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

Mauro Sardara
Cisco Systems Inc.
msardara@cisco.com

Luca Muscariello
Cisco Systems Inc.
lumuscar@cisco.com

Alberto Compagno
Cisco Systems Inc.
acompagn@cisco.com

ABSTRACT

In this paper we present the design of a transport layer and socket API that can be used in several ICN architectures such as NDN, CCN and hICN. The current design makes it possible to expose an API that is simple to insert in current applications and easy to use to develop novel ones. The proliferation of connected applications for very different use cases and services with wide spectrum of requirements suggests that several transport services will coexist in the Internet. This is just about to happen with QUIC, MPTCP, LEDBAT as the most notable ones but is expected to grow and diversify with the advent of applications for 5G, IoT, MEC with heterogeneous connectivity. The advantages of ICN have to be measurable from the application, end-services and in the network, with relevant key performance indicators. We have implemented an high speed transport stack with most of the designed features that we present in this paper with extensive experiments and benchmarks to show the scalability of the current systems in different use cases.

CCS CONCEPTS

• **Networks** → **Programming interfaces**; Network experimentation;

KEYWORDS

ICN, Transport Services, Socket API

ACM Reference Format:
Mauro Sardara, Luca Muscariello, and Alberto Compagno. 2018. A Transport Layer and Socket API for (h)ICN: Design, Implementation and Performance Analysis . In *ICN '18: 5th ACM Conference on Information-Centric Networking (ICN '18), September 21–23, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3267955.3267972

1 INTRODUCTION

Information-Centric Networking (ICN) is a network paradigm that enables location independent and connectionless communications. In this paper, by using ICN, we refer to the NDN/CCN ([33], [34]) architectures but also to the more recent Hybrid ICN [36] that implements NDN/CCN into IPv6. We use the symbol (h)ICN to indicate all of these architectures.

There has been a considerable amount of work on developing novel applications on top of (h)ICN. However, there has been little effort into developing an intermediate stack, namely the session and transport layers as well as an API that is feature reach and simple at the same time. The network stack as implemented in current operating systems, exposes Internet sockets (INET) using the BSD socket API for all *NIX OS or Windows Socket API for Windows OS. They are all very similar and application developers are used to deal with this kind of API to develop networked applications. There has been a remarkable amount of work to develop new transport services in the Internet in order to meet different applications' requirements, such as LEDBAT, QUIC ([25]), MPTCP. In the future, it is likely that novel transport services will be developed to respond to upcoming needs and to adapt to different environments, namely mobile networks, delay tolerant networks, multi-homed access use cases, just to cite some examples. The advent of new transport services are also expected to proliferate to serve the development of new applications for the next decades. In this case, developing and defining an API that can be inserted in applications in a simple manner, with extend-able features, is a mandatory requirement for the success of new transport services. The current IETF WG TAPS [37] is the most notable effort in this direction and also the work done in the NEAT project [9].

(h)ICN is not an exception, and its proliferation also depends on the definition and implementation of simple and an easy to use session and transport layer. Currently, (h)ICN lacks of this definition as well as the implementation of a socket API that can facilitate its insertion in existing applications as well as the development of new ones. In this paper, we describe one such layer that is based on the consumer/producer communication abstraction model and we show how it gracefully fits into a comprehensive middle-ware between applications and the (h)ICN layer. Moreover, we discuss several transport services and we provide an implementation for them. We compare our implementation with the corresponding services available in TCP and UDP and we show that we achieve comparable performace in terms of goodput.

The paper is organized as follows. Section 2 gives an overview of the ICN namespaces and their relation with the transport and network layers. Section 3 describes the overall architecture with assumptions and constraints; Section 4 presents the transport layer as well as the socket API. Section 5 describes the details of an high speed implementation of the proposed transport stack that is benchmarked and analyzed in Section 6 and 7. Section 8 reports related work on the topic while Section 9 concludes the work.

ICN '18, September 21–23, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-5959-7/18/09…$15.00
https://doi.org/10.1145/3267955.3267972

2 NAMESPACES IN (H)ICN

In NDN/CCN, it is often assumed that application names are the same names used by the routing plane in the form of routable prefixes. In this section, we argue that application names and network names should not coincide in order to provide a greater flexibility to organize data at application layer, while maintaining routing scalability, network domains separation and better supporting mobility.

Today applications manage data using different kind of namespaces that are used for the purpose and functioning of the specific application itself. For instance a web server typically makes data available using a URL which embeds the hostname of the server and defines a locator to determine where the data can be retrieved. The data itself inside the server is then organized using a namespace that is inherited by the local file systems. Domain names are managed by an authority that maintains a registry of allocated names to registrants.

Other collections of data are organized using URN for objects (RFC 3061 [30]), ISSN (RFC 3043 [31]), DOI (ISO 26324 [18]) and many mores. It is convenient to organize data using a standardized namespace as it allows simple migration of the data, interoperability and integration of different applications among themselves. However, this is not always the case, and most of the time each application develops namespaces autonomously.

The same level of flexibility and customization can be hardly achieved with network names, whose definition usually follows some strict rules and their allocation is handled by the network administrator (who is usually unaware of the applications running at the end-host devices). Additionally, using the same names at network and application layers also issues severe concerns to the routing system. In this case names (prefixes usually) used by applications would be distributed in the routing systems in order to set the forwarding plane. The variety of namespaces used by different applications would make complicated to exploit aggregation to maintain FIB small in the routers, thus affecting routing scalability. Lastly, the data producer would have to attest the ownership of the prefix to the routing system and the different network domains exchanges prefixes to assure reachability.

In order to avoid the issues raised above, an efficient solution is multiplex/demultiplex application names into network names. We consider the case of hICN, in which routable name prefixes are IPv6 prefixes and follow the usual rules on IPv6 routing including prefix attestation. The way name prefixes are assigned to a data producer is similar to what would happen in LISP ([16]) when end host identifiers (EID) are assigned. However in LISP an EID is not routable and goes into a mapping system for translation in a routable locator. EID and routable addresses both belong to an entity that acquires them from a resource pool managed by an external entity. In this respect operations of name mapping (between application and network layers), name translations (mobility, roaming) will need to be further developed and implemented to make the whole system scale and inter-operate in different environments.

The mapping between network and application namespaces has a direct influence on the ICN data aggregation, routing scalability and data transmission. Application data moves from one location to another by means of data segmentation that has to fit into the minimum maximum transfer unit across the link traversed from the source to the destination(s). If the data does not fit into the MTU of a link it is either discarded or reassembled before retransmission. Instead, the network namespace is data agnostic.

In NDN/CCN and hICN data packets are directly indexed using a hierarchical name. The hierarchy allows to organize application level data inside name prefixes, but also to better scale routing by name (aggregation) and to define lower level indexes as segment identifiers. One of the compelling usage of (h)ICN architectures is to reduce redundant transmissions because multiple requests for the same data can be satisfied by a single transmission. Immutability of the data that is associated to a given name is a strong requirement at least for the lifetime of a transport session, otherwise reassembly of the data at consumers would not complete successfully.

In NDN/CCN, segmentation suffixes including additional metadata such as versioning are included as additional TLVs as name components. In hICN [36] segmentation information is included in the L4 header: name suffix, lifetime etc. Fragmentation poses issues, as in IP, as immutability is not preserved and data sharing is suboptimal (some solutions are proposed in [35], [20]). The risk is that, as in IP today, fragmentation makes multi-path transport difficult to optimize, as the characteristics of a network path can influence the way data is segmented in the namespace. For (h)ICN this can be a significant limitation. Hop-by-hop reassembly of fragments seems the best solution at a non negligible cost. Moreover, end-points can charge the network with additional cost for segmentation/reassembly operations.

