A Transport Layer and Socket API for (h)ICN
Design, Implementation and Performance Analysis

Luca Muscariello, Mauro Sardara, Alberto Compagno

ICN RG Interim Meeting – Montreal – 15th of July 2018
Motivation

• provide new API for apps
• socket API
• post-socket API
• evolutionary deployment of (h)ICN in existing applications
• portability
• (h)ICN = \{NDN, CCN, hICN\}
Protocol layers and namespaces

• Application namespaces
  • Manage application resources
  • Longer lifetime
  • Scope and delegation can be local or global to the participants

• Network namespaces
  • Used to manage network resource sharing
  • Shorter lifetime
  • Network resources are highly constrained

• Resources are multiplexed and demultiplexed by solving a resource allocation problem
Transport Services in (h)ICN

Application Logic

Transport
- Producer Services
- Consumer Services

Routing and Forwarding

### Application

- ADU

### Socket Library
- Segmentation
- Authentication
- Integrity
- Naming
- Fetching
- Reassembly
- Verification
- Congestion Control

### Forwarding Engine
- L3 PDU (I/D)
Transport Manifest

- **Indexing**
  - network names mapping and synchronization between consumer/producers sockets

- **Signature Verification**
  - Manifest always signed

- **Integrity Verification**
  - It contains hashes of data packets

- **Performance**
  - Amortizes verification cost of each content object
Authentication and Integrity

• Native security features, transparently offered to each application
• Two approaches: **Manifest authentication vs per packet authentication**

  **Manifest Authentication**
  
  Integrity verified with hash inside the signed manifest.
  
  **Per Packet Authentication**
  
  Integrity verified with the signature itself
Implementation

- Transport stack implemented as C++ user-space library
- Client and Server stack
- Server stack based on VPP and DPDK
- Client stack as a portable app
- Open sourced under APACHE 2.0 license on FD.io (https://wiki.fd.io/view/Cicn)
- Supported Platforms: Linux, macOS, Android, iOS, Window
(h)ICN Transport Services API

- How transport services are exposed to applications?
  - Legacy BSD socket API
    - socket(), send(), recv(), bind()...
    - Simple, clear and widely adopted...
    - ...yet limited and difficult to integrate in an evolving Internet

- Modern post-socket API
  - Intent based, event driven asynchronous interface
  - Simple and more expressive than BSD sockets
  - Easy to integrate with the variety of internet protocols and access technologies
(h)ICN with BSD socket API

- Take hICN namespaces based on IPv6 prefixes
- Based on socket interface extension for IPv6\(^1\)
- Domain: \texttt{AF_ICN/AF_HICN} address and protocol family
- Socket types: \texttt{SOCK_CONS} and \texttt{SOCK_PROD}
- Protocol types: \texttt{CONS\_REL/CONS\_UNREL} and \texttt{PROD\_REL/PROD\_UNREL}
// Create a new producer socket
int sockfd = hicn_socket(AF_HICN, SOCK_PROD, PROD_REL);

// Set the namespace for publishing contents
struct sockaddr_hicn producer_namespace {
  .sin_family = AF_HICN,
  .sin_prefix = 64 // Limit the namespace to a /64 namespace
};

hicn_pton(AF_HICN, "b001:a:b:c::", &producer_namespace.sin_addr);

// Bind a local face with forwarder and set the corresponding local route
hicn_bind(sockfd, &producer_namespace, sizeof(producer_namespace));

// Publish a content with name b001:a:b:c::1234
struct sockaddr_hicn content_name {
  .sin_family = AF_HICN,
};

hicn_pton(AF_HICN, "b001:a:b:c::abcd", &producer_namespace.sin_addr);

char *buf = "This string will be published in the hicn socket";
ssize_t n = hicn_sendto(sockfd, buf, BUF_SIZE, 0, &content_name, NULL);
Example: pull a content using a SOCK_CONS

```c
// Create a new consumer socket
int sockfd = hicn_socket(AF_HICN, SOCK_CONS, CONS_REL);

// Set the name of the content to pull
struct sockaddr_hicn content_name {
  .sin_family = AF_HICN,
  .sin_prefix = 64 // Limit the interest namespace to a /64 namespace
};
hicn_pton(AF_HICN, "b001:a:b:c::1234", &content_name.sin_addr);

// Bind a local face with forwarder and enforce the namespace size limit
hicn_bind(sockfd, &content_name, sizeof(content_name));

// Pull the content with name b001:a:b:c::1234 and store it inside buf
char buf[BUF_SIZE];
ssize_t n = hicn_recvfrom(sockfd, buf, BUF_SIZE, 0, &content_name, NULL);
```
hICN with Post-Socket API

- Generic API: application describes the required transport service, enforcing preferences and constraints
- hICN is a transport service provided by the system
- If application requirements can be satisfied with an hICN transport protocol, it is selected and used

I want to reliably retrieve these data, and I don’t want to use WiFi!

