the response to an internal canonical form. This enables the node to be a "universal gateway", that is able to interface to any of the ICN domains or IP with the corresponding adapter. This canonical form for query/response is internal to the gateway and does not appear on the links. We have our gateway implementation open sourced on Github [28]. Additional domains can be supported with a corresponding adapter on the interface to that domain.

Incoming request processing involves recognizing whether the request is for static or dynamic content. When converting a request or subscription to the canonical form, the adapter determines if it is for dynamic content (POST) or static content (GET) for IP and MF. For NDN, a request with a specific version is seen as a request for dynamic content while a request with just a prefix (and exclude) is for static content. It also determines the destination domain based on the "Host" field in case of HTTP, the destination GUID in MF, and the domain prefix in NDN. If the values of these fields cannot be parsed as a domain prefix (*e.g.*, google.com in the originating HTTP request), the destination domain is the same as the incoming domain (*e.g.*, in an IP-NDN-IP scenario). The opaque string (from the originating domain's perspective) that is the name on the destination domain will be extracted from the request (marked as the field"$< NAME >$" in Fig.14. For dynamic requests, the incoming request processing recognizes the body of the POST in MF and HTTP, and the penultimate component of NDN name, as the request input.The demultiplexing entity ("$< Demux >$") depends on the different cases. For static content requests, we use the tuple $< domainname, contentname, exclude >$. For dynamic content requests, we use client $< IP, port >$ (socket) for HTTP case, client $< GUID, reqID >$ in MF case and $< reqID >$ in NDN case.

The incoming request processing results in an internal canonical request (orange boxes in the middle column). The gateway can respond to requests for static content from the local cache, aggregate requests for the same static content (with same exclude) or consume them. The remaining requests (in canonical form) are sent to the "switching fabric", where inter-domain routing is determined and forwarded to the proper outgoing request processor.

The outgoing request processing forms a domain-specific outgoing request.One exception is that when the outgoing domain is the destination domain, a native request is formed

(no domain prefix in NDN; content GUID is used directly as destination GUID in MF, and native HTTP request is formed based on the "$< NAME >$").

In the current implementation, we encode the request body (name and input) as part of the Interest name in NDN. This implementation can be optimized using the solution in [16] when the body of the request is large (*e.g.*, client uploading an image). To enable such an optimization, we only need to modify the NDN adapter, and leave the other adapters unaltered.

When the response (*e.g.* NDN Data packet) returns, the gateway matches it based on the demultiplexing key and forward the content to all the pending requests on this key (similar to matching a PIT entry in NDN). This enables native multicast, similar to NDN. We use the "Last-modified" field in HTTP and MF and the version field in NDN Data name as the version of the response. The gateway sends the version using the domain-specific format.

IV. Formal Analysis

A formal analysis of interoperability addressing design choices and requirements is used to demonstrate correctness of our proposed framework. Using the formal model together with an automated verification tool, we generate all possible network configurations to understand the system and make sure of its logical consistency. We prove the essential properties of our interoperability framework protocol exchange, *i.e.*, check that a client is able to remotely search for a keyword and receive the content ID as a result (dynamic content retrieval); it can query for a content ID and retrieve the content (static content retrieval); subscribe to a content descriptor and receive all possible future relevant publications (pub/sub). We investigate additional properties such as conforming to NDN reverse path forwarding policy, absence of conflict (mismatch) between request and response by correctly distinguishing dynamic data, and receiving all publications of a subscribed category at different levels in the topic hierarchy with NDN/COPSS publish/subscribe.

We employ a model finding method with three main steps: 1) describing the model, through formalizing the mechanism in a model to represent all essential entities, and the relationships and interactions between them; 2) specifying required properties; through formalizing the major requirements of the framework so that it would capture what is expected of the model and 3) verifying the properties against the model, through automatically looking up each property among all generated instances of the model. The model description focuses on what each piece of the protocol delivers, *i.e.*, what information is carried in packets and stored at the gateway.

We use the modeling language Alloy [8] as our description/specification language and the Alloy Analyzer as our automated verification tool. Alloy has been used for various applications, including in [29] to model interoperation between networks such as PSTN and SIP, and prove connectivity properties. Alloy is a language based on first-order logic and relations. Its solver tool, The Alloy Analyzer, translates the high-level Alloy first-order logic expressions into conjunctive normal form (CNF) statements, applies satisfiability (SAT) algorithms to those CNFs, and checks if any predicate can be sometimes true, *i.e.*, an instance exists, and if an assertion is always true, *i.e.*, no counterexamples exist. Properties are specified as assertions, *i.e.*, invariants of the system, and their verification is successful only if no counterexample for them is found among all possible instances of the model.

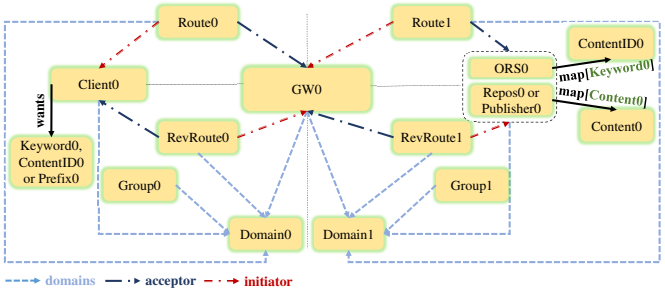


Fig. 15: General formal model for interoperability framework.

Since here we are mainly interested in representing how different interoperability components are related, *i.e.*, "which domain a piece of content resides in", "what client ID and demux values are associated with a request", "what names are understandable to a client/provider", *etc.*, a mathematical relation-based language such as Alloy is a good choice to represent n-ary relations. In this section, we present formal models for each of the three services, *i.e.*, object resolution (OR) as an example of DCR, SCR and Pub/Sub.

A. Connectivity and Availability

The model supports any number of domains, clients, servers and gateways. A high-level schematic example model of a supported 2-domain configuration is shown in Fig. 15, with one client, one object resolution server, repository and publisher, and one gateway connecting each pair of domains. The client in this scenario wishes to look up a (set of) keyword(s), request to retrieve content or subscribe to a prefix. The query packet containing the query gets routed across domains through routes, and delivered to the remote target. The response (content ID/content) gets routed back to the client through reverse routes. Objects of type "route" (and "reverse route") couple the notion of "a series of links" and "packets carried over them". Fig. 15 is only a part of the model, and each box denotes an object of a primitive (atom) type in our Alloy model. Each arrow denotes a relationship between entities. We also add a number of additional constraints (Alloy facts) such as node ID uniqueness, absence of routes with same initiator and acceptor nodes, and make sure for every NDN reverse route there is a (forward) route where the two are interest/data pairs for the same name (reflect NDN RPF policy). Any counterexample found in a scenario implies that client cannot generate a request native to its domain; server/publisher receives a request he cannot understand; gateway does not know what to do with a returned response; *etc.*.

1) Dynamic Content Retrieval (Object Resolution as example): In this subsection, we present the formal model for the object resolution service. The model supports multiple domains, clients, servers and gateways. An example model of a supported configuration is shown in Fig. 15, with one client, one OR server, two domains and one gateway connecting each pair of domains. The client in this scenario wishes to look up a keyword (or a set of keywords), the query packet containing the keyword gets routed across domain (through routes) and delivered to the remote ORS, the ORS generates a content ID (or set of content IDs), and the response packet containing the content ID associated with that keyword gets routed back (through reversed routes) to the client.