MTU path discovery protocols can eliminate this cost if used by the end-points when possible. The design of this kind of protocols needs additional work for (h)ICN as a key requirement is to preserve location independence and multi-path communication efficient.

In summary, data namespaces in (h)ICN have an impact on the full stack: from the application, through transport and down to the single data packet in case of segmentation or fragmentation. Each layer has different objectives and constraints. Each layer manages resources of different capacities implying trade offs while multiplexing/demultiplexing name data from one layer to another. Some of these trade offs have been discussed in this section, others will be described in the paper in detail concerning specific transport layer functions. In this paper we consider the transport layer that has adjacencies with application and network layers and has a key role in the overall architecture.

3 ARCHITECTURE

In this section, we present the network layer requirements we identified to design our socket API and transport Services. Hybrid ICN (hICN) [36] is an examples of (h)ICN implementations that brings information-networking functionalities into IPv6, without creating overlays with a new packet format as an additional encapsulation. hICN reflects all the key ICN properties implemented on the CCN/NDN architectures, such as: named data, dynamic named-based forwarding, data-centric security, receiver-driven connectionless transport. The major design difference between hICN and NDN/CCN resides on the data names which have a fixed size and are mapped into IP6 addresses. Instead, interest and data packets

forwarding follows the same rules defined in NDN/CCN. Additional details can be found in [36].

The transport layer that we present in this paper sits on top of a network layer that provides the semantics of the NDN/CCN architecture. The services provided by the network layer protocols to the upper layers are characterized by request/reply semantics, meaning that data is transmitted only upon reception of the corresponding request. Each data is transmitted in a Data Packet that is unambiguously identified by a hierarchical name. Such hierarchical name is used at the network and transport layer to realize location independent name-based forwarding, as well as multiplexing and demultiplexing of communication flows, data segmentation and reassembly. We assume that the network layer does not perform fragmentation, i.e. the L3 PDU fits into link-layer maximum transfer unit (MTU). For this reason, we assume as well that the transport layer relies on a path MTU discovery protocol (PMTUd). The more general problem of designing a PMTUd mechanism that can be effective for multi-path and multi-homed end-points is an important problem but it is out of scope for this paper, where simple naive approaches are adopted.

Moreover, we assume that application level data is associated to a namespace that applications use to organize, uniquely identify and securely exchanged each data. We believe, and encourage, that application level namespaces and names differs from the (h)ICN network layer prefixes and names. This allows ICN architectures with a finite pool of network names (such as hICN) to deal with the several orders of magnitude bigger application level data that can be permanently addressed in an end-host. Additionally, decoupling application-layer names from their network-layer counterparts provides several benefits in term of packet processing and security [21]. Translation between the application level names to the corresponding network layer names is done in the transport layer. How the translation between network and application name is done is left for future work. In principle, this is not different to the current application layer in today's Internet which relies on session and transport layers to conceal details of the networking semantics. Nevertheless this is not entirely true in practice as many APIs do not properly respect such basic principle. For instance `gethostbyname()` is one of those infringements.

For a given name prefix, used to exchange an application data unit, the name hierarchy is also used in the transport layer for segmentation and reassembly operations using name suffixes as part of the network namespace to unambiguously identify Data Packets in the network and in a communication flows.

4 TRANSPORT SERVICES

In this section we present in details how communication sessions are created, mapped into network namespaces and prefixes, and for how long. We describe also how resources are allocated at the end-points in order to serve the communications needs of the applications. We highlight that our socket API and transport services run on every (h)ICN architecture, e.g., NDN/CCN as well as hICN.

4.1 End-points description

We identify two kinds of communication sockets each with a specific API: the producer and consumer sockets. These socket types are designed to exchange data in a multi-point to multi-point manner. The producer-consumer model is a well-known design concept for multi-process synchronization where a shared memory is used to let multiple consumers to retrieve the data that is made available by producer processes into the same memory. In (h)ICN we have the same concept that is applied to a network where memories are distributed across the communication path. The first memory in the path is the production buffer of the producer end-point that forges Data Packets and copies them into a shared memory isolated into a namespace. Consumer sockets can retrieve data from such memory by using the (h)ICN network layer. The model just described is an inter-process communication example (IPC) that requires data to cross a communication network by using a transport protocol.

The way consumers and producers synchronize depends on application requirements and the transport layer exposes a variety of services: stream/datagram, reliable/unreliable, with or without latency budgets etc. Independently of the specific requirements of the applications, producer sockets always perform data segmentation from the upper layer into Data Packets, as well as compute digital signatures on the packet security envelop. This envelop can also be computed across a group of packets, by including a cryptographic hash of each packet into the transport manifest, and eventually signing only such manifest. This is a socket option that can bring significant performance improvement.

The consumer socket, on the other end, always performs reassembly of Data Packets, hash integrity verification and signature verification. The usual assumption is that the producer socket uses an authentic-able identity while using namespaces that it has been assigned. The end-point must be able to manage the mapping of her identity and the allocated namespace by issuing digital certificates about the mapping. The consumer end-point must retrieve the associated certificate to perform the basic operations. It is out of scope for this paper how to design and implement a scalable system to perform such certificate operations.

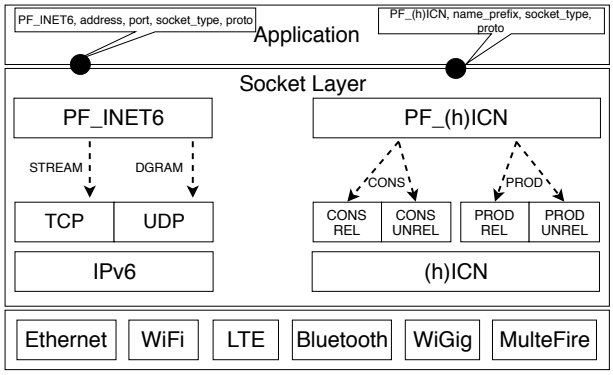


Figure 1: Description of the network stack and socket API.

4.2 Network namespaces

The session layer takes care of sharing local resources among all communication sessions such as consumer and producer sockets. Any time an application wants to open a socket, the session layer allocates space from a memory pool and securely isolate it within the

valid namespace. Both producer/consumer sockets are successfully instantiated to match against a valid prefix that must be available in the local FIB. Communication flows are multiplexed into the network layer using the namespace itself and do not require the usage of L4 ports as in TCP/IP. The new session is multiplexed to the network stack by registering the new session as a unidirectional application face. FIB entries have to be configured accordingly in case the new application face interconnects a producer or a consumer socket. This kind of faces can be seen as shared memories bound to a name prefix with read/write permissions.

The session is instantiated by passing socket types, options, parameters and is released to the resource pool at closure, including the used namespace. By consequence it implies that the certificate used by a producer end-point is no longer valid and revocation of the certificate must be enforced.