Ok, let me check if I can satisfy your needs!
Interface objects

- Pre-connection
  - Set of parameters and constraints of the application on path/protocol selection
  - It is used for electing a transport service able to satisfy application needs

- Connection
  - Transport protocol stack on which data can be received
  - Send interests to a remote endpoint (data name)

- Listener
  - Listen for incoming interests and publishes application data
  - Set up of a local endpoint (production namespace)
Example: publish data using post-socket APIs

```cpp
class HIcnProducer {
...
void onConnectionReceived(Error &error, Connection &connection, Name &name) {
    // Publish data upon consumer request reception
}
void run() {
    // Describe the desired transport
    transport_parameters_.setPreference(TransportProperty::TRANSPORT_MANIFEST, PreferenceLevel::REQUIRE);
    transport_parameters_.setPreference(InterfaceType::ETHERNET, PreferenceLevel::PROHIBIT);
    security_parameters_.add(SecurityProperty::IDENTITY, producer_identity_);
    security_parameters_.setPreference(SecurityProperty::SIGNATURE, PreferenceLevel::REQUIRE);
    // Describe the endpoint
    local_endpoint_.add(EndpointProperty::NAMESPACE, "b001:a:b:c::/64");
    // Use WiFi
    local_endpoint_.add(EndpointProperty::INTERFACE, Interfaces::LTE);
    // Create preconnection
    preconnection_ = new Preconnection(remote_endpoint_, transport_parameters_, security_parameters_);
    // Initiate a new listener starting from the preconnection.
    listener_ = preconnection_->listen();
    // Packetize a buffer of data, using manifests
    listener_->publish("b001:a:b:c::1234", buffer, size);
    // Wait for events on the socket
    listener_->processEvents(onConnectionReceived);
}
private:
    LocalEndpoint local_endpoint_;
    TransportParameters transport_parameters_;
    SecurityParameters security_parameters_;
    Identity producer_identity_;
    Preconnection *preconnection_;
    Listener *listener_;
};
```
Example: pull a content using post-socket API

class HIcnConsumer {
...
void onContentReceived(Error &error, Connection &connection, Name &name) {
    // Process received data..
}
void run() {
    // Describe the desired transport
    transport_parameters_.setPreference(TransportProperty::TRANSPORT_MANIFEST, PreferenceLevel::REQUIRE);
    transport_parameters_.setPreference(TransportProperty::RELIABLE_DATA_TRANSFER, PreferenceLevel::REQUIRE);
    transport_parameters_.setPreference(TransportProperty::DATA_ORIGIN_VERIFICATION, PreferenceLevel::REQUIRE);

    // Describe the endpoint
    remote_endpoint_.add(EndpointProperty::NAME, "b001:a:b:c::1234");
    // Use LTE
    remote_endpoint_.add(EndpointProperty::INTERFACE, Interfaces::LTE);
    // Create preconnection
    preconnection_ = new Preconnection(remote_endpoint_, transport_parameters_, security_parameters_);
    // Initiate a new connection starting from the preconnection.
    connection_ = preconnection_ ->initiate();
    // Asynchronously start the content download
    connection_ ->asyncRecv(buffer, size, onContentReceived);
    // Wait for events on the socket
    connection_ ->processEvents();
}
private:
    RemoteEndpoint remote_endpoint_;  
    TransportParameters transport_parameters_;  
    SecurityParameters security_parameters_;  
    std::string producer_public_key_;  
    Preconnection *preconnection_;  
    Connection *connection_;
Performance Evaluation of the server stack

- Cisco UCS Type-C
- Intel Xeon CPU E5-2695 v4
- 45 MB cache, 2.10 GHz, 18 cores
- 256 GB of RAM
- Intel 82599ES 10-Gbps NIC
- 10Gbps Cisco-Nexus 5k

- Best performance obtained by
  - NUMA nodes aware configuration of the hardware and software.
  - provide CPU affinity, reduce context switching to minimum
  - avoid PCI bus bottleneck

- segmentation and reassembly
- crypto operations
- no hardware offloading involved
Performance Evaluation

- TCP vs hICN REL transport service
- TCP newreno for VPP stack
- TCP Cubic for Linux stack

- hICN REL_CONS transport service w/ options:
  - Window based flow control (AIMD)
  - delay-based congestion control
  - w/ or w/o manifest
  - w/ or w/o ADU prefetching

- hash is SHA-256
- Signatures RSA-1024 or ECDSA-192

<table>
<thead>
<tr>
<th>Type of test</th>
<th>Average</th>
<th>99% CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>(h)ICN Asynchronous Publication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manifest RSA-1024</td>
<td>928Mbps</td>
<td>[919 936]</td>
</tr>
<tr>
<td>Packet-wise RSA-1024</td>
<td>290Mbps</td>
<td>[283 297]</td>
</tr>
<tr>
<td>Manifest ECDSA-192</td>
<td>531Mbps</td>
<td>[523 538]</td>
</tr>
<tr>
<td>Packet-wise ECDSA-192</td>
<td>28Mbps</td>
<td>[27 28]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type of test</th>
<th>Average</th>
<th>99% CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>(h)ICN Synchronous Publication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manifest RSA-1024</td>
<td>525Mbps</td>
<td>[518 532]</td>
</tr>
<tr>
<td>Packet-wise RSA-1024</td>
<td>26Mbps</td>
<td>[26 27]</td>
</tr>
<tr>
<td>Manifest ECDSA-192</td>
<td>530Mbps</td>
<td>[522 537]</td>
</tr>
<tr>
<td>Packet-wise ECDSA-192</td>
<td>28Mbps</td>
<td>[28 29]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type of test</th>
<th>Average</th>
<th>99% CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>(h)ICN Crypto Operations disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No signature</td>
<td>5.79Gbps</td>
<td>[5.77 5.81]</td>
</tr>
<tr>
<td>No signature, 2 transfers</td>
<td>5.80Gbps</td>
<td>[5.73 5.86]</td>
</tr>
<tr>
<td>No signature, 3 transfers</td>
<td>6.46Gbps</td>
<td>[6.44 6.49]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type of test</th>
<th>Average</th>
<th>99% CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP - Iperf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux TCP (w/ TSO)</td>
<td>9.19Gbps</td>
<td>[9.09 9.30]</td>
</tr>
<tr>
<td>Linux TCP (w/o TSO)</td>
<td>5.00Gbps</td>
<td>[4.88 5.12]</td>
</tr>
</tbody>
</table>
Hardware offloading and software optimization

- HW offloading
- Crypto
- Intel OAT (I/O acc. tech)
- arm acceleration
- Generic Segmentation Offload
- Generic Receive Offload
- Portability and features support tradeoffs.

- SW optimization such as VPP
- Based on batching
- Reduce memory access
- Improve core optimization
- to be used carefully for latency/goodput tradeoffs
Conclusions

- The development of API new is crucial to facilitate integration
- Backward compatible integration for existing applications
- Advanced applications such as AR/VR, IoT can benefit from novel API and transport services
- Deployment in software, user-space integration
- High-speed for server stack
- Highly portable for client stack
- Hardware offloading is an important topic to be supported as much as possible
Demos
ICN producer transport vs TCP transport

Video Server
Producer Socket
Video Segment

Pull Based
Stateless
One socket for n clients
E.g. live content broadcasting: clients are retrieving the same video chunk

Push Based
Stateful
n socket for n clients

Interest
WebRTC over hICN

- New sockets (reuse of TCP or UDP header)
- Mapping of application to network names (RTP and HTTP)
- Replacement of transport with hICN transport for some or all communications
- hICN-enriched IP network layer (names, forwarding strategies)
hICN with BSD socket API

**Common API (Consumer/Producer)**

- `int hicn_socket (int domain, int socket_type, int protocol)`
- `int hicn_bind(int sockfd, struct sockaddr *addr, socklen_t addrlen)`
- `ssize_t hicn_sendmsg(int sockfd, const struct msghdr *msg, int flags)`
- `ssize_t hicn_recvmsg(int sockfd, struct msghdr *msg, int flags)`
- `int hicn_setsockopt(int sockfd, int level, int __optname, const void *optval, socklen_t optlen)`

**Consumer specific API**

- `ssize_t hicn_recvfrom (int sockfd, void *buf, size_t n, int flags, struct sockaddr *addr, socklen_t *addr_len)`

**Producer specific API**

- `ssize_t hicn_sendto (int sockfd, const void buf, size_t n, int flags, const struct sockaddr *addr, socklen_t addr_len)`
## Crypto operations cost (no HW offloading)

<table>
<thead>
<tr>
<th></th>
<th>Vector of packets</th>
<th>Single packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Consumer: Signature verification</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSA-1024</td>
<td>52.2us</td>
<td>[51.5 52.9]</td>
</tr>
<tr>
<td>ECDSA-192</td>
<td>412us [406 417]</td>
<td>757us</td>
</tr>
<tr>
<td><strong>Producer: Signature computation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSA-1024</td>
<td>440us [437 443]</td>
<td>775us</td>
</tr>
<tr>
<td>ECDSA-192</td>
<td>380us [377 383]</td>
<td>701us</td>
</tr>
<tr>
<td><strong>SHA-256 hash computation on MTU packet</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5kB</td>
<td>9.44us [9.38 9.50]</td>
<td>28.62us [31.03 32.08]</td>
</tr>
<tr>
<td>9kB</td>
<td>31.55us [31.03 32.08]</td>
<td>68.26us [63.63 72.89]</td>
</tr>
</tbody>
</table>