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

Christopher A. Wood
www.christopher-wood.com
woodc1@uci.edu

June 9, 2014

Agenda

Today's Internet: Communication Networks as Distribution Networks

The communication-centric design enables point-to-point communcation between any two parties:

- Names and interfaces
- Supports end-to-end conversations
- Provides unreliable packet delivery via IP datagrams
- Compensates for simplicity of IP via complexity of TCP

Important observation: Helped facilitate today's concent-centric world, *but was never designed for it!*

NDN is a new architecture designed for content-centric networking

Data vs Communication Networks

Distribution/data (DN) and communication (CN) networks differ in several key ways:

	CN	DN
Naming	Endpoints	Content
Memory	Invisible & limited	Explicit (storage = wires)
Security	Communication process/channel	Content

NDN Overview

Content-centric networking flips around the host-based model of the Internet architecture

- *Content names*, rather than content locations, become addressable.
- Content is retrieved via *interests*, which are similar to URLs:

  `ccnx://rit/gccis/cs/spr/ramsey_survey`
- The network is permitted to store (cache) content that is in high demand
- End result: less traffic to/from the content's original source, better usage of network resources, less latency, etc etc.

NDN Overview (continued)

How is data actually retrieved?

- A consumer C sends out an *interest* for content they desire.
- A router R_i use the information in their forwarding information base (FIB) table and data in cached in their content store (CS) to handle incoming interests:
  1. If content with the same name matches what's stored in the CS, return that content
  2. Else, store the interest in their pending interest table (PIT) (including the downstream router R_{i-1} or consumer C that made the request), and forward the request upstream to the next router R_{i+1} based on their FIB.
  3. FIBs are configured using protocol similar to OSPF
- Once the interest is satisfied in R_i, the PIT entry is cleared, the content is cached, and the data is sent downstream to C or R_{i-1}.

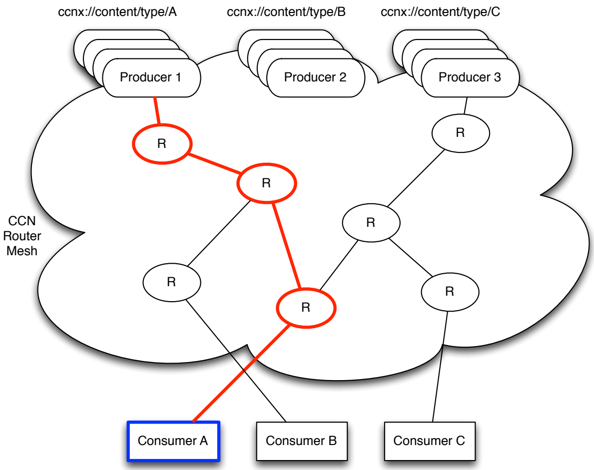
Interest Format

- Interests are similar to URLs:

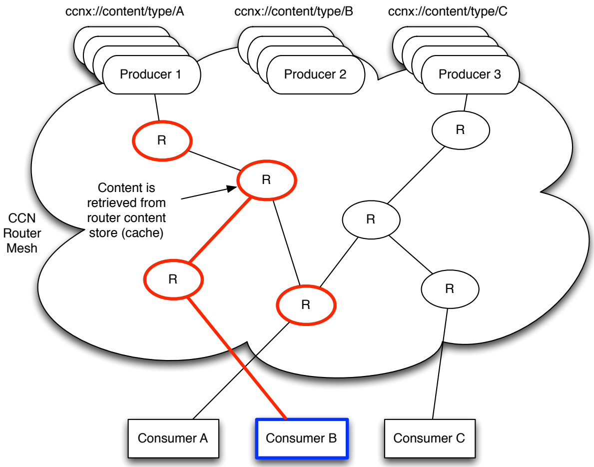
  `ccnx://rit/gccis/cs/spr/ramsey_survey`

- The / character is a delimeter that separates name *components*
- A component can be *anything*, including binary data (e.g. ciphertext)
- Interests are matched to providers in FIBs using a standard longest-prefix rule (to my knowledge, interests in CSs must match completely)