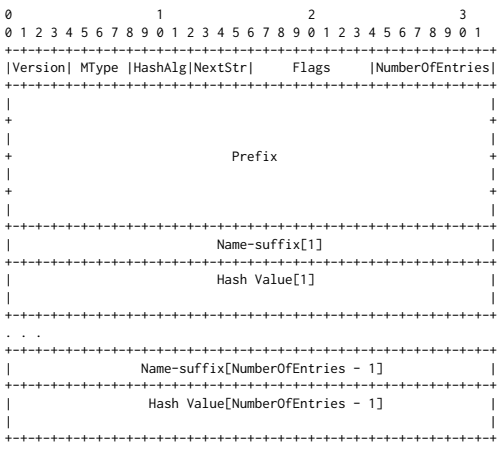


Figure 2: Manifest encoding: compact encoding.

4.3 Socket API

The system calls of the socket API are based on the socket interface extensions for IPv6 [23] and are shown in Figure 3.

Applications are supposed to call the `socket()` system call to create descriptors representing a communication end-point, i.e., a consumer or a producer. The domain `AF_ICN` defines the address family and the socket type defines the communication semantics: `SOCK_CONS` will create a consumer socket, `SOCK_PROD` will create a producer socket. The consumer socket takes care of data reception, while the producer socket of data transmission. The protocol parameter specifies the protocol used for the data retrieval and the data production: `CONS_REL`/`CONS_UNREL` for reliable/unreliable content retrieval and `PROD_REL`/`PROD_UNREL` for reliable/unreliable data production. Section 5 clarifies the meaning of such parameter.

Both sockets bind to a socket address, who is initialized by specifying the address family `AF_ICN` and the name prefix (`struct sockaddr_icn`). The name prefix enforces the namespace in which a producer socket is allowed to publish data and a consumer socket is allowed to request data.

The `bind()` system call takes care of setting up a local face to the forwarder, which in the case of the producer also sets a FIB entry *(name_prefix, socket_id)*. The `recvmsg()` and the `recvfrom()`

```
Common API
int socket (int domain, int socket_type, int protocol);

int bind(int sockfd, struct sockaddr *addr,
         socklen_t addrlen);

ssize_t sendmsg(int sockfd, const struct msghdr *msg,
                int flags);

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

```
Consumer specific API
ssize_t recvfrom(int sockfd, void *buf, size_t len,
                 int flags, struct sockaddr *src_addr,
                 socklen_t *addrlen);
```