The model consists of nodes, which can be of types client, gateway or ORS (object resolution server). Each node belongs to (is attached to) one or more domains, which can be of types IP domain, NDN domain or MF domain. Also, each node has one or more Node IDs (IP address, NDN client ID or Node GUID) and a set of demux values (ports, NDN request IDs or MF request IDs). A client wants to search for a set of keywords. The gateways keep states as a three-tuple relation made up of the $< nodeID, demux >$ of the requesting side and $< demux >$ of the serving side. Each ORS contains a mapping between keywords and content IDs. This map can be a static table or an algorithm ORS has to run in order to generate the content ID associated with a searched term. The formal model, being a description of what the system does, does not capture how content IDs are generated by the ORS.

```
abstract sig Node{domains: set Domain, id: set
    NodeID, demux: set Demux}
sig Client extends Node{want: set
    Keyword}{#domains=1 && #id=1 && #demux>=1}
sig GW extends Node{state: (NodeID set -> set
    Demux) set -> set Demux}
  {#domains=2 && (no disj d1,d2: Domain | (d1 in
      IPdomain && d2 in IPdomain) || (d1 in
      NDNdomain && d2 in NDNdomain) || (d1 in
      MFdomain && d2 in MFdomain)) && #id=2 &&
      #demux>=2}
sig ORS extends Node{map: Keyword one -> one
    ContentID}
  {#domains=1 && #id=1 && #demux>=1}
abstract sig NodeID{}
sig IPaddress extends NodeID{}
sig NDNclientID extends NodeID{}
sig NodeGUID extends NodeID{}
abstract sig Domain{}
sig IPdomain extends Domain{}
sig NDNdomain extends Domain{}
sig MFdomain extends Domain{}
abstract sig Demux{}
sig Port extends Demux{}
sig NDNreqID extends Demux{}
sig MFreqID extends Demux{}
sig Keyword{}
sig ContentID{}
```

In the above description lines, the keyword sig denotes definition of objects, $extends$ is used to define a subtype of an object that would inherit all the original attributes and abstract means the object type will only be generated in the form of its extended types. Additional constraints ($facts$) can be added to the model as separate statements or right after definition of each object. In the above piece of code, we are limiting the number of domains (and node IDs) associated with a client (or ORS) to one and for gateways to two. Also, each node has at least one demux values to use. The rest of the constraints for gateway make sure a gateway is connecting two distinct (disjoint) domains to each other. We also defined constraints (not shown here) on how the type of node ID and demux should be associated with a node's domain.

To model communication between nodes, we define an object type called $Route$, which couples together two notions of a series of links and packets carried over them. Each route is a directed edge of the network graph starting from an initiator node going to an acceptor node, both attached to the same domain. A route contains the keyword, client domain, ORS domain and demux value (which is generated by the initiator node). Constraints dictate that initiator and acceptor of a route need to be distinct to avoid loops. It should be noted that client and initiator are not necessarily the same for a route. Specific types of routes, i.e. IP routes, NDN routes and MF routes, can contain additional attributes and constraints specific to their domains.

```
abstract sig Route{initiator, acceptor: Node,
    demux: Demux, domain: one Domain, keyword:
    Keyword, clientdomain: Domain, orsdomain:
    Domain}
  {initiator!=acceptor && domain in
      initiator.domains && domain in
      acceptor.domains && demux in initiator.demux}
sig IProute extends Route{srcIP, dstIP: IPaddress,
    dstPort: Port}
  {domain in IPdomain && srcIP in initiator.id &&
      dstIP in acceptor.id && dstPort in
      acceptor.demux && demux in Port}
sig NDNroute extends Route{clientID: NDNclientID}
  {domain in NDNdomain && clientID in initiator.id
      && demux in NDNreqID}
sig MFroute extends Route{srcGUID, dstGUID:
    NodeGUID}
  {domain in MFdomain && srcGUID in initiator.id &&
      dstGUID in acceptor.id && demux in MFreqID}
```

As the data carried in query packets differs from that in response packets, namely keyword as opposed to content ID, routes need to reflect both types of data. As adding the feature of toggling between keyword and content ID in route will make the model exponentially larger, we define a new object for reverse routes called $RouteR$ and copy the rest of Route to RouteR.

```
abstract sig RouteR{initiator, acceptor: Node,
    demux: Demux, domain: one Domain,
    contentID:ContentID}
  {initiator!=acceptor && domain in
      initiator.domains && domain in
      acceptor.domains && demux in acceptor.demux}
```

As with Route, $IProuteR$, $NDNrouteR$ and $MFrouteR$ can be similarly extended from RouteR. The constraints for both sets of objects are the same. An additional constraint on reverse routes explicitly says that in case of NDN domains,

for each reverse route inside NDN there should be one (query) route corresponding with it. This is to ensure that our model respects the reverse path forwarding policy of NDN. There is no such policy for IP and MF domains.

```
fact NDNReversePath{
  all rr:NDNrouteR|
    some r:NDNroute, o:ORS|(r.keyword ->
        rr.contentID) in o.map =>
        rr.initiator=r.acceptor &&
        rr.acceptor=r.initiator &&
        rr.domain=r.domain &&
        rr.contentID=o.map[r.keyword] &&
        r.demux=rr.demux && r.clientID=rr.clientID
}
```

We define a special object called *connections* with only one instance, that captures the relationship between routes in the network. The relation *connected* keeps ordered pairs of routes where the acceptor of the first route is the same as the initiator of the second route. $connectedR$ does the same thing for reverse routes.

```
one sig Connections{connected: Route -> Route,
    connectedR: RouteR -> RouteR}
```

The following constraint says that two routes are in connections's connected relation if and only if they are actually connected and carrying the same data (belong to the same session). There is a similar constraint for reverse routes, with the difference that for reverse routes, there is one extra condition: the gateway that connects the two routes must be keeping the state associated with the session, i.e. the three tuples explained earlier.

```
fact connected{
  all r1,r2:Route , c:Connections |
    (r1->r2) in c.connected iff r1.acceptor =
        r2.initiator && r1.keyword=r2.keyword &&
        r1.clientdomain=r2.clientdomain &&
        r1.orsdomain=r2.orsdomain
  all r1,r2:RouteR , c:Connections |
    (r1->r2) in c.connectedR iff r1.acceptor =
        r2.initiator && r1.contentID=r2.contentID
        && (some gw:GW| r1.acceptor=gw &&
        r2.initiator=gw => some n:NodeID,
        d1,d2:Demux| (n->d1->d2) in gw.state && n
        in r2.acceptor.id && d1 in
        r2.acceptor.demux && d2 in
        r1.acceptor.demux && d1=r2.demux &&
        d2=r1.demux)
}
```

The above fact only models direct connections between routes. However, there can be routes that are connected indirectly, i.e. through multiple domains in non-adjacent way. To make sure the model captures this, we define the following constraint:

```
fact pathexists{
  all co:Connections, disj n1,n2:Node, k:Keyword,
      cd:Domain, od: Domain|
    (some c:Client, o:ORS| cd in c.domains && od in
        o.domains => (some r1,r2:Route | (r1->r2)
        in ^(co.connected) && r1.initiator=n1 &&
        r2.acceptor=n2 && r1.keyword=k &&
        r2.keyword=k && r1.clientdomain=cd &&
        r2.clientdomain=cd && r1.orsdomain=od &&
        r2.orsdomain=od))
  all co:Connections, disj n1,n2:Node,
      conid:ContentID, cd:Domain, od:Domain| (some
      c:Client, o:ORS| cd in c.domains && od in
      o.domains => (some r1,r2:RouteR | (r1->r2) in
      ^(co.connectedR) && r1.initiator=n1 &&
      r2.acceptor=n2 && r1.contentID=conid &&
      r2.contentID=conid))
}
```

The above constraint says that there should be a path (of any length) between any two nodes of the network; *i.e.*, no node is unreachable. To model that, we use the transitive closure of the connected (and connectedR) relation. If the ordered direct connections are represented by $C = \{(r1, r2), (r2, r3)\}$, then its transitive closure $C^+ = \{(r1, r2), (r2, r3), (r1, r3)\}$ will represent existing paths, of length one or more.

