# iStack: A General and Stateful Name-based Protocol Stack for Named Data Networking

Tianlong Li[†] Tian Song[‡, *] Yating Yang[‡]
[†] *School of Computer Science and Technology*
[‡] *School of Cyberspace Science and Technology*
*Beijing Institute of Technology*

## Abstract

Named Data Networking (NDN) shifts the network from host-centric to data-centric with a clean-slate design, in which packet forwarding is based on names, and the data plane maintains per-packet state. Different forwarders have been implemented to provide NDN capabilities for various scenarios, however, there is a lack of a network stack that is integrated with operating systems (OS) for general purpose. Designing a stateful and entirely name-based protocol stack in OS kernel remains a challenge due to three factors: (i) an in-kernel name resolution architecture for packet demultiplexing is necessary, (ii) an entirely name-based stack requires to be compatible with the current address (MAC/IP/port)-based architecture in OS kernel, and (iii) maintaining per-packet state introduces a trade-off between performance and resource consumption.

In this paper, for the first time, we take NDN into OS kernel by proposing iStack, an Information-Centric Networking (ICN) protocol stack. The main innovations of iStack are threefold. First, we propose a name resolution architecture to support both network-layer forwarding and local packet demultiplexing. Second, a two-layer face system is proposed to provide abstraction of address-based network interfaces. Third, we design socket-compatible interfaces to keep the uniformity of current network stack in OS. Besides, we design compact forwarding data structures for fast packet processing with low memory footprint. We have implemented prototypes on multiple platforms. The evaluation results show that iStack achieves 6.50 Gbps throughput, outperforming the NDN-testbed forwarder by a factor of 16.25x, and reduces 46.08% forwarding latency for cached packets with its in-kernel packet caching. iStack is not just another forwarder for NDN, but a step forward for practical development of ICN.

## 1 Introduction

Named Data Networking (NDN) [1] has been regarded as a leading Information-Centric Networking (ICN) design, which addresses data directly with hierarchical names rather than hosts. Its novel architecture enables a number of valuable features, including in-network caching [2, 3], securing data directly [4, 5], mobility support [6, 7], and native multicast support [8, 9]. Furthermore, a gateway for integrating NDN into the Internet has been designed [10]. Many studies have explored NDN's advantages in diverse scenarios [11–14].

NDN Forwarding Daemon (NFD) has been implemented and intended to be used as a general-purpose forwarder since 2014 [15, 16]. It is effective in supporting architectural research and being used on the public NDN project testbed [17]. However, it is a user-space forwarder rather than an operating-system resident. In comparison, in the following three ways, an in-kernel network stack like TCP/IP gains additional benefits for practical deployment with operating systems support.

First, a stack integrated into the operating system is designed to work with the socket mechanism. It can provide system-level network functionality and be shared by multiple applications with varying purposes and requirements. Second, an in-kernel stack runs as kernel threads, which gain system-level security from the OS kernel protection. Typically, the x86 architecture has 4 rings of protection, with the kernel running in ring 0 with the greatest privileges and user programs running in ring 3. Because user-space programs cannot access kernel-space memory, the in-kernel networking architecture is naturally isolated and protected from applications. Third, an integrated stack can evolve with operating systems for the long term, providing a stable foundation for network communication. As operating systems are updated or new hardware is introduced, an in-kernel network stack can adapt to these changes seamlessly. Hence, it is worthwhile to take NDN, a clean-slate networking architecture, into operating systems.

However, there are three challenges for designing an in-kernel stack for NDN. First, name-based packet demultiplexing requires a well-thought-out design. A general stack needs to distinguish and deliver packets among local applications, which is done by transport-layer ports in TCP/IP stack. Specifically, an application creates a socket and binds it to a port. The port number is carried in packets and identifies the appli-

[*] Corresponding author: Tian Song (songtian@bit.edu.cn)

cation to the receiving end, whereas NDN is entirely based on names. An in-kernel stack for NDN requires a name resolution architecture for packet demultiplexing.