```
Producer specific API
ssize_t sendto(int sockfd, const void *buf, size_t len,
               int flags, const struct sockaddr *dest_addr,
               socklen_t addrlen);
```

Figure 3: System calls for socket initialization and binding to a socket address, as well as for data reception and transmission.

system calls are used by a consumer for retrieving a content, while `sendmsg()` and `sendto()` are used by a producer for publishing data and making it available for the consumers. Notice that the consumer socket can just use the `recvmsg()` and `recvfrom()` as they are the two system calls capable of specifying the name of the content to be retrieved, and the producer socket can use only the `sendmsg()` and the `sendto()` for publishing data under a certain name. For both cases, the name used for pulling/publishing data has to be in the range previously specified in the `bind()` system call. The `setsockopt()` allows application to tune the available socket options: its semantic with respect to the current sockets API does not change. Some of the available options will be described in Section 5.

The consumer socket can use the `sendmsg()` for sending arbitrary interests in a datagram fashion, while the producer socket can call `recvmsg()` for receiving and processing requests from the consumers. In particular, an application can use the `sendmsg()` system call with a consumer socket to announce to a specific server (or to a set of servers) that it is publishing data under a certain name. In this way the remote server can retrieve the piece of content by triggering a reverse pull induced by this signalization. The reverse pull can be realized with the following steps: *(1)* the client endpoint prepares a transport manifest with the information for pulling the content; *(2)* by using `sendmsg()`, it sends an interest containing the manifest to the server endpoint; the interest name is a special prefix used by the server for receiving these control messages; *(3)* the server receives the interest through the `recvmsg()` and gets the manifest from it; *(4)* the manifest can be used by the server endpoint for triggering the reverse pull, by exploiting the `recvfrom()`.

We highlight that our socket API design is general and it does not restrain developers to develop a variety of different applications patterns that differs from the content distribution applications (in which a set of clients that retrieve content from one or multiple replicated servers). For example: collaboration applications in which a number of users share data in a real-time fashion with

each other can be implemented using one consumer socket and one producer socket per user. The design of a specific transport protocol, and its selection through the socket API, will guarantee the communication-delay constraints required by the real-time nature of the communication. And again, multi-write distributed databases in which a number of entities write simultaneously to many replica is an example of how to exploit the reverse pull in order to write data to the distributed database. More in general, the reverse pull mechanism plays a fundamental role in each application patterns that do not directly fit with the request-reply ICN communication and require a push mechanism, e.g., REST-based and pub-sub applications.

5 IMPLEMENTATION

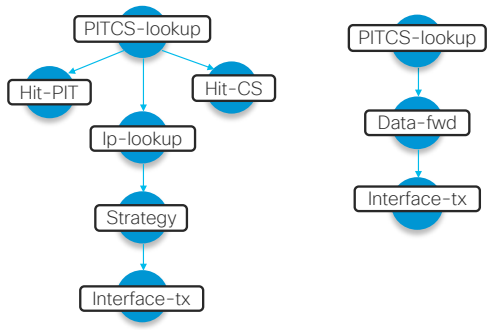
The (h)ICN transport service has been implemented in most of the components in a C++ userspace library. It can be used to connect to the CCNx 1.0 metis forwarder or the VPP [27] based forwarder, as well as to the hICN forwarder. It has been also used with the NDN forwarder daemon (NFD). The source code of two CCNx 1.0 forwarders is available in the CICN open source project [17], while the implementation of the hICN forwarder is not yet available as open source project. In the present paper experiments reported below are made by using the hICN forwarder as it provides best performance compared to the others. To connect an application with the underlying forwarder, we developed a *Forwarder Connector* which exposes a uniform interface for sending and receiving packets to/from the network stack. The pipeline is reported in Fig. 5 and 6. A more detailed description is presented in Section 5.2. Below we report a short introduction to VPP which constitutes the packet processor that allows to obtain scalable performance in software.

5.1 VPP: vector packet processor

VPP is a high-speed software based packet processor that provides advanced data plane function in commercial off-the-shelf (COTS) hardware. VPP design has two major pillars: *(1)* completely bypassing the kernel in order to avoid the overhead associated with kernel-level system calls, *(2)* maximize the number of instructions per clock cycle (IPC). Kernel-bypass is achieved through low-level building blocks, such as Netmap [28] and DPDK [42]. These mechanisms provide Direct Memory Access (DMA) to the memory region used by the NICs, therefore avoiding the need of the kernel to interact with the underlying hardware. Maximization of IPC is achieved carefully designing the VPP implementation and exploiting data pre-fetching, multi-loop, function flattening and direct cache access. In the following we briefly present the VPP architecture and the design of our hICN plugin for VPP.

The VPP code is organized in a set of nodes, each implementing specific functions (e.g., ip4 forwarding or fib lookup). There are three types of nodes in VPP: namely *internal*, *process* and *input*. Internal and input nodes form a *forwarding graph* determine the processing paths each packet follows during it processing. Input nodes interact directly with the NICs, reading packets from the rx ring buffer and injecting them in the forwarding graph. Internal nodes implement packet processing functions (e.g., packet forwarding, address rewrite, fib lookup) as well as they move packets to the tx ring buffer in order to let the hardware to forward a packet. VPP processes packets in a *vectorized* fashion: input nodes create a vector of packets, which is moved from internal node to internal node by the graph node dispatcher. Thus, every node executes its processing function on the entire vector. This design allows to minimize cache misses, to reduce the overhead of selecting the next node in the graph, as well as to simplify pre-fetching [26]. Beside supporting DPDK and Netmap compatible NICs, VPP provides other several additional types of interfaces. Among all, memory-based interfaces (in short *memif*) are designed to allow applications to send packets to a local VPP forwarder. Memif interfaces are designed to be efficient in order to forward several gigabits per second. Connectivity between an application and a VPP forwarder is provided by a pair of memif interfaces; one managed by VPP and the other by the application. Such pair of interfaces are virtually connected through a bidirectional link, thus letting packets to flow from one interface to the other. The virtual link is implemented using two circular buffers (one per each direction of the link) stored in a portion of virtual memory shared between the VPP forwarder and the application.

5.1.1 hICN plugin. The hICN ([36]) forwarder is implemented as a VPP plugin which adds six new nodes in the forwarding graph, enabling two new forwarding sub-graphs: the interest and the data forwarding pipeline and shown in Fig. 4.



(a) Interest sub-graph **(b) Data sub-graph**

Figure 4: hICN plugin for VPP

We exploit the VPP memif interface to implement the application faces, i.e., hICN faces that forward traffic to/from local application. In particular, we designed two different APIs that the transport layer can use to connect to a consumer face and a producer face. Both APIs instantiate a new memif interface, exposing the memif shared memory to the transport layer. Additionally, the producer face API also creates a new entry in the FIB in order to forward Interest Packets to the producer socket.

5.2 Forwarder Connector

The forwarder connector is the software module in our transport services that interacts with the underlying forwarders. We designed two specific connectors, one for our hICN-VPP forwarder, and one for the client forwarder. Both connectors expose the same northbound interface to the consumer socket and the producer socket. In particular, the interface is composed of four APIs that the transport layer can use to create a consumer/producer connector and to

send/receive packets: `connectConsumer()`, `connectProducer()`, `sendPacket ()`, `recvPacket()`. The purpose of the first two API two is to create a (or connect to an existing) consumer or producer application face in the hICN forwarder. The `connectProducer ()` also takes care of setting a new FIB entry in order to allow the forwarder to send Interest Packets to the producer socket itself. The `sendPacket()` and `recvPacket()` are intended to send/receive packets to/from the hICN forwarder once it is connected.