There are a number of more constraints on the model to make sure we only deal with interesting instances. These constraints include having at least one gateway for every domain, each client wanting at least one keyword and no two nodes having the same node ID.

After the description is done, we need to specify what we expect from the model. In particular, we need to formalize the essential requirements in form of properties. Before verifying the properties, we run empty predicates to make sure the model generates all interesting instances. In the rest of this subsection, we provide definitions of each property, determine the bound within which each should be checked and provide the outcome of each verification.

Property 1.1. DCR Reachability: Ability to get dynamic content identified by a user specified string. For every client that wants to get dynamic content (*e.g.* search for a keyword) and has a direct route to a gateway, there is a server reachable from the gateway to provide the dynamic content (ID) for that query. This assertion, is an invariant that needs to be true in all instances.

```
pred reach[c:Client, k:Keyword,o:ORS,gw:GW]{
  all co: Connections| k in c.want => (some
      r:Route, cid:ContentID| r.initiator=c &&
      r.acceptor=gw && r.keyword=k && r.demux in
      c.demux => some r1,r2:Route | (r1->r2) in
      ^(co.connected) && r1.initiator=gw &&
      r2.acceptor=o && r1.demux in gw.demux &&
      r1.keyword=k && r2.keyword=k && (k->cid) in
      o.map && r1.clientdomain in c.domains &&
      r2.clientdomain in c.domains && r1.orsdomain
      in o.domains && r2.orsdomain in o.domains)
}
```

This property defines a predicate $reach$, which is the formalization of the logical statement described in the definition of property 1.1. We run this predicate to see if the predicate is true in "some" instances, *i.e.*, in instances where it is known to be true (we do this sanity check for all subsequent properties, but do not state it each time).

```
run reach for 1 Client, 2 GW, 1 ORS, 6 NodeID, 3
    Domain, 6 Port, 3 NDNreqID, 3 MFreqID, 1
    Keyword, 1 ContentID, 12 Route, 1 Connections,
    12 RouteR
```

To check this assertion, we have to define the checking bound, i.e. the maximum number of distinct objects of each type. This bound gives us a large enough instance space. For this property, we pick the maximum number of three domains, two gateways, one client, one ORS and one keyword. We pick those values since they are large enough to check all possible "interesting" instances and yet not too large that would lead to problem size explosion and computation limitation while running the automated verifier. For example, with 4 nodes, a maximum of 12 query routes are possible. After running the predicate within the bound and verifying the formalization, we then define the assertion associated with this predicate.

```
assert reach{
  all c:Client, k:Keyword| some o:ORS, gw:GW|
      reach[c,k,o,gw]
}
```

It should be checked that the predicate reach holds for all clients and keywords and for at least one ORS and gateway.

```
check reach for 1 Client, 2 GW, 1 ORS, 6 NodeID, 3
    Domain, 6 Port, 3 NDNreqID, 3 MFreqID, 1
    Keyword, 1 ContentID, 12 Route, 1 Connections,
    12 RouteR
```

We check the assertion within that bound. Checking this assertion within a reasonable bound, *i.e.*, the maximum number of each object type that will generate all interesting configurations, Alloy Analyzer looks for a counterexample among all possible instances, and if none is found, the property is preserved. Our model passed this property verification successfully.

Property 1.2. DCR Returnability: Ability to get the dynamic content back from server. For every client reaching a server, there is a path back to client to carry associated dynamic content.

```
pred return[c:Client, k:Keyword, o:ORS, gw:GW]{
  all co:Connections| some gw1:GW |
      reach[c,k,o,gw1] => (some r,r1,r2:RouteR|
      (r1->r2) in ^(co.connectedR) &&
      r1.initiator=o && r2.acceptor=gw && r2.demux
      in gw.demux && r1.contentID=o.map[k] &&
      r2.contentID=o.map[k] && r.initiator=gw &&
      r.acceptor=c && r.demux in c.demux &&
      r.contentID=o.map[k])
}
assert return{
  all c:Client, k:Keyword, o:ORS| some gw:GW|
      return[c,k,o,gw]
}
```

We check this property within the same bound as property 1.1, i.e., a reasonable value for maximum number for each object so that we check all interesting instances, and observe that it passes the verification successfully.

Property 1.3. No conflict between two distinct requests from the same client. For every client that searches for two distinct requests, two distinct responses should return, and be correctly associated with each request. This (and next) property is important to meet the requirements for dynamic data and for the associated sessions to be unique.

```
pred noconflict1[c:Client, disj k1,k2:Keyword]{
  some o1,o2:ORS, gw1,gw2:GW | return[c,k1,o1,gw1]
      && return[c,k2,o2,gw2] => some n1,n2: NodeID,
      d1,d2,d3,d4:Demux| (n1->d1->d2) in gw1.state
      && (n2->d3->d4) in gw2.state && n1 in c.id &&
      d1 in c.demux && d2 in gw1.demux && n2 in
      c.id && d3 in c.demux && d4 in gw2.demux &&
      !(n1=n2 && d1=d3 && d2=d4) && (some disj
      r1,r2:RouteR| r1.initiator=gw1 &&
      r1.acceptor=c && r1.contentID=o1.map[k1] &&
      r1.demux=d1 && r2.initiator=gw2 &&
      r2.acceptor=c && r2.contentID=o2.map[k2] &&
      r2.demux=d3)
}
assert noconflict1{
  all c:Client, disj k1,k2:Keyword |
      noconflict1[c,k1,k2]
}
```

Since we need to have at least two keywords for this property, we increase the number of keyword elements in the bound from 1 to 2. We still check the assertion for one client and one ORS, and We observer that the verification passes successfully. One interesting thing about this property is that it shows the importance of the use of request IDs (both for NDN and MF) in our framework as they demultiplex distinct requests made by the same client. If we remove the request IDs from query packets, this property will be violated and counterexamples will arise. In particular, gateway will not know how to match two ORS request for keywords $k1$ and $k2$ from client c, if the state information at it looks like <c,gw_demux1> and <c, gw_demux2>, instead of one that also contains a field for client demux values associated with each requests.