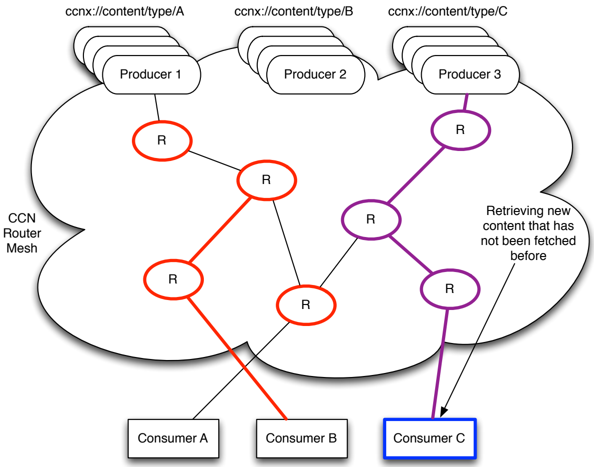
NDN in Action - #1



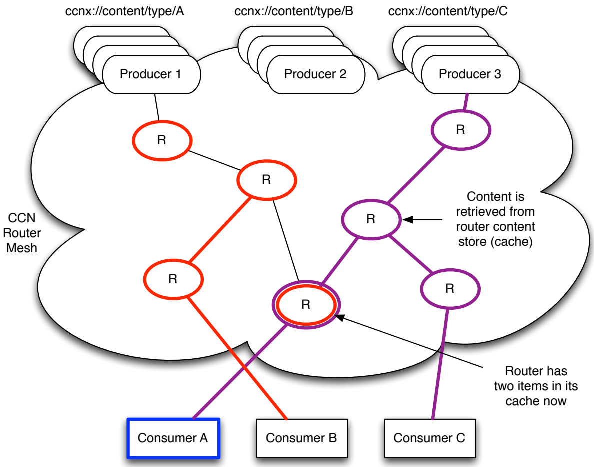
NDN in Action - #2



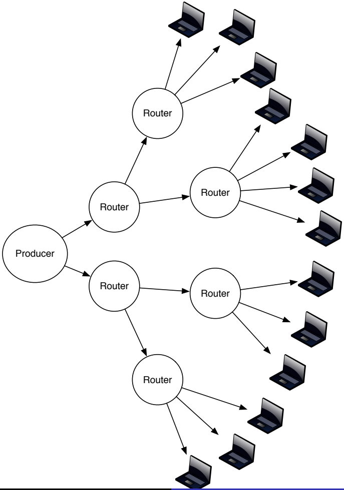
NDN in Action - #3



NDN in Action - #4

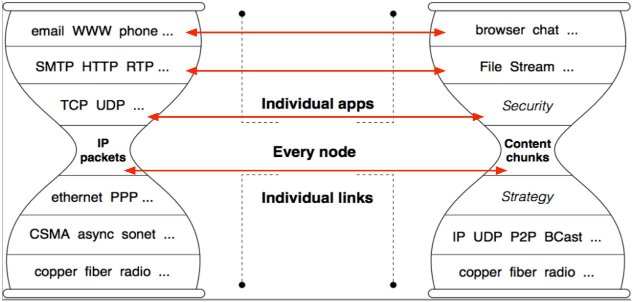


NDN at a Larger Scale



Underlying Network Differences

How will similar applications on both networks communicate with vastly different network stacks?



Motivation for CCNSink

Question: If adopted, how will NDN be deployed?

1. "Turn off" the Internet, swap in new hardware, and then flip the switch again
   - Bad idea...
2. Incrementally "roll out" NDN hardware and slowly make it interoperable with existing IP network
   - How to enable NDN-based applications to communicate with IP-based applications (and vice versa)?
   - ...and how to do this without re-writing the transport/network layer of IP-based applications to use CCNx (i.e., implement NDN functionality on top of IP)?