5.3 Consumer Socket

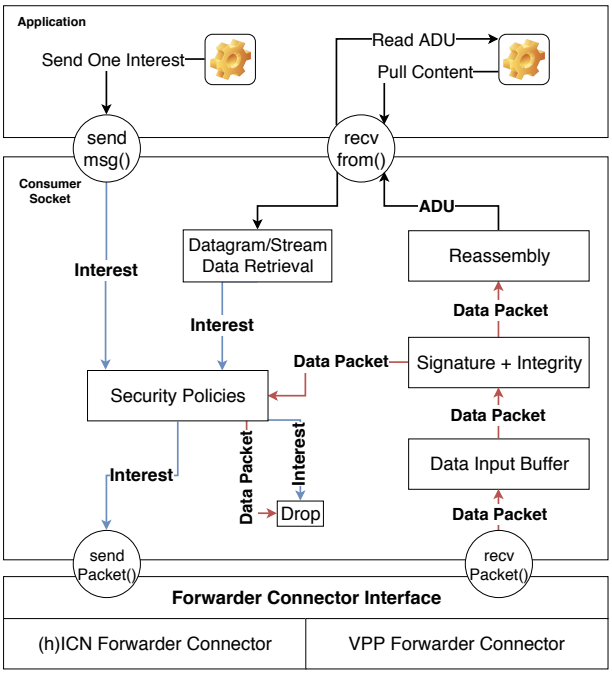


Figure 5: Consumer socket processing pipeline

Applications can instantiate a consumer socket for *(1)* retrieving a specific content with a certain name (e.g. Http Client) or *(2)* directly sending an interest to retrieve one specific Data Packet (Ping, RTP Client).

Figure 5 shows the internal processing pipeline of the socket and points out the path followed by Interest and Data Packets during the content retrieval. The content download is triggered by the call to the `recvfrom` function: the application specifies the name of the content to pull and some download options, such as the actions to perform in case of signature verification failure. After the initial setup phase, the control is taken by the *Data Retrieval protocol*, specified at the socket creation with the `socket` system call in the protocol parameter (`CONS_REL`, `CONS_UNREL`). The protocol then starts generating Interest packets for retrieving the content requested by the application. Before being forwarded to the forwarder connector, the *Security Policies* block applies the policies specified by the application as socket options: this includes for instance signing the Interest or verifying that the Interest matches all the application and socket constraints (e.g. the interest name belongs to the prefix specified in the `bind` system call). If all the requirements are satisfied, the Interest is passed to the forwarder connector that takes care of delivering it to the next hICN node.

In the same manner, the forwarder connector takes care of delivering the Data Packets coming from the hICN forwarder to the application. As soon as the Data Packet is received, it is stored it in the *Input Buffer* and then passed to the *Verification Routine* that performs the data-origin authentication check. As mentioned in Section 4, this operation can be performed either by using a transport manifest or verifying the signature of each Data Packet, depending on how the producer decided to sign the content. If this check fails, it means either the Data Packet has been corrupted or has been produced by a malicious producer: in this case the security policies set by the application will define if either dropping the Data Packet and sending a new Interest, using it anyway or aborting the download. If the data-origin authentication check succeeds, the Data Packet is used for reassembling the application content. When all the Data Packets are successfully reassembled and verified, the consumer socket returns the content to the application.

Since the operation of verifying the data origin authentication is expensive, as we show in Section 6, the flow control operations performed by the *Data Retrieval Protocol* must be decoupled from the operations performed by the *Verification Routine*, otherwise the crypto operations would limit the rate of the flow control protocol.

5.4 Producer Socket

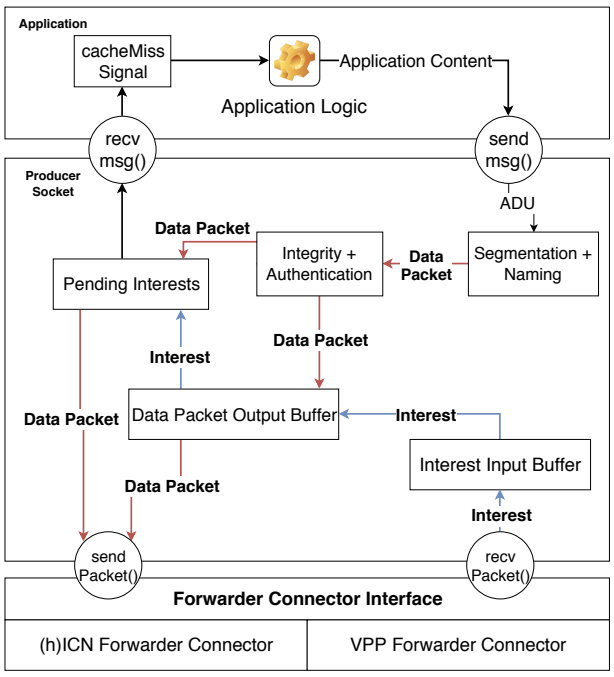


Figure 6: Producer socket processing pipeline

The producer socket is responsible for producing Data Packets and making them available for the consumers. Data Packets can be published in two different fashions: *(1)* asynchronous production, if the application publishes data without receiving any Interest (e.g. file repository) or *(2)* synchronous production, if the application

publishes data upon the reception of one Interest (e.g. real time applications). We describe first the case of asynchronous content publication and then we will see how applications can use the producer socket for dealing with dynamic requests.

Application can publish data by calling the `sendto` API with the ADU, name, crypto suite for signing the Data Packets and some optional publication options, such as whether using a transport manifest or not.

The first operation performed by the producer socket is the *data segmentation and naming*: the application content is segmented into MTU-size Data Packets which are named by combining the prefix received by the application and a suffix, for unambiguously identifying the packets in the network. Each Data Packet is then processed by the *Signature routine*: depending on the publication options provided by the application, the routine decides if signing every packet or groups of packets by using a transport manifest ([11]). Generally speaking, applications producing a large amount of data such as HTTP servers generally should opt to transport manifest, since it avoids to compute the signature of every Data Packet. Other applications, such as RTP clients and servers, could opt for a per-packet-signature approach, but this are choices made by the application developer. We will analyze the cost of signing packets in section 6. As soon as the Data Packet is authenticated, it is stored in the *Data Packet Output Buffer*, which function is to store the Data Packets for matching incoming Interests. Since the size of this buffer can significantly increase, it likely cannot fit in the main memory: for this reason we designed it as a two layer cache, where the L1 cache has a limited size and it is stored in the main memory, while the L2 cache can be really large and is saved on permanent storage. The replacement policy of the L1 cache is a socket option defined by the application itself, and can be FIFO, LRU or LFU. If the application creates the socket selecting the `PROD_UNREL` protocol, the L2 cache is disabled and whenever a Data Packet is removed from the L1 cache is lost. This case is typical of live and real time application, where the *Data Packet Output Buffer* needs to always store new Packets and discard the old ones. On the other hand, applications like a File Repository needs their data to be always available: in this case the L2 cache is required and the `PROD_REL` protocol has to be used.

When an Interest Packet is received through the forwarder connector's `recvPacket` API, the Producer Socket puts it in an input buffer and then tries to perform a lookup for it in the *Data Packet Output Buffer*: if a corresponding Data Packet is found, the socket replies directly by forwarding it to the connector through the `sendPacket` call. In this way the application does not perform any computation for the incoming request, as they are satisfied directly by the transport layer (Section 7). If the output buffer does not contain any matching Data for the interest, the latter is registered in a *Pending Interests* table and the socket signals to the application that the cache miss happened: the signalization is done through the `recvMsg` system call, by passing to the application the name of the Interest and other information (such as a possible interest Payload). The application can then react to the cache miss by publishing the content required by the interest, with the production procedure described above. At the moment of storing the Data Packet into the *Data Packet Output Buffer*, one further lookup is done on the Pending Interests table for checking if it contains a pending request for the data that is going to be published: if the lookup succeed, the data packet is forwarded directly to the underlying forwarder connector. This whole procedure allows the socket and the application to jointly perform the dynamic production of content.

6 PERFORMANCE ANALYSIS

In the following sections, we report the performance evaluation of our transport layer. We show the performance we achieve using the per-packet signature and the manifest approach, reporting the average goodput as well as the communication latency. Moreover, we compare the performance of our implementation with the TCP implementation in the Linux kernel and the TCP stack implemented in VPP. Our goal is to show the state of the art performance of our transport layer using as a reference the today's transport layer.

6.1 Experimental settings

The experiments presented in this section are performed using the vICN framework [40] the following setup. Two Linux containers [2] deployed on a Cisco UCS Type-C server with an Intel(R) Xeon(R) CPU E5-2695 v4 and 256 GB of RAM. The two containers manage an Intel 82599ES 10-Gbps NIC and are connected together through a 10Gbps Cisco-Nexus 5k. The two NICS are installed in the PCI of two different NUMA nodes, in order to distribute the workload on 2 different CPUs and improve the memory usage. Processes running inside LXC, such as VPP and applications, are run with CPU affinity to cores that are installed into the NUMA node belonging to the NIC VPP is using. This allows to achieve optimal performance in terms of memory access latency.