Property 1.4. No conflict between two identical requests from two distinct clients. For every dynamic content requested (*e.g.* keyword searched) by two distinct clients, two distinct appropriately associated responses should come back.

```
pred noconflict2[c1,c2:Client, k:Keyword]{
  some o1,o2:ORS, gw1,gw2:GW | return[c1,k,o1,gw1]
      && return[c2,k,o2,gw2] => some n1,n2: NodeID,
      d1,d2,d3,d4:Demux| (n1->d1->d2) in gw1.state
      && (n2->d3->d4) in gw2.state && n1 in c1.id
      && d1 in c1.demux && d2 in gw1.demux && n2 in
      c2.id && d3 in c2.demux && d4 in gw2.demux &&
      !(n1=n2 && d1=d3 && d2=d4) && (some disj
      r1,r2:RouteR| r1.initiator=gw1 &&
      r1.acceptor=c1 && r1.contentID=o1.map[k] &&
      r1.demux=d1 && r2.initiator=gw2 &&
      r2.acceptor=c2 && r2.contentID=o2.map[k] &&
      r2.demux=d3)
}
assert noconflict2{
  all c1,c2:Client, k:Keyword| noconflict2[c1,c2,k]
}
```

We change the upper bound in property 1.4 to one with two (distinct or same) clients and one keyword. It holds as

observed. This property shows the importance of the addition of NDN client IDs (IP and MF nodes have IP addresses and Node GUIDs respectively). Removing the client ID from NDN Interests, violates this property. In particular, gateway will not know how to match two ORS request for keyword k from clients $c1$ and $c2$, if the state information at it looks like <demux_c1,gw_demux1> and <demux_c2, gw_demux2>, instead of one that also contains a field for who each demux value belongs to.

Property 1.5. Same exit and entry gateway for NDN clients. For every NDN client, the first gateway on the request path is the same as the last gateway on the return path. This ensures the NDN reverse path forwarding policy.

```
pred NDNgw[c:Client, k:Keyword, o:ORS, gw1,gw2: GW]{
  reach[c,k,o,gw1] && return[c,k,o,gw2] && (some
      n:NDNdomain | n in c.domains) => gw1=gw2
}
assert NDNgw{
  all c:Client, k:Keyword, o:ORS | some gw1,gw2:GW
      | NDNgw[c,k,o,gw1,gw2]
}
```

We check the above property within the same bound as the one for property 1.1 and observe that our model preserves this policy.

2) Static Content Retrieval: The scenario for content retrieval service is very similar to the one for OR, except for the fact that the request packets carry content IDs and the response packets carry the data associated with that content ID. Most of the entities here are similar to those in the OR scenario. Client nodes in CR model issue requests using content IDs rather than keywords, gateway nodes keep state, and we have a new type of object called repository ($Repos$) that keep contents associated with content IDs.

```
sig Repos extends Node{map: ContentID one -> one
    Content}
  {#domains=1 && #id=1 && (some ipd:IPdomain| ipd
      in domains => (all cid:ContentID, c:Content|
      (cid->c) in map => cid in URL)) && (some
      ndnd:IPdomain| ndnd in domains => (all
      cid:ContentID, c:Content| (cid->c) in map =>
      cid in Name)) && (some mfd:IPdomain| mfd in
      domains => (all cid:ContentID, c:Content|
      (cid->c) in map => cid in ContentGUID))
}
```

The additional constraints determine the type of content IDs each repository can have based on what its domain is. In the CR service, as opposed to the OR model, we distinguish between various extended types of content ID objects.

```
abstract sig ContentID{}
sig URL extends ContentID{}
sig Name extends ContentID{}
sig ContentGUID extends ContentID{}
```

We also have a new object type $Content$, which represents the piece of data the client wishes to retrieve.

```
sig Content{}
```

As packet fields carried in OR and CR models differ, route and reverse route objects in this model contain content ID and content respectively. MF routes need not necessarily carry destination GUIDs in this model. If the repository is in MF domain, only the route inside the destination MF domain needs to carry repository node GUID, supposing the GNRS-based binding happens in the final domain.

```
sig MFroute extends Route{srcGUID: NodeGUID,
    dstGUID: lone NodeGUID}
```

The rest of the major parts of the model, including object definitions and constraints, are similar to the OR model. At this point, we go over the two essential properties required in CR scenarios and attempt verifying them.

Property 2.1. SCR Reachability: Ability to reach repository containing the content. For every client that wants to retrieve content associated with a content ID and has a direct route to a gateway, there is a repository, reachable from that gateway, with the content associated with that content ID.

Property 2.2. SCR Returnability: Ability to get the content back from the repository. For every client that reaches a repository with a request, there is a path back to the client to carry the associated content.

We specify these two properties similar to the analogous ones in the OR section and check them within the same bound. Both verifications were successful. Thus, the model preserves both properties.

B. Publish/Subscribe

Subscribers subscribe to a prefix. The term prefix is used generically: it can be interpreted as a class of objects based on a topic-based hierarchy or just one particular object. In addition to routes, we have the notion of groups and prefixes, for which gateways maintain state. We have an object type, publisher (an 'active repository') that publishes content associated with a content ID. Content IDs have a new attribute, the prefix, which based on domain-specific interpretation can be exactly the same as the content ID or the prefix in the hierarchy the content ID is an immediate successor (child) of.

Prefix Structure. For the case of NDN, we need to model the hierarchy of existing prefixes in form of a tree. Figure 16 shows a small prefix tree example with 5 prefixes. The root prefix $P1$ is "/" and the rest of the prefixes can be something like this: $P2$ = "$/sports$", $P3$ = "$/news$", $P4$ = "$/sports/football$" and $P5$ = "$/sports/basketball$". It should be noted that the tree in the figure only displays prefix nodes and not content name nodes. If we want to show a content name such as "$/sports/football/1.mp4$", it would be an additional leaf node of the tree, as a direct child of its closest prefix, which is $P4$ in this case.

To represent the structure of hierarchical prefixes, we use binary relations to model immediate parent-child relationship between prefixes, and its transitive closure to model the ancestor-descendant relationships. For example, in the tree in Figure 16, if $P = \{(P1, P2), (P1, P3), (P2, P4), (P2, P5)\}$

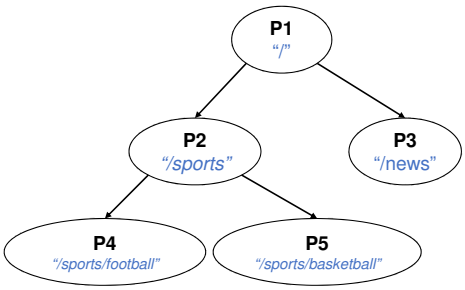


Fig. 16: Prefix tree example.

represents the immediate relationships of names, its transitive closure

$$P^{+} = \{(P1, P2), (P1, P3), (P2, P4), (P2, P5), (P1, P4), (P1, P5)\}$$

represents all ancestral relationships.

The following code shows how this can be modeled in Alloy:

```
sig Prefix{parent: lone Prefix, domains: some
    Domain}
one sig PTree {map: Prefix set -> set Prefix}
```

The following constraints on the prefix tree collectively enforce the non-existence of loops, existence of exactly one root prefix and the root prefix being the ancestor of all other prefixes.

```
fact PrefixRules{
  all p:Prefix| p.parent!= p
  all p1,p2:Prefix, pt:PTree| p1 in p2.parent <=>
      (p1->p2) in pt.map
  one p:Prefix| #p.parent=0
  all pt:PTree| no p:Prefix| (p->p) in ^(pt.map)
  all disj p1,p2: Prefix, pt:PTree | #p1.parent=0
      => (p1->p2) in ^(pt.map) }
```

We also define a new object type called "group", representing a set of subscribers and a source provider associated with a prefix and based on the domain. It can be an IP multicast group, MF multicast group (with multicast GUID) or COPSS with subscription tables.