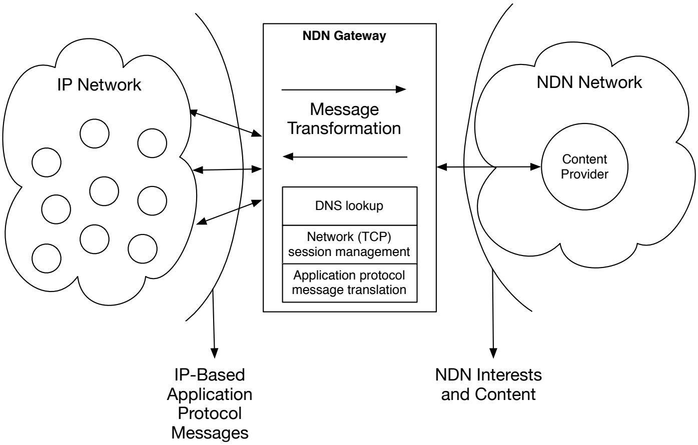
Motivation for CCNSink

Question: If adopted, how will NDN be deployed?

Answer (in other words, what CCNSink does):

- Use NDN-network edge gateways to hide the details of NDN/IP communication mechanics
- Translate IP messages to compliant NDN interests (and vice versa)
- Use NDN-network edge bridges to connect isolated NDN "islands"

Gateway Semantic Translations



IP-to-NDN Traffic

- HTTP GET requests issued to get content with a similar name
  - e.g., `GET X.X.X.X:80/ndn/ccnx/name/of/content`
  - The request path is mapped to the outgoing interest name
- TCP connections established to stream data to NDN producers
  - Socket connection between IP-based client and gateway established, NDN producer name first sent, and then all remaining data is streamed
  - The gateway partitions data from the socket and packs it into an interest for the desired NDN producer

NDN-to-IP Traffic

Interests are encoded according to a special grammar to enable the gateway to parse interests and issue them using the appropriate IP-based protocol

<ip-interest>: '/.../ip/'<protocol>.

<protocol>: 'http/'<http-cmd>['/'<http-path>] | 'tcp/'<tcp-ident>'/'<uri-encoded-string>.

<http-cmd>: 'GET' | 'PUT' | 'POST' | 'DELETE'.

<http-path>: <uri> | <ip-address>[port]['/'<uri-encoded-string>]

<tcp-ident>: <SHA256-hash>'/'<nonce>.

Bridging NDN Networks

CCNSink

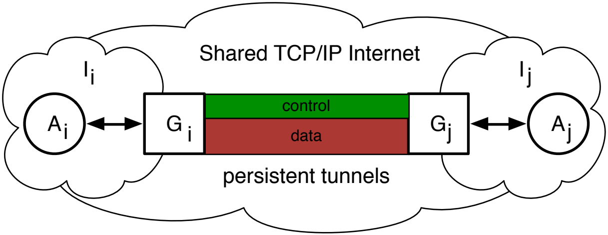
is used to bridge interests and their corresponding content across physically disjoint NDN networks.

... But the devil is in the details

- NDN stipulates that all content is signed by its producers
  - Content must be signed and verified as it crosses between two bridges, and then *re-signed* before sent to the intended NDN consumer
- Bridging should incur minimal overhead and handle high loads
  - Use keyed MAC algorithms (e.g., HMAC) to tag and verify content as it traverses bridges (instead of digital signatures)
- Bridges must be able to locate and connect to other bridges
  - Use a central directory service to maintain knowledge of all bridges (updated periodically via heartbeat messages)

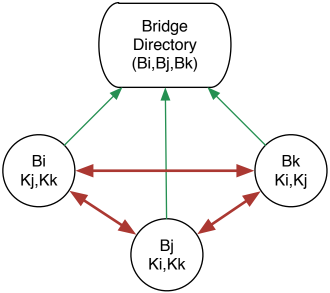
The Bridge

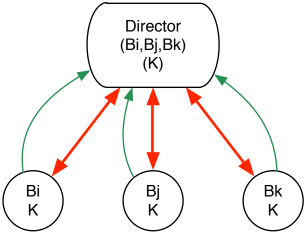
Bridges establish persistent TCP connections to stream interests and content between disjoint networks.