The hICN traffic is forwarded using the VPP forwarder augmented with the hICN plugin described in Section 5, and running inside both containers. The goal of this experiment is to measure the performance of our hICN transport, by forwarding traffic between the two containers. To this end, we wrote a simple application that sits on top of our transport library and provides statistic regarding the transport itself, just like iperf [6] for TCP/UDP. To compare hICN transport layer performance TCP, we use default TCP in the Linux kernel[1] (TCP Cubic) and in VPP (TCP New Reno). We use the iperf3 tool [6] to test the performance of TCP[6]. The reliable transport service used in this set of experiments is based on [14] which implements delay based flow and congestion control.

In the following, we report a summary of an extensive experimentation campaign to benchmark the performance of our transport services. In particular, we first study the computational cost of the crypto operations required to compute and verify signatures, then we evaluate their impact on transport services. To this end, we consider different scenarios: *(1)* distribution of static content that the producer publishes asynchronously, namely *Asynchronous publication (2)* communications in which the content is requests as soon as it is available in the application, namely *Synchronous publication*. In *(1)* the producer segments, names and signs every requested content a priory, i.e., upon receipt of each interest, the corresponding Data Packet is already available in the *Data Packet Output Buffer*. In *(2)* the producer segments, names and signs the application content upon the receiving of the first interest for it. In all our experiments, the producer publishes (and the consumer

[1] Kernel version 4.4.0104

retrieves) a content with size 200MB. Moreover, we use RSA and ECDSA as signing algorithm, choosing a key size of 1024 and 192bits respectively [2]. Finally, we used SHA256 as cryptographic hash algorithm. The crypto library used for the crypto operations is openssl 1.0.2o [5]. All statistics reported below are obtained from at least 30 independent experiments and mean values are reported with a t-student confidence interval with 99% significance.

6.2 Results

Table 1 shows the goodput of our transport. In both the Synchronous and Asynchronous publication scenarios, the crypto operations have a considerable impact on the goodput performance. Obviously, the Asynchronous publication offers better performance than the Synchronous publication as the signature is computed offline and it has no impact on the goodput. If we compare the per-packet verification with the manifest approach, the latter is able to provide a higher throughput. This is because, signing only manifests reduces the number of signature verification that a consumer has to perform (about 93% reduction). In this case, the throughput is limited by the per-packet hash computation (about 140000 hash calculation - 1.3s), and the cost for the signature verification (about 4000): using RSA-1024 the latter is around 0.208s, whereas using ECDSA-192 is about 1.6s (cfr. Table 2). We want to stress that we are not exploiting any dedicated hardware acceleration to speedup the crypto operations or the segmentation operations. Following this approach will significantly improve the performance.

In any case, the crypto operations put a boundary on the goodput of the application: with our software implementation, in case of verification with manifest, RSA-1024 takes around 1.508s for verifying 140000 Packets of 1500 bytes, while ECDSA-192 takes 2.9s. This turns in a maximum goodput of approximately 1.1 Gbps for RSA-1024 and 580 Mbps for ECDSA-192.

The approach without manifests requires producers and consumers to sign/verify every data packet: in the case of asynchronous publication, verifying with RSA-1024 allows to reach an acceptable goodput of 290 Mbps; on the other hand, verifying with packet-wise ECDSA-192 drops the throughput to 28 Mbps. Synchronous publication with per-packet signature drops the performance both with RSA-1024 and ECDSA-192 to 28 and 26 Mbps respectively. These goodputs are still acceptable for real time applications, where the rate is low and using manifest could increase the latency.

At last, we compare our performance with the one obtained by TCP in the same condition. We highlight that the hICN transport services should not be directly compared with TCP, as the latter does not provide neither integrity nor data-origin authentication. Therefore, to have an idea of the gap between our prototype and TCP, we disable integrity and data-origin authentication and we report the goodput our hICN transport achieves. In this case, the kernel implementation of TCP, with hardware accelerations disabled, achieves a goodput equal to 5Gbps while our transport stops at 2.45Gbps. We stress that, at the time of writing, none of the existing hardware acceleration for TCP (e.g., TSO, GSO, GRO) is compatible with our hICN transport layer. Therefore, we believe it would be unfair to exploit hardware acceleration for TCP. Still, we report the performance of TCP exploiting the hardware acceleration for the sake of completeness.

If we consider the two cases of *(1)* two flows in parallel and *(2)* three flows in parallel, the sum of the goodput of each flow reaches respectively 3.16 and 3.69 Gbps, with a fair share of the available bandwidth (Jain index ≈ 1). The sum of the three flows goodput is an indicator of the maximum performance achievable by our hICN plugin.

Type of test	Average	99% CI
hICN Asynchronous Publication		
Manifest RSA-1024	928Mbps	[919 936]
Packet-wise RSA-1024	290Mbps	[283 297]
Manifest ECDSA-192	531Mbps	[523 538]
Packet-wise ECDSA-192	28Mbps	[27 28]
hICN Synchronous Publication		
Manifest RSA-1024	525Mbps	[518 532]
Packet-wise RSA-1024	26Mbps	[26 27]
Manifest ECDSA-192	530Mbps	[522 537]
Packet-wise ECDSA-192	28Mbps	[28 29]
hICN Crypto Operations disabled		
No signature	2.45Gbps	[2.43 2.46]
No signature, 2 transfers	3.16Gbps	[3.13 3.19] Jain=0.99
No signature, 3 transfers	3.69Gbps	[3.43 3.95] Jain=0.98
TCP - Iperf		
Linux TCP (w/ TSO)	9.19Gbps	[9.09 9.30]
Linux TCP (w/o TSO)	5.00Gbps	[4.88 5.12]
VPP TCP stack	9.24Gbps	[9.22 9.26]

Table 1: Average goodput of (a) hICN - Asynchronous publication (b) hICN - Synchronous publication (c) hICN - Asynchronous publication w/o crypto operations (d) TCP - Iperf3.

Table 2 shows the time required to compute the crypto operations. We report the measurements considering two different cases: *(1)* crypto-operations performed inside loop on a vector of packets and *(2)* crypto-operations performed per packet with a frequency of 1s. In the first case, the cost of the crypto operation is lower than in the second case. The explanation for this behavior is the following: performing crypto-operations on a vector of packets requires the CPU to spend more time to complete this operations and therefore it is less likely that the kernel runs a different process on the same CPU. Therefore, L1 and L2 cache locality is improved and the number of cache miss is drastically reduced improving the performance on computing the crypto operations. On the other hand, processing one packet per second does not allow to exploit the CPU optimization described before, and this turns in a performance worsening. We also show how using jumbo frame helps in reducing the cost of computing crypto operation. The adoption of jumbo frames reduces the number of packets generated during the segmentation operation, thus lower the computational cost per byte of the hash calculation (from 0.006us with MTU=1.5kB, to 0.003us with MTU=9KB).

Table 3 reports the impact of the signature computation and verification on the end to end latency. Upon the interest reception,

[2]RSA 1024 and ECDSA 192 are considered to offer the same level of security.

	Vector of packets		Single packet	
Consumer: Signature verification				
RSA-1024	52.2us	[51.5 52.9]	140us	[132 149]
ECDSA-192	412us	[406 417]	757us	[697 817]
Producer: Signature computation				
RSA-1024	440us	[437 443]	775us	[733 818]
ECDSA-192	380us	[377 383]	701us	[661 740]
SHA-256 hash computation on MTU packet				
1.5kB	9.44us	[9.38 9.50]	28.62us	[31.03 32.08]
9kB	31.55us	[31.03 32.08]	68.26us	[63.63 72.89]

Table 2: Cryto operations cost in case of vector of packets and single packet computation.

the producer creates a new 1500 Bytes Data Packet, sings it and sends it back to the consumer that verifies the signature. This is a typical communication pattern of real time application where the rate is low and the application data can fit in one MTU packet (e.g. RTP). In this scenario, using a manifest is not the best choice, since the packets need to be forwarded as soon as the corresponding interests are received.