In the pub/sub model, there is no need for a formal distinction between forward and reverse routes. However, we add a relation, "chain" (and "reverse chains"), to the special atom "connection" to model connections between adjacent groups (on either side of a gateway) that serve the same prefix, *i.e.*, an agent can join to receive relevant data being pushed. Groups G1 and G2 form a chain if and only if the publisher of G1 can be a subscriber of G2, and is then able to relay data received from G2 to his subscribers in G1. Chain and reverse chain relations represent chains for subscriptions and retrieving publications.

We add constraints on how paths exist between any two nodes and also chains exist between any two groups. With a few minor additional constraints, we specify and check the essential properties.

Property 3.1. Ability to subscribe to any prefix. For every client that wants to retrieve future publications under/associated with an existing prefix and has a direct route to a gateway, there is some publisher that will publish content under that prefix and is accessible through a chain of subscription groups.

```
pred sub[c:Client, p:Prefix]{
  all co:Connection | p in c.want => ((some
      pub:Publisher, cid:ContentID, con:Content|
      (cid->con) in pub.map && cid.prefix=p =>
      (some r1,r2:Route| r1.initiator=c &&
      r2.acceptor=pub && (r1->r2) in
      ^(co.connected) && some g1,g2:Group|
      g1.domain=c.domain && g2.domain=pub.domain &&
      g1.prefix=p && g2.prefix=p && (g1->g2) in
      ^(co.chain))))
}
assert sub{
  all c:Client, p:Prefix| sub[c,p]
}
```

We check this property within a reasonable bound (we need it to be at least 3 prefixes) where the upper bound on the number of prefixes is 3. It gives us interesting combinations, such as one *e.g.*, "$P1$ is parent of $P2$ and $P2$ is parent of $P3$" and "$P1$ is parent of both $P2$ and $P3$".

Property 3.2. Ability to receive any content published directly associated with the subscribed prefix. For every client that has subscribed to a prefix and can reach the associated publisher, there is a path back to the client to carry any content with a content ID belonging to that prefix.

```
pred rcv[pub:Publisher, con:Content, cid:ContentID]{
  all co:Connection| (cid->con) in pub.map =>
      ((some c:Client, p:Prefix| p in c.want &&
      cid.prefix=p => (some r1,r2:Route|
      r1.initiator=pub && r2.acceptor=c && (r1>r2)
      in ^(co.connected) && some g1,g2:Group|
      g1.domain=pub.domain && g2.domain=c.domain &&
      g1.prefix=p && g2.prefix=p && (g1->g2) in
      ^(co.chainR))))
   }
assert rcv{
  all pub:Publisher, con:Content, cid:ContentID |
      rcv[pub,con,cid]
}
```

Property 3.3. Ability to receive all content published that is associated with prefixes under the subscribed prefix. This property only applies when the publisher is in the NDN domain. It says that for every client that has subscribed to a prefix and has reached the associated publisher, there is a path back to the client to carry any content with content ID either directly belonging to that prefix or under it downstream on the prefix tree. The following code segment is the specification of this property:

```
pred rcvall[pub:Publisher, con:Content,
    cid:ContentID]{
  all co:Connection, pt:PTree| (cid->con) in
      pub.map and pub.domains in NDNdomain =>
      ((some c:Client, p:Prefix| (p in c.want or
      (all p1:Prefix| (p1->p) in ^(pt.map) && p1 in
      c.want)) && cid.prefix=p => (some
      r1,r2:Route| r1.initiator=pub &&
      r2.acceptor=c && (r1>r2) in ^(co.connected)
```

```
      && some g1,g2:Group| g1.domain=pub.domain &&
      g2.domain=c.domain && g1.prefix=p &&
      g2.prefix=p && (g1->g2) in ^(co.chainR))) }
assert rcvall{ all pub:Publisher, con:Content,
    cid:ContentID| rcvall[pub, con, cid] }
```

The above assertion $rcvall$ depends on how relationships among groups and also between content IDs and prefixes are represented by $Connection$ and $PTree$ atoms. If we had an NDN domain with a namespace that does not capture relationships between prefixes, *i.e.*, not leveraging the hierarchical structure of names, then $rcvall$ would be equivalent of receiving a single content element (Property 3.2.). We check that this property also holds. Properties 3.1-3 collectively model and verify the hierarchical subscription/publication we expect from this service.

The formal model is a means to verify and show correctness of our interoperability framework and raise the confidence on its essential requirements.

C. Failure and Mobility

In addition to the basic invariants, there are other important aspects of formal analysis of networks that warrant a more quantitative analysis; among them are failure and mobility analysis. Failures and mobility of nodes can occur in a network, causing disruption and lack of content availability. To better compare how different network architectural components, *e.g.*, routing, impact the number of success and violation scenarios, we perform model counting [30].

While Alloy Analyzer (v4.20) [31] allows for a limited, graphical iteration over instances, it does not enable an explicit counting of instances in an efficient manner. To perform model counting, we wrote an application [32] that counts all SAT solutions, using the SAT4J solver [33] (SAT4J can be replaced by any off-the-shelf SAT solver). We feed the Alloy model and properties, in Kodkod format [34], to our application. Predicates and assertions are used for counting instances that satisfy or violate (counterexamples) respectively. Through this counting, we can also look into the details (relations and values) within each instance, and gain insight such as possible cause of violations (in case of counterexamples) and calculate the probability of occurrence of each instance in real-world scenarios.

1) Failure: Our interoperability framework depends on gateways that retain state information. What would happen to a response packet if that state is lost at the gateway for any reason? For reliability, we consider state sharing between redundant gateways that have the same domains on either side. Fig. **??** depicts an example for this. Consider the gateway that received the request and created the state as the *primary* gateway for the request ($GW1$ in the Fig.), and the replicas that have the shared state as the *secondary* gateways ($GW2$ and $GW3$). Formally, we add an extra condition to our reachability and returnability properties such that, for two routes to connect, the gateway attaching them must be up and running at the time the packet is received. Additionally, for returnability, the state information must be present at the gateway. If any gateway goes down, the corresponding potential path going through it ($p1$–3) back for the content cannot be leveraged. If the gateway is neighboring an NDN domain (*e.g.*, in $Domain_n$ or $Domain_{n-1}$), then the gateway has be the primary only, for correct operation with the NDN reverse-path-forwarding (RPF) policy [2]. For other domain types, a secondary gateway that is active and has the shared state information is adequate to forward the response data back. We model the conditions representing this in Alloy as shown in Listing 1.

Listing 1: Failure scenario constraints.

```
all r1,r2: Route, c:Connections| --forward routes
    (request) condition
   (r1->r2) in c.connected iff r1.acceptor =
       r2.initiator && r1.initiator.status1 in Up
       && r2.initiator.status1 in Up
all r1,r2: RevRoute, c:Connections| --reverse
    routes (response) condition
   (r1->r2) in c.connectedR iff r1.acceptor =
       r2.initiator && r1.initiator.status1 in Up
       && r2.initiator.status1 in Up &&
  ((r1.domain in NDNdomain or r2.domain in
      NDNdomain) => r1.acceptor.type in Primary) --
      NDNdomain enforces RPF policy
```

Gateways can go down due to various reasons such as completely failing or just losing state information due to a software failure. Our method can be used to reason about various scenarios and measure failure probability given an input configuration space, *i.e.*, a set of Alloy facts that set constraints on some objects or variables while relaxing others. As Table I shows, a simple model finding analysis does not provide a helpful comparison between different such constraints: it will say that both cases lead to counterexamples are raised (*e.g.*, for the case that all gateways go down). To gain a better assessment of which constraint does better, we resort to model counting (Table II). Using model counting, we can count (satisfying) instances (I) and counterexamples (C), and calculate (even if approximately) the probability of reliability ($R = I/(I + C)$). This reliability indicates to what degree interoperability is impacted in presence of failure, given certain conditions (*i.e.*, choice of domain policies, *etc.*).