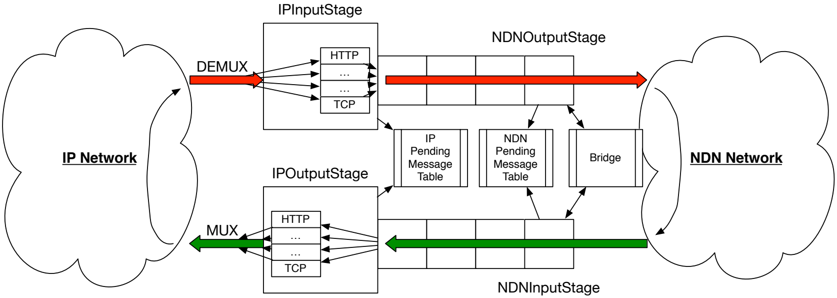
The Bridge Directory (and the Director)

A central directory (to help establish pair-wise keys) or directory (to help establish a shared group key) can be used to manage the bridges and their keys.





Pipeline-Based Load Balancing Design



Design Highlights

1. Simple pipeline stage interface:
    - Thread-safe input buffer queue of `OutgoingMessage` objects
    - Reference to “next” stage
2. Simple (and extensible) NDN-to-IP protocol multipliexing encoding
3. Intuitive IP-to-NDN encoding method via HTTP GET requests
4. All (IP and NDN) messages are handled asynchronously
    - But synchronization primitives (semaphores/events) are used to wait for message responses

Implementation Highlights

- CCNSink gateway/bridge:
  - Multi-threaded application written entirely in Python
    - Uses native Python libraries for thread synchronization primitives, IP-based communication (e.g., httplib)
  - Uses CCNx 0.82 for NDN communication
- CCNSink bridge directory:
  - Written entirely in Python
  - Uses Python Flask library to communicate with bridge clients using HTTP

Performance: Experiments and Metrics

We assessed the design and implementation performance with the following experiments:

- Bidirectional “application-layer” and “transport-layer” communication across the gateway
- Unidirectional messages sent from IP and NDN hosts

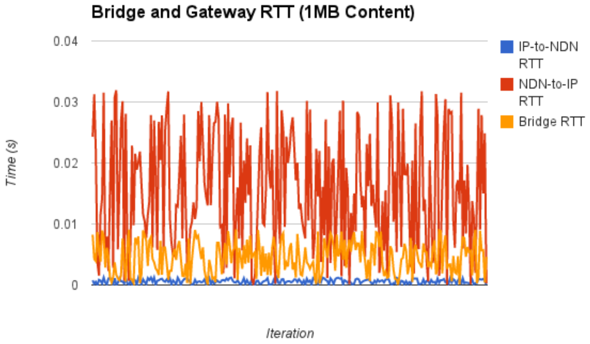
We collected the following metrics:

- Unidirectional message translation overhead
- Unidirectional message trip time (RTT)
- Bridge mode message latency (RTT)
- Bridge mode symmetric key establishment overhead time

Translation Overhead

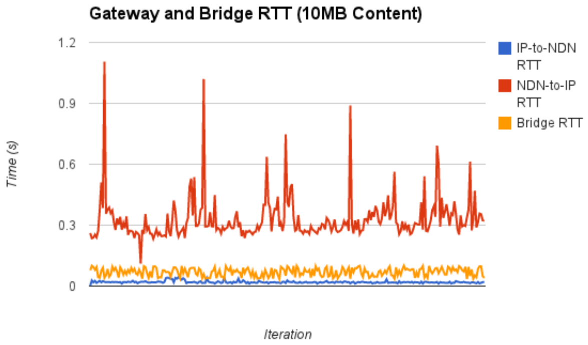
- IP-to-NDN translation: $\approx$0.00078s for interest names composed of one (1) to five (5) components
- NDN-to-IP translation: $\approx$0.0005s.

RTT Results



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

RTT Results (cont'd)



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

Key Establishment Overhead

- TODO

Future Work

Project paper submitted to ACM ICN 2014.

Next tasks include:

- Implement group-based key establishment routine for bridges
- Expand NDN-to-IP encoding grammar to support more TCP/IP protocols
- Test current CCNSink implementation under heavy message load with geographically distributed hosts
- Test CCNSink on top of actual NDN hardware (i.e., not using CCNx as NDN interface)