The delay introduced by the real time signature/verification of each packet is significant: with respect to the RTT measured without any crypto operation (145 us), the delay is one order of magnitude bigger (1173us for RSA and 1667us for ECDSA).

Type of test	Average	99% CI
No signature	145us	[136 155]
RSA-1024	1173us	[1142 1205]
ECDSA-192	1667us	[1621 1712]

Table 3: Average end-to-end latency growth in case of signature computation and verification.

7 LINEAR VIDEO DISTRIBUTION: MULTICAST AND SERVER LOAD

Linear video distribution is a challenging test case as it requires provable QoE in terms of video quality and application responsiveness, and is also supposed to scale to a very large number of watchers. Nowadays, one of the main limitations of the linear video distribution system are the server endpoints: the fact that a server needs to maintain a stateful TCP session per user does not scale when the number of users increases exponentially (think about popular sport events followed by millions of user). To cope with this, content providers use to load balance the clients requests among several instances of the same server for dealing with the problem of keeping so many TCP sessions open.

A fundamental difference between a TCP and a hICN socket is the fact that a hICN socket can serve/retrieve data to/from multiple destinations/sources. This allows to significantly reduce the server load, which can produce data once, instead of sending it to each client in unicast. Therefore, consumers' requests can be served by the transport itself once the data is made available after production.

In this section, we highlight the benefits produced by inserting our transport implementation at server side: we show a significant saving in terms of CPU and memory usage with respect to the current state of art.

7.1 Experimental setting

We set up a cluster of 150 video clients connected to an Apache Traffic Server (ATS) [1], configured as an HTTP reverse proxy with a 2GB cache (1GB of RAM cache and 1GB of raw device cache). As origin server we use an *nginx* [3] server live-feeded by a RTMP stream generated by the Open Broadcaster Software (OBS) [10]. *Nginx* utilizes the `nginx- rtmp` [4] module to provide multi-quality HLS streams. We stream 48 channels, each one encoded in 4 qualities (using bit rates suggested in [7]) with 2 second segments, ranging from 360p at 1Mbit/s to 1080p at 6Mbit/s.

To compare our hICN transport with TCP, we connect the ATS plugin to our hICN sockets and we create one producer socket per channel, with unreliable publication protocol (`CONS_UNREL`). For both TCP and hICN, we run 2 hours experiments for each different user population (50 100 or 150 clients). Each client requests one of the 48 available channels; the channel selection follows a Zipf distribution with $\alpha = 1.4$ or $\alpha = 0.7$. In order to increase the accuracy of our sampling distribution, we let each client switch channel every 10 minutes. Fig. 7 summarizes the results. The number of clients and corresponding average number of channels watched at the same time are reported in the header of the table.

7.2 Results

Results show that hICN socket considerably reduces the load at ATS w.r.t. all considered metrics. In particular, they confirm that server load is proportional to the number of clients while using a TCP socket, whereas is proportional to the number of channels being watched, in case of hICN. Moreover, they show that the system load reduction is considerable. For example, with 150 clients, ATS receives 80% less GET requests with hICN. Additionally, the hICN socket reduces memory utilization in ATS to about 60MB. This is because the hICN socket satisfies subsequent requests without passing them to the application. Therefore, as the caching replacement algorithm implemented in ATS is *scan resistant*, i.e., one-time GET requests do not affect the cache state, no entry is added into the cache.

Fig. 8 shows the ratio between the traffic sent by ATS and the total amount of traffic received by the clients using hICN. We show the results only for hICN, because for TCP this value is always equal to 100%, since all requests are served directly by ATS. The plot confirms that hICN can reduce the load on the server by more than 80%, depending on the considered workload.

We expect that the absence of a state machine in sockets will further diminish CPU and memory requirements on the server. Those metrics are however difficult to compare in the present state of implementation and we plan to perform a thorough comparison in future work.

8 RELATED WORK

Most of the work on transport layer for ICN has focused on congestion control which is receiver-driven as opposed to the current

Metric	N=50 [C=14]		N=100 [C=22]		N=150 [C=26]	
	IP/TCP	(h)ICN	IP/TCP	(h)ICN	IP/TCP	(h)ICN
R	438.6	128.5	937.8	192.0	1380.0	247.8
h/m	202.1 / 236.5	0 / 128.5	488.2 / 449.6	0 / 192.0	753.4 / 626.6	0 / 247.8
mem	1243.1	58.6	1311.7	62.8	1430.5	60.7
$\overline{\text{cpu}}$	17.4 $\pm_{9.3}$	5.7 $\pm_{5.2}$	30.7 $\pm_{12.1}$	7.5 $\pm_{6.0}$	45.2 $\pm_{15.5}$	8.7 $\pm_{7.5}$

R: #requests ($.10^3$), h/m: cache hits/miss ($.10^3$), mem: total memory (MiB) and $\overline{\text{cpu}}$: avg CPU usage (%)

Figure 7: ATS performance metrics with N clients (resulting in C active channels) with channel popularity ~ Zipf(α=1.4).

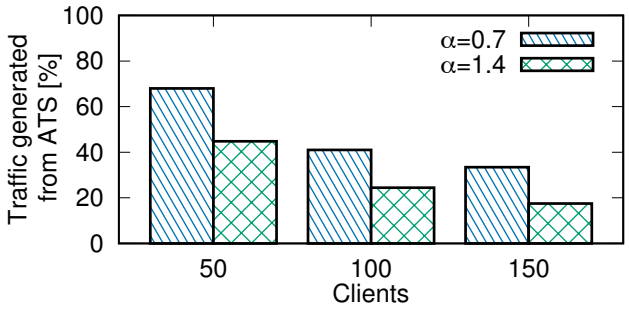


Figure 8: Percentage of traffic served by ATS using hICN.

sender-based TCP/IP model. Also (h)ICN transport does not require connection instantiation and accommodates retrieval from possibly multiple dynamically discovered sources.

It builds upon the flow balance principle guaranteeing corresponding request-data flows on a hop-by-hop basis [8]. A large body of work has looked into ICN transport (surveyed in [38]), not only to propose rate and congestion control mechanisms [39, 44] – especially in the multipath case [14], [29], [41] – but also to highlight the interaction with in-network caching [13], the coupling with request routing [14, 24], and the new opportunities provided by in-network hop-by-hop rate/loss/congestion control [12, 15, 43] for a more reactive low latency response.

Other work on the description of a transport layer and a socket API has appeared in [19] and [32]. However, none evaluates the performance of such transport stack and the design trade offs that need to be considered for different applications. Moreover, unlike [32], our socket API does not require to the application developer to implement signature calculation and verification, but offers them as a transport service. In the area of transport services and related API recent work at the IETF is considering the problem of providing and novel transport service API to replace the BSD like socket API. Relevant work is [37] and more generally the effort in the TAPS WG and the NEAT project [9]. [22] considers Hadoop on top of NDN by means of a Socket API that is very specific to this application, i.e. the ambition is not to provide a general transport service to any kind of application like in this paper.

9 DISCUSSION AND CONCLUSION

In this paper we have presented the design, implementation and benchmarking of a transport layer for (h)ICN with a socket API for applications. The implementation is based on a fast transport stack in userspace with kernel bypass based on VPP and DPDK. The present contribution applies to NDN/CCN and hICN and is currently open sourced in the Linux Foundation project FD.io [17].

We believe that such a transport layer allows to easily insert (h)ICN in today's applications such as web services and to develop new ones in a simpler way. The choice to build a transport stack in userspace allows to integrate new extension rapidly with little effort with no performance trade-off.

We have evaluated the transport layer in terms of performance and compared the implementation with the Linux TCP and the VPP TCP stack to have a baseline reference in terms of performance target for benchmark workloads. Experimental results in Section 6 show that the crypto operations have a significant impact on the overall transport performance, both in terms of application goodput and latency. This overhead can be easily reduced by offloading them to dedicated hardware such as Intel OAT acceleration available in the core itself or in external chips. Performing them in software is expensive and impairs the long term scalability of the transport stack. Even using a recent CPU technology, the delay introduced by crypto operations significantly restricts the maximum achievable throughput, and also the signature/verification performance becomes really system dependent.