2) Mobility: To model and analyze mobility (Fig. 18), we add the notion of "time" to our model. In particular, we associate timeout values to state entries at gateways and $birthTime$ and $deathTime$ to routes (and similarly for reverse routes). We assume gateways are stationary, but a consumer and/or producer can move, causing the "death" of their route ($route1$) to/from their closest gateway. A new route to the gateway is "born" ($route2$) after some time, assuming the existence of a domain-specific method to handle mobility. Temporal conditions must be incorporated into reachability/returnability properties. The most interesting case is when a mobility event occurs while the packet is in-flight. At high-level, the sum total latency formulated as $firstDeliveryAttempt + recovery + secondDeliveryAttempt$, must be below a certain $expiration$ threshold (at every gateway and consumer). $firstDeliveryAttempt$ is the incomplete partial delivery la-

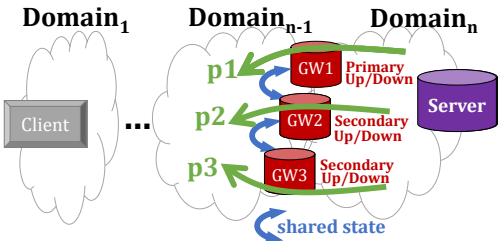


Fig. 17: Gateway failure scenario.

TABLE I: Model finding.

Domain n constraints	Return-ability
Const. 1	x
Const. 2	x

TABLE II: Model counting.

$Domain_n$ constraints	Returnability		
	I	C	R
Const. 1	x_1	y_1	$x_1/(x_1+y_1)$
Const. 2	x_2	y_2	$x_2/(x_2+y_2)$

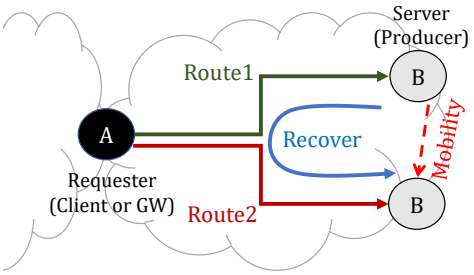


Fig. 18: Mobility scenario example: Route 2 established after B moves and changes its point of attachment.

tency via $route1$ and $secondDeliveryAttempt$ is the delivery via $route2$ (continuation in MF, and complete retransmission in IP and NDN). The $recovery$ delay is the time it takes for the packet to be transmitted back on the new path again; it includes re-registration (MF and IP), FIB re-population (IP and NDN in case of provider mobility) and/or PIT re-population (for NDN in the case of consumer mobility) delays. Using this formal method, we check properties in the presence of mobility, find appropriate values for a timeout threshold on gateways and investigate the effect of domain-specific mobility handling methods on interoperability. Listing 2 generally specifies how the reachability property (to deliver a named request) depends on the condition of mobility (stationary or mobile) and the domain policy on handling mobility (early binding or late binding). Returnability is similarly specified (for content). Predicates $stationary$, $mobileEarlyBinding$, and $mobileLateBinding$ specify timing conditions for successful delivery assuming their corresponding conditions (details of the three properties are omitted here due to space but are in [32]). As shown in Fig. 18, we only consider intra-domain mobility here, *i.e.*, the mobile node changes its location and point of attachment, but stays within its domain.

Listing 2: Reachability in presence of mobility.

```
pred reach[c:Client, p:Producer, cid:ContentID]{ --
    a client and content producer
  (stationary[c,p,cid] && p.mobility in Stationary)
      -- producer stationary
  or (mobileLateBinding[c,p,cid] && p.mobility in
      Mobility && Domain.binding in LateBinding) --
      producer mobile, domain does late binding
  or (mobileEarlyBinding[c,p,cid] && p.mobility in
      Mobility && Domain.binding in EarlyBinding)}
      -- producer mobile, domain does early binding
```

D. Formal Analysis Results

We implemented the ICI framework discussed in our model, with gateways for interoperation among IP, NDN, and MF in a software testbed (implementation details in [28]). This section provides the description and results of our analysis of the ICI framework.

To check for correctness, we performed verification (supported by Alloy Analyzer's model finding engine) of our ICI framework model, against the information-centric services properties. In order to reach convincing proofs (as advised in [35]), we pick the scopes for verification in Alloy that are large enough to contain all necessary cases, and small enough so that we do not encounter model explosion. The scopes, *i.e.*, upper bounds on the number of key objects, are provided in Table III. For most properties, we consider 1 $Client$, 1 $Server$, 1 $Content$, and 1 $ContentID$. That is, different <client, request> pairs are considered independent of each other. However, for Properties 2.3.a/b, such a dependency matters, and we want to show lack of conflicts. For Property 2.3.a, we set 1 $Client$ and 2 $Content$s (to generate scenarios where *one* client makes *two* separate request for *two* different contents), and for Property 2.3.b, we set 2 $Client$s and 1 $Content$ (to look for conflicts between request for *one* content but by *two* clients). We use 3 $Domain$s for most properties, as it contains all cases with 1, 2, or 3 domains of any type, *i.e.*, IP, NDN, or MF. Also, with upper bound n on the total number of $Node$s, *i.e.*, sum of $Client$s, $Server$s, and GWs, we specify the upper bound on the number of $Route$s (as well as $RevRoute$s) to be $n(n-1)$, enabling the existence of any possible (uni-directional) route. For pub/sub services (*i.e.*, Properties 3.1–3), we set 3 $Prefix$es, $ContentID$s, and $Content$s, to capture inter-relationship of content IDs in a large enough namespace. Additionally, with the upper bound on $Domain$s and $ContentID$s both set at 3, we set the upper bound on total number of $Group$s (and $GroupID$s) to be 3×3=9, so as to contain cases with one group per content ID per domain. The blank cells in Table III indicate either "N/A" or "no particular upper bound set", in which case Alloy picks a default value. Within this scope, our verification passes successfully for each property, showing that the stated properties are *invariants* of our ICI framework. In other words, the framework design ensures that *any sequence of interconnected IP, NDN, and MF domains are information-centrically interoperable*.

We use our proposed model counting approach to analyze

TABLE III: Verif. scopes for ICI services properties.

Property	Client	GW	Repos/ Server/ Publisher	Domain	ContentID/ Keyword	Prefix	PTree	Content	Connections	Route	RevRoute	NodeID	Port	NDNreqID	MFreqID	Group	GroupID	Verif. Result
1.1	1	2	1	3	1			1	1	12	12	6						YES
1.2	1	2	1	3	1			1	1	12	12	6						YES
2.1	1	2	1	3	1			1	1	12	12	6	6	3	3			YES
2.2	1	2	1	3	1			1	1	12	12	6	6	3	3			YES
2.3.a	1	1	1	2	2			2	1	8	8	4	4	4	4			YES
2.3.b	2	1	1	2	1			1	1	8	8	5	4	4	4			YES
3.1	1	2	1	3	3	3	1	3	1	12	12					9	9	YES
3.2	1	2	1	3	3	3	1	3	1	12	12					9	9	YES
3.3	1	2	1	3	3	3	1	3	1	12	12					9	9	YES

scenarios with the failure of one or multiple gateways. The most important factor affecting returnability in scenarios with the possibility of failure, is domain-specific routing policies, in particular, whether or not it allows for a secondary (backup) gateway to relay the returning response content. Different domains have different policies; MF and IP decouple the forward (request) and return (response) paths, and they can be through different gateways, while NDN strictly requires the two paths to be the same, due to RPF policy. To investigate the impact of that policy, we considered a scenario of two domains, with two gateways between them (one primary and one secondary), sharing state. Both gateways are Up (working) when the request is forwarded, and either *may* go $Down$ (failing) when the response is one its way back. Table IV shows different scenarios for reachability and returnability, with different domain constraints (with different routing policies). In particular, the two domain constraints we consider are the following: 1) no constraint on what any of the domains are; and 2) one domain is definitely NDN. The table shows the values of I (instances), C (counterexample), and R (reliability) for each scenario. Our results for R in Table IV prove that having an NDN domain on one side dramatically reduces the returnability reliability ratio, since basic NDN forwarding strictly forbids data coming back on a different path than the original path taken by the request.