Second, an in-kernel NDN stack should be compatible with the existing TCP/IP stack in OS for dual-stack usage. The traditional network stack relies on address (MAC/IP/port) based communication. In contrast, NDN has no endpoints or host addresses. Therefore, an abstraction of network interfaces to support network processing is necessary. NFD incorporates a face mechanism intending to achieve this [16]. However, an in-kernel stack needs to interact with both application sockets and link-layer physical/virtual devices, presenting significant differences from a user-level forwarder.

Third, a general-purpose network stack for NDN should balance its performance and resource consumption. Unlike IP's stateless data plane, NDN adopts a stateful forwarding model. For the first time, a network stack requires large memory for in-network caching and involves multiple table lookups, which remains a challenge for an efficient stack design.

In order to address these challenges, we propose iStack, a general and stateful name-based protocol stack in OS kernel for NDN. The stack itself is native to OS kernel and compatible with other in-kernel network protocol suites. Meanwhile, iStack is an information-centric stack that is entirely based on names and enables kernel-level NDN functionality. Our major contributions are listed as follows:

First, we propose iStack, which for the first time, takes NDN into OS kernel to support application hosts. iStack has a fully name-based architecture. It adopts NDN forwarding model and introduces a name resolution architecture for demultiplexing. Its architecture can support applications with the native socket mechanism in OS.

Second, we design a two-layer face system as the network interface abstraction, which maps both of high-layer socket interfaces and low-layer network devices into logical *FaceIDs*. Hence, iStack uses logical *FaceIDs* for network processing.

Third, we design high-speed forwarding data structures, which efficiently maintain per-packet state and cache packets directly in the kernel. Furthermore, we also implement a fast two-level lock mechanism to ensure multi-threading safety while leveraging parallel processing for higher performance.

Besides, we implemented iStack prototypes in two different Linux kernel versions, four different Linux distributions (Ubuntu, CentOS, Raspberry Pi OS Lite, and OpenWrt), and four different platforms (x86 server, personal computer, Raspberry Pi and edge router (MT7621AT)). Our evaluation results show that iStack achieves 6.50 Gbps throughput, measured with the standard size (1500 bytes) of Ethernet frames. Its throughput outperforms the general NDN forwarder, NFD, by a factor of 16.25x. Moreover, iStack takes in-network caching into kernel space, reducing 46.08% forwarding latency for cached packets. Unlike the previous forwarders, iStack focuses on providing NDN functionality on end hosts for applications rather than mainly supporting high-speed routing/forwarding. Our prototypes demonstrate that iStack is promising to advance ICN deployment.

The rest of this paper is organized as follows. Sec. 2 introduces the background as well as the related work. Sec. 3 details the design and architecture of iStack. Sec. 4 describes a suite of efficient extensions and miscellaneous considerations for iStack. Sec. 5 presents the evaluation results of our implemented prototypes and includes lessons learned. Sec. 6 finally concludes this paper and discusses our future work. This work does not raise any ethical issues.

## 2 Background

### 2.1 Named Data Networking

NDN is one of the most important instances of ICN [1]. It uses hierarchically structured names to address the content. NDN uses '/' to indicate the boundaries of name components. For example, segment 2 of version 1 of a video provided by Alice may have the name: */Alice/videos/DemoA.flv/1/2*, which contains three parts: where to forward it (*/Alice*), which application to deliver it (*/videos*), and application-specific information (*/DemoA.flv/1/2*). As NDN matters the content rather than where it is from and caches *Data* in the network, security is built into content itself [4]. It adopts a stateful forwarding model that primarily consists of three structures: Pending Interest Table (PIT), Forwarding Information Base (FIB) and Content Store (CS).