The availability of hardware offloading techniques for the transport layer tends to be appear when needed. As an example, TSO/LRO is available already in any end device with Gbps interfaces, including laptops. The large usage of TLS in the Internet is also going to make several cryptographic accelerations available in silicon through most used crypto library, both for client end devices, including mobiles, and server ends. In this paper we have shown that a pure software implementation can still provide decent performance with several limitations. This reality check allows to claim that many applications, such as video delivery and real time communications, are feasible with an (h)ICN software stack that is portable into a large set of end devices. The current design and implementation has been already used to enable (h)ICN transport in WebRTC for RTP transported media.

While the transport layer that we present in this paper constitutes a fundamental component for integration of ICN into today's and future application, we recognize that named data provides a larger degree of freedom in the way application namespaces are mapped into network namespaces. Future work is needed to make further progress to define an additional session layer, where application namespaces are multiplexed into multiple network namespaces.

REFERENCES

[1] 2018. Apache Traffic Server. http://trafficserver.apache.org/
[2] 2018. *lxc*. https://linuxcontainers.org/.
[3] 2018. *nginx*. https://nginx.org/en/.
[4] 2018. *nginx* RTMP module. https://nginx.org/en/.
[5] 2018. *openSSL*. https://www.openssl.org/.
[6] 2018. Iperf. https://iperf.fr.
[7] 2018. Live encoder settings, bitrates, and resolutions. http://goo.gl/tDtc1i.
[8] 2018. NDN project, Named-Data Networking Principles. https://named-data.net/project/ndn-design-principles/.
[9] 2018. The NEAT project. https://www.neat-project.org
[10] 2018. Open Broadcaster Software (OBS). https://obsproject.com/.
[11] M. Baugher, B. Davie, A. Narayanan, and D. Oran. 2012. Self-verifying names for read-only named data. In *Proc. of IEEE INFOCOM 2012 Workshops*. Orlando, FL, USA, 274–279.
[12] G. Carofiglio, M. Gallo, and L. Muscariello. 2012. Joint Hop-by-hop and Receiver-driven Interest Control Protocol for Content-centric Networks. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 491–496.
[13] G. Carofiglio, M. Gallo, and L. Muscariello. 2013. On the Performance of Bandwidth and Storage Sharing in Information-centric Networks. *Computer Networks* 57, 17 (Dec. 2013), 3743–3758.
[14] G. Carofiglio, M. Gallo, and L. Muscariello. 2016. Optimal multipath congestion control and request forwarding in information-centric networks: Protocol design and experimentation. *Computer Networks* 110 (2016), 104–117.
[15] G. Carofiglio, L. Muscariello, M. Papalini, N. Rozhnova, and X. Zeng. 2016. Leveraging ICN In-network Control for Loss Detection and Recovery in Wireless Mobile Networks. In *Proc. of the 3rd ACM SIGCOMM ICN ('16)*. New York, NY, USA, 50–59.
[16] Dino Farinacci, Vince Fuller, David Meyer, and Darrel Lewis. 2013. The Locator/ID Separation Protocol (LISP). RFC 6830. https://doi.org/10.17487/RFC6830
[17] Linux Foundation FD.io. 2018. CICN project, wiki page. https://wiki.fd.io/view/Cicn.
[18] International Organization for Standardization (ISO). 2012. ISO 26324:2012 Information and documentation – Digital object identifier system.
[19] M. Gallo, L. Gu, D. Perino, and M. Varvello. 2014. NaNET: Socket API and Protocol Stack for Process-to-content Network Communication. In *Proc. of the 1st ACM SIGCOMM ICN ('14)*. New York, NY, USA, 185–186.
[20] C. Ghali, A. Narayanan, D. Oran, G. Tsudik, and C. A. Wood. 2015. Secure Fragmentation for Content-Centric Networks. In *2015 IEEE 14th International Symposium on Network Computing and Applications*. 47–56. https://doi.org/10.1109/NCA.2015.34
[21] C. Ghali, G. Tsudik, and C. A. Wood. 2016. Network Names in Content-Centric Networking. In *Proc. of the 3rd ACM SIGCOMM ICN ('16)*. New York, NY, USA, 132–141.
[22] Mathias Gibbens, Chris Gniady, Lei Ye, and Beichuan Zhang. 2017. Hadoop on Named Data Networking: Experience and Results. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 1, Article 2 (June 2017), 21 pages. https://doi.org/10.1145/3084439
[23] R.E. Gilligan, J. McCann, J. Bound, and S. Thomson. 2003. *Basic Socket Interface Extensions for IPv6*. Technical Report 3493. https://rfc-editor.org/rfc/rfc3493.txt
[24] S. Ioannidis and E. Yeh. 2017. Jointly optimal routing and caching for arbitrary network topologies. In *Proc. of the 4th ACM SIGCOMM ICN ('17)*. Berlin, Germany, 77–87.
[25] A. Langley et al. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 183–196. https://doi.org/10.1145/3098822.3098842
[26] Linguaglossa, Leonardo and Rossi, Dario and Pontarelli, Salvatore and Barach, Dave and Marjon, Damjan and Pfister, Pierre. 2018. High-speed Software Data Plane via Vectorized Packet Processing. https://perso.telecom-paristech.fr/drossi/paper/vpp-bench-techrep.pdf.
[27] Linux Foundation FD.io. 2018. White Paper - Vector Packet Processing - One Terabit Software Router on Intel Xeon Scalable Processor Family Server. https://fd.io.
[28] University of Pisa Luigi Rizzo. 2018. Netmap - the fast packet I/O framework. http://info.iet.unipi.it/ luigi/netmap/.
[29] Milad Mahdian, Somaya Arianfar, Jim Gibson, and Dave Oran. 2016. MIRCC: Multipath-aware ICN Rate-based Congestion Control. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking (ACM-ICN '16)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2984356.2984365
[30] Michael H. Mealling. 2001. A URN Namespace of Object Identifiers. RFC 3061. https://doi.org/10.17487/RFC3061
[31] Michael H. Mealling. 2001. The Network Solutions Personal Internet Name (PIN): A URN Namespace for People and Organizations. RFC 3043. https://doi.org/10.17487/RFC3043
[32] I. Moiseenko, L. Wang, and L. Zhang. 2015. Consumer / Producer Communication with Application Level Framing in Named Data Networking. In *Proc. of the 2nd ACM SIGCOMM ICN ('15)*. New York, NY, USA, 99–108.
[33] Marc Mosko, Ignacio Solis, and Christopher A. Wood. 2018. *CCNx Messages in TLV Format*. Internet-Draft draft-irtf-icnrg-ccnxmessages-08. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-irtf-icnrg-ccnxmessages-08 Work in Progress.
[34] Marc Mosko, Ignacio Solis, and Christopher A. Wood. 2018. *CCNx Semantics*. Internet-Draft draft-irtf-icnrg-ccnxsemantics-09. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-irtf-icnrg-ccnxsemantics-09 Work in Progress.
[35] M. Mosko and C. A. Wood. 2015. Secure Fragmentation for Content Centric Networking. In *2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems*. 506–512. https://doi.org/10.1109/MASS.2015.51
[36] Luca Muscariello, Giovanna Carofiglio, Jordan Auge, and Michele Papalini. 2018. *Hybrid Information-Centric Networking*. Internet-Draft draft-muscariello-intarea-hicn-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-muscariello-intarea-hicn-00 Work in Progress.
[37] Tommy Pauly, Brian Trammell, Anna Brunstrom, Gorry Fairhurst, Colin Perkins, Philipp S. Tiesel, and Christopher A. Wood. 2018. *An Architecture for Transport Services*. Internet-Draft draft-pauly-taps-arch-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-pauly-taps-arch-00 Work in Progress.
[38] Y. Ren, J. Li, S. Shi, L. Li, G. Wang, and B. Zhang. 2016. Congestion Control in Named Data Networking - A Survey. *Computer Communications* 86, C (July 2016), 1–11.
[39] L. Saino, C. Cocora, and G. Pavlou. 2013. CCTCP: A scalable receiver-driven congestion control protocol for content centric networking. In *Proc. of IEEE ICC 2013*. Budapest, Hungary, 3775–3780.
[40] M. Sardara, L. Muscariello, J. Augé, M. Enguehard, A. Compagno, and G. Carofiglio. 2017. Virtualized ICN (vICN): Towards a Unified Network Virtualization Framework for ICN Experimentation. In *Proc. of the 4th ACM SIGCOMM ICN (ICN '17)*. ACM, New York, NY, USA, 109–115.
[41] Klaus Schneider, Cheng Yi, Beichuan Zhang, and Lixia Zhang. 2016. A Practical Congestion Control Scheme for Named Data Networking. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking (ACM-ICN '16)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/2984356.2984369
[42] The Linux Foundation Projects. 2018. Data Plane Development Kit. https://dpdk.org.
[43] Y. Wang, N. Rozhnova, A. Narayanan, D. Oran, and I. Rhee. 2013. An Improved Hop-by-hop Interest Shaper for Congestion Control in Named Data Networking. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 55–60.
[44] F. Zhang, Y. Zhang, A. Reznik, H. Liu, C. Qian, and C. Xu. 2014. A transport protocol for content-centric networking with explicit congestion control. In *Proc. of the 23rd ICCCN 2014*. Shanghai, China, 1–8.