When a content producer (server) moves while a content request is in-flight (Fig. 18), the domain's handling of mobility recovery determines the reachability probability. NDN and IP use early binding with retransmissions, while MF supports late binding with rerouting. We compare the impact of these mechanisms and techniques using our model counting method, with results shown in Table V. Our modeled scenario consists of two nodes in a domain, one requester (client or gateway) and one server (producer) with a route established among them. The 'Stationary' columns in the table show reachability results in the stationary server case. With 'Mobile', the route dies due to a server mobility event (at time t=10), leading to the birth of the second route. We set the re-registration and re-population delays to 1 each. Also, a retransmission is initiated 1 time unit after the mobility event. Different binding techniques for mobility, *i.e.*, late and early binding, are also shown in Table V. We compare cases with different ranges for *Delivery Latency* (DL), which is time approximately needed for a packet to travel from requester to server. For a delivery latency range of $[0, 20]$, we see a higher R for stationary *vs.* mobility cases. The reason is that when the server does not move, the original route stays active, thus providing a higher chance for requests to reach the server. Comparing the two binding techniques, late binding leads to higher chance of reachability compared to early binding, as it allows for packets to be re-routed on the newly-born route, rather than retransmitting from the original requester. These results serve as proof that under similar scenarios, late binding outperforms early binding in ICI. Also, changing the delivery latency ranges, we can find out at what points, reachability is an invariant (if ever) under mobility conditions. As the table shows, with ranges within $[0, 18]$, $[0, 15]$, and $[0, 10]$ (rows in Table V labeled in first column accordingly), reachability becomes an invariant in cases of Stationary, Late Binding, and Early Binding, respectively; as zero counterexamples are raised. With a small enough delivery latency ranges, namely $[0, 10]$, reachability becomes an invariant, no matter the mobility conditions. Our approach can be used to find such points of invariance, comparing different techniques, and prove them.

V. Evaluation

A. Forwarding Efficiency

In order to evaluate the performance of our interoperability framework, primarily the gateway implementation performance, we set up a testbed consisting of a number of virtual machines. We use a representative installation of NDN, MF and IP: client and server as well as routers for each domain. For NDN, we used the CCNx v0.8.0, which contains all the components of NDN that is essential to our framework, including our COPSS implementation.

Our testbed consists of five VMs: $C \rightarrow R_1 \rightarrow GW \rightarrow R_2 \rightarrow P$ where client C and router R_1 are in domain D_1 and provider P and router R_2 are in domain D_2. Gateway GW, an implementation of our interoperability framework, is in between the two domains and perform the request/response translation. With distinct domains D_1 and D_2, we test all six possible scenarios. We tested functionality with a client being able to ask for a content residing in a remote domain of a different architecture and get the content back via the gateway. Table VI are the measurements for each scenario the five-node testbed. The content

TABLE IV: Failure analysis results.

Cases	Reachability			Returnability		
	I	C	R	I	C	R
No domain constraints	290	0	1.00	56	210	0.21
One NDN domain	176	0	1.00	8	168	0.04

TABLE V: Mobility analysis results.

Cases	Stationary			Mobile					
				LateBinding			EarlyBinding		
DL range	I	C	R	I	C	R	I	C	R
[0,20]	100	8	0.92	72	24	0.75	92	64	0.58
[0,18]	96	0	1.00	72	8	0.90	92	48	0.65
[0,15]	84	0	1.00	64	0	1.00	92	24	0.79
[0,10]	64	0	1.00	44	0	1.00	84	0	1.00

Scenarios (value in ms)	Overall Resp.	Provider Service	GW Processing	
			Req.	Resp.
NDN-MF	367.50	8.25	2.50	1.75
MF-NDN	363.25	0.10	12.25	2.00
IP-MF	72.75	10.50	11.50	5.00
MF-IP	64.50	5.50	9.75	5.75
NDN-IP	356.50	2.50	10.25	9.50
IP-NDN	350.75	0.18	6.00	5.25

TABLE VI: Forwarding efficiency (static content).

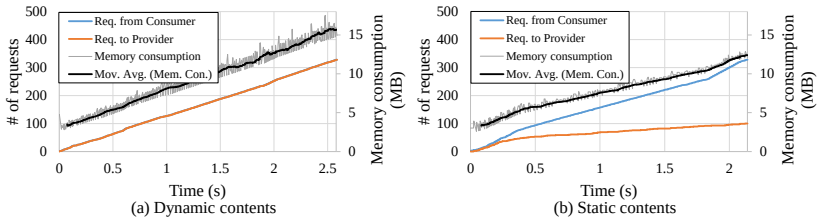


Fig. 19: Scalability: memory consumption *vs.* # of requests.

TABLE VII: Interoperability: dynamic content retrieval latency.

Scenario	Provider Service Time (ms)	Gateway Processing Delay (ms)	
		Request Processing	Response Processing
IP-MF	20	3	9
NDN-MF	29	1	11
IP-NDN	26	4	26
NDN-IP	34	17	5
MF-IP	13	16	13
MF-NDN	27	15	4

requested is 1000 bytes, and we focus on the primary component of concern: the average gateway processing latency.

Table VI, shows the average overall response time, provider service and gateway processing times (averaged over several runs, discarding outliers). The processing time at the gateway, including reformatting and maintaining state between the two domains, while not negligible, is reasonable for an initial software implementation. It contributes between 5-16 ms of processing delay, compared to the total response time of 60-360 ms in this small-scale topology. Especially in the ICN cases, the gateway contributes a relatively small portion of the overall response time.

To compare it with the response time when the client and the provider are on a single domain (replacing gateway GW with a native router R_0). We believe that the cost of interoperability is reasonable enough to encourage interoperation as a means of accessing content residing in other domains rather than replicating it in each domain. We only report results for access to static content; experiments for dynamic content access are shown in Table VII where the dynamic content providers generate a small-size response upon each dynamic content request from a remote client. We observe the absolute (average) value for gateway processing time delays, independent of the topology and total response time experienced by the end consumer, is still small enough to justify the use of interoperability gateways to access content form a foreign domain.

B. Scalability

We then use the ORBIT test bed [36] to evaluate the scalability of the proposed solution. ORBIT has a grid with 400 nodes and allows customized network topology using SDN. We run each router (forwarding engine), provider, consumer and gateway on separate physical nodes. Each machine has 4GB memory and runs Ubuntu 14.04. In our topology, we included 50 consumers (in one domain), 1 provider (in another domain) and 1 gateway. The consumers are connected to the gateway via a pre-configured access network.

We measure the amount of state stored in the gateway (memory consumption) *vs.* different number of requests from the consumers. The implementation is in Java, which has automated memory management. We call garbage collection very frequently during run-time to get a better estimate of memory consumption. This makes our gateway slightly slower compared to the regular use. We evaluate the requests to static and dynamic content separately and only the IP→NDN scenario is reported here.

Evaluation on dynamic content: Here, 50 clients send 328 dynamic content requests simultaneously. Our server stalls the response to each request for 3 seconds to try and have more requests accumulated on the gateway (3 sec. is picked since request timeout time in NDN is around 4 sec.). Fig. 19a shows the instantaneous memory consumption (and moving average over 20 values) *vs.* # of incoming and outgoing requests.

Since the consumers are requesting dynamic content, we don't see any aggregation at the gateway – each request from the client yields an outgoing request to the provider. Therefore, the incoming and outgoing request values are very close to each other in the Fig. We observed that the memory consumption grows linearly with the number of incoming requests since we keep the states for each request.

Evaluation on static content: Now the clients make 328 requests over 100 static content items simultaneously. The popularity of the content follows a Zipf distribution (α=0.81). The server still waits 3 seconds before sending the response to allow request accumulation. Fig. 19b shows the results.

Since we keep the state on the gateway, we can aggregate multiple requests for the same content (name). Therefore, the amount of requests to the provider is smaller than the amount of requests from the consumers. The memory consumption also grows sub-linearly towards the incoming requests. The memory consumption is lower here compared to the value for dynamic content with the same amount of requests.

Summary: We ran the same experiments in other domain combinations (NDN→IP, MF→NDN, *etc.*) and see similar

results. While the exact memory consumption value might be specific to the implementation, the relationship gives us some hints. We acknowledge that keeping per-session state puts more burden on the gateways (amount of state grows with the number of flows), but it is analogous (and no worse than) maintaining PIT state at NDN routers. Yet, thanks to the information-centric design, we allow aggregation at the gateways, with the potential for the architecture to scale better.

VI. Conclusion

We presented a framework for interoperability across NDN, MF and IP network domains, ensuring that the individual architectures (clients, routers, naming schema) do not change to accommodate interoperability. It is achieved by having an appropriate translating gateway at the intersection between domains. We describe the protocol and verify key properties in supporting the different communication patterns across domains. The performance of our approach is reasonable, albeit our first attempt at the gateway implementation has room for improved performance. We have identified a number of opportunities for optimization. We believe this approach is a means for building bridges between islands of different ICN networking architectures, such as NDN, MF and with IP.

References

[1] V. Jacobson *et al.*, "Networking Named Content," in *CoNEXT*, 2009.

[2] L. Zhang *et al.*, "Named Data Networking (NDN) Project," PARC, Tech. Report NDN-0001, 2010.

[3] D. Raychaudhuri, K. Nagaraja, and A. Venkataramani, "MobilityFirst: A Robust and Trustworthy Mobility-Centric Architecture for The Future Internet," *SIGMOBILE*, 2012.

[4] D. Naylor *et al.*, "XIA: Architecting A More Trustworthy and Evolvable Internet," *SIGCOMM CCR*, pp. 50–57, 2014.

[5] N. Fotiou *et al.*, "Developing Information Networking Further: from PSIRP to PURSUIT," in *BROADNETS*, 2012, pp. 1–13.

[6] S. S. Adhatarao *et al.*, "Comparison of Naming Schema in ICN," in *LANMAN*, 2016.

[7] S. S. Adhatarao *et al.*, "ORICE: An Architecture for Object Resolution Services in Information-Centric Environment," in *LANMAN*, 2015.

[8] "Alloy: A Language and Tool for Relational Models," http://alloy.mit.edu/alloy/.

[9] T. Vu *et al.*, "Dmap: A Shared Hosting Scheme for Dynamic Identifier to Locator Mappings in The Global Internet," in *ICDCS*, 2012.

[10] A. Sharma *et al.*, "A Global Name Service for A Highly Mobile Internetwork," in *SIGCOMM*, 2014.

[11] Y. Hu, R. D. Yates, and D. Raychaudhuri, "A Hierarchically Aggregated In-Network Global Name Resolution Service for the Mobile Internet," *MobilityFirst*, 2016.

[12] W. Shang *et al.*, "NDN.JS: A JavaScript Client Library for Named Data Networking," in *NOMEN*, 2013.

[13] J. Chen *et al.*, "COPSS: An Efficient Content Oriented Pub/Sub System," in *ANCS*, 2011.

[14] J. Chen *et al.*, "Coexist: Integrating Content Oriented Publish/Subscribe Systems with IP," in *ANCS*, 2012.

[15] V. Jacobson *et al.*, "VoCCN: Voice-over Content-Centric Networks," in *ReArch*, 2009.

[16] I. Moiseenko and D. Oran, "TCP/ICN: Carrying TCP over Content Centric and Named Data Networks," in *ICN*, 2016.

[17] D. Trossen *et al.*, "IP over ICN – The better IP?" in *EuCNC*, 2015.

[18] X. Marchal *et al.*, "A Virtualized and Monitored NDN Infrastructure Featuring a NDN/HTTP Gateway," in *ICN*, 2016.

[19] F. Bronzino *et al.*, "In-Network Compute Extensions for Rate-Adaptive Content Delivery in Mobile Networks," in *ICNP*, 2014.

[20] S. Wang *et al.*, "On Adapting HTTP Protocol to Content Centric Networking," in *CFI*, 2012.

[21] I. Moiseenko, M. Stapp, and D. Oran, "Communication Patterns for Web Interaction in Named Data Networking," in *ICN*, 2014.

[22] J. Rosenberg *et al.*, "SIP: Session Initiation Protocol," RFC 3261, IETF, Jun. 2002.

[23] A.-V. T. W. Group *et al.*, "RTP: A Transport Protocol for Real-Time Applications," RFC 1889, IETF, Jan. 1996.

[24] M. Baugher *et al.*, "The Secure Real-time Transport Protocol (SRTP)," RFC 3711, IETF, Mar. 2004.

[25] B. Ahlgren *et al.*, "Design considerations for a network of information," in *ReArch*, 2008.

[26] Wikipedia, "Representational state transfer," https://en.wikipedia.org/wiki/Representational_state_transfer.

[27] S. Mukherjee *et al.*, "Achieving Scalable Push Multicast Services Using Global Name Resolution," in *GLOBECOM*, 2016.

[28] "ICN Interoperability," https://github.com/saidprotocol/icninteroperability.

[29] P. Zave, *A Formal Model of Addressing for Interoperating Networks*. Springer Berlin Heidelberg, 2005, pp. 318–333.

[30] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting: A new strategy for obtaining good bounds," in *AAAI*, 2006.

[31] "Alloy: A Language and Tool for Relational Models," http://alloy.mit.edu/alloy/.

[32] https://www.cs.ucr.edu/~mjaha001/ICI.zip.

[33] D. Le Berre and A. Parrain, "The sat4j library, release 2.2, system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, 2010.

[34] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *TACAS*, 2007.

[35] P. Zave, "A practical comparison of alloy and spin," *Form. Asp. Comput.*, vol. 27, no. 2, Mar. 2015.

[36] "Open-Access Research Testbed for Next-Generation Wireless Networks (ORBIT)," http://www.orbit-lab.org/.