In NDN, a consumer (client) requests content by sending a packet with the content name, namely *Interest*, to the network. Any node receiving it tries to find the corresponding *Data* identified by that name in its CS. If it is found, the *Data* will be returned. Otherwise, the node inserts this *Interest* into PIT and then forwards it based on both the matching result from FIB and the forwarding strategy. When the *Interest* finally meets the corresponding *Data* at the producer (server) or a CS on the *Interest* forwarding path(s), the *Data* is sent back in the reversed path of the *Interest* and meanwhile removes those records in PITs. In particular, different *Interests* with the same name can be merged into a single PIT entry without redundant forwarding. In this case, the merged PIT entry records all *Interests* incoming interfaces. When the *Data* is back, it is forwarded via multicast based on that PIT record. With this design, NDN gains a number of benefits such as network-layer request aggregating and native multicast support.

### 2.2 Related Work

In the last decade, NFD, the most important NDN forwarder, has been designed and implemented for architectural research [15]. Besides, several dedicated forwarders are proposed to meet the requirements of different scenarios.

NFD is designed modularly and extensively to support diverse NDN technology experimentation [15]. It is written

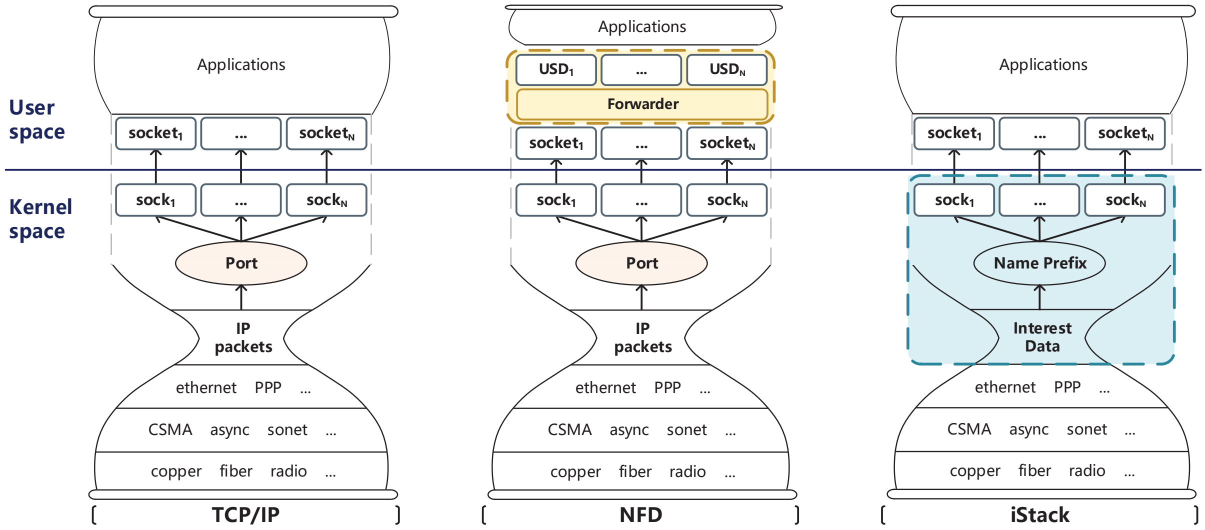


Figure 1: Comparison among TCP/IP stack, NDN user-space forwarders, and iStack from a high-level perspective

in C++ language and has a well-modular design. Although it intends to be used for general purposes, NFD is a user-space forwarder rather than an in-kernel stack. A user-space forwarder must use an inter-process communication (IPC) mechanism (specifically in NFD, Unix domain socket [16]) to communicate with local applications. It does not provide system-level support for applications.

NDN-RIOT is proposed as a lightweight implementation for RIOT-OS [18]. RIOT is a micro-kernel operating system for resource-limited IoT devices. It does not support virtual memory and separate of user and kernel space, which is common in modern general OS. Hence, although NDN-RIOT integrates the core NDN forwarding logic into the RIOT-OS kernel and demonstrates the feasibility of bringing NDN's data-centric communication and security model to constrained IoT platforms, it is a dedicated solution for particular scenarios rather than a general forwarder.

For further supporting embedded/constrained environments, a low-overhead forwarder, NDN-Lite is carried out [19]. Unlike NFD, NDN-Lite simplifies the forwarder design and is written in C language. However, NDN-Lite has to integrate with the thread of a NDN application and can only support a single application simultaneously. This is suitable for low-end platforms but meanwhile limits its generality.

In order to achieve high-speed NDN forwarding, NDN-DPDK is proposed [20] and reaches a peak forwarding rate of 1.84 MPPS[1] by leveraging the fast user-space packet processing framework Data Plane Development Kit (DPDK) [21]. For paralleling, NDN-DPDK dispatches the incoming *Interests* to the private sharded data structures of threads by hashing their name prefixes. For dispatching retrieved *Data* to the correct thread, this work proposes PIT token, a small hop-by-hop header field added to each packet. Due to its high performance, NDN-DPDK may be suitable for throughput-sensitive cases such as backbone routers and data centers. However, the DPDK-based solution limits the generality of it.

MW-NFD is proposed to provide a high-performance forwarder without DPDK support [22]. It splits the forwarding pipeline into multiple threads and follows the packet dispatching method of NDN-DPDK. Its polling-based design has a 100% CPU use rate, which is not suitable for general purposes.

The forwarders discussed above either achieve high performance but are unsuitable for general purpose or vice versa. Given the opportunity that it lacks a more efficient NDN forwarder for edge, YaNFD is proposed [23]. YaNFD is a multi-threaded alternative to existing software packet forwarders for the NDN architecture. The performance of it is higher than that of NFD and lower than that of MW-NFD, while it keeps a reasonable resource usage for generality. Nevertheless, it is written in Go language as a user-space forwarder, which suffers from the garbage collection overhead.

From the practical usage perspective, the genuine requests and content are finally from and to applications running in OS. Hence, a forwarder enabling NDN packet forwarding may be suitable for supporting architectural research purposes, whereas a network stack integrated with OS kernel is still required to provide system-level supporting for real applications. This is also our motivation for proposing iStack. Fig. 1 illustrates the comparison among TCP/IP stack, NDN user-space forwarders, and iStack from a high-level perspective.

## 2.3 Network Stack in OS Kernel

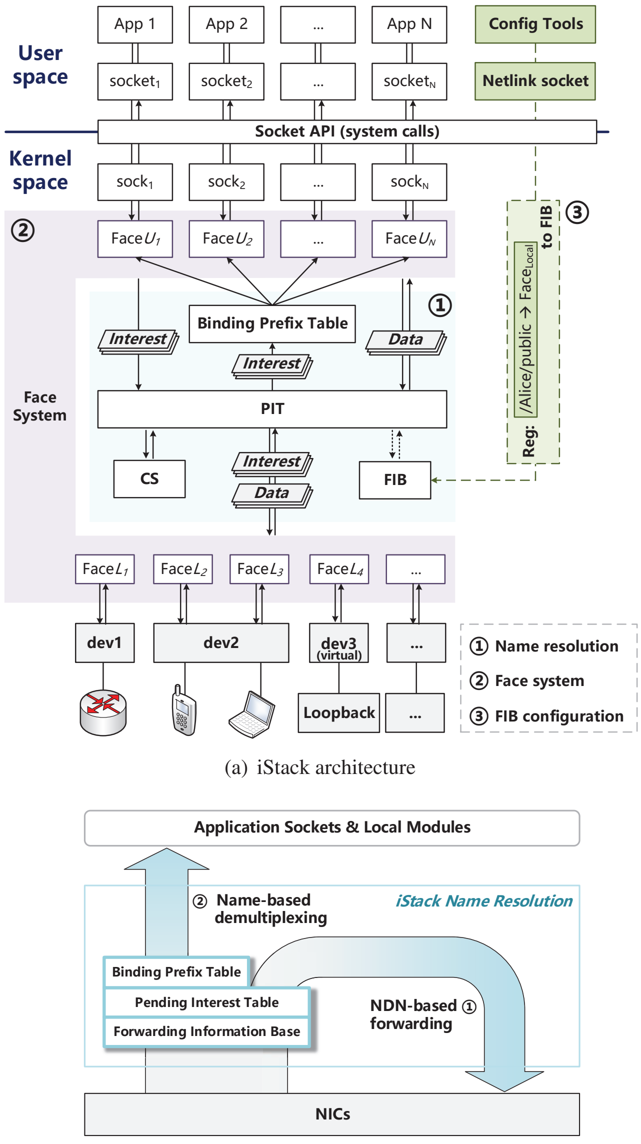
Modern operating systems implement network protocols from the link layer to the transport layer. They also implement

[1] In form of throughput, it achieves 22 Gbps with 1500 Bytes standard Ethernet frame and 108 Gbps with 8000 Bytes jumbo frames.

a version of Berkeley socket as application programming interfaces (API). Different layers in a network stack have their own abstractions. Taking the Linux kernel network stack for example, in its link layer, NICs and their drivers are abstracted as *net_device*(s), bound with their hardware MAC addresses.

In the network layer, *address_family* is used to specify the network stack suites, such as AF_INET and AF_INET6 for IPv4 and IPv6, respectively. As the transport layer provides end-to-end services for endpoints of network communication, it is integrated with the network socket. A socket created by an application has two parts: *socket* as an abstraction in user space for the application and *sock* as the in-kernel network representation. The transport-layer functionalities are integrated with the *sock* and specified by *sock_type*. For example in IP stack, SOCK_STREAM and SOCK_DGRAM represent for TCP and UDP, respectively. Hence, the network layer is shared in a host, whereas the transport layer is isolated by different applications with their created sockets separately. Notably, a non-port-based raw type socket allows applications to directly access network/link-layer packets. It catches all packets and shifts the responsibility of (de)multiplexing from the kernel-space network stack to the user-space forwarders/applications. Yet a general-purpose in-kernel network stack requires suitable (de)multiplexing models.

As presented in Fig. 1, the current network stack is address-based: packets go through a process of filtering by network-layer addresses, which are then distinguished by transport-layer ports before finally being delivered to their intended sockets of applications. In contrast, NDN takes all functionalities into name prefixes and there is no port. Therefore, an in-kernel network stack for NDN requires a name resolution architecture for network-layer forwarding and local packet demultiplexing. The current NDN forwarder-based solutions use IPC mechanisms for applications, leaving system-level support for applications as an open issue. iStack is designed to address this and takes NDN into OS kernel.



(a) iStack architecture

(b) Two paths of name resolution

Figure 2: iStack architecture and name resolution

# 3 iStack Design and Architecture

## 3.1 Design Overview

As an in-kernel network stack, iStack operates between the applications sockets and the link-layer devices. As illustrated in Fig. 2(a), iStack primarily consists of three parts: a name resolution architecture, a two-layer face system, and socket-compatible interfaces for applications and configuration.

The name resolution architecture enables packet demultiplexing for local applications with two packet paths as shown in Fig 2(b). For supporting name-based *Interests* demultiplexing, it introduces a Binding Prefix Table (BPT) which records the relationship between application sockets and their prefixes. iStack separates application binding prefixes from registering routable prefixes to the network.

In order to integrate the name resolution architecture with OS kernel compatibly, iStack involves a two-layer face system, which abstracts upper layer handles and lower layer communication channels as general interfaces. It separates the upper and lower layer due to the fact that different layers treat semantics of names differently.

iStack provides interfaces based on socket mechanism. Applications can use network sockets to communicate with iStack, following a practice similar to TCP/IP. The only difference is that in TCP/IP, a socket needs to bind an <*address, port*> pair, whereas in entirely name-based iStack, there are no addresses or ports. Instead, an iStack socket binds a local *faceID* to represent where packets are from. A provider-side application socket also needs to bind a prefix to notify iStack where *Interests* should be delivered to. iStack configuration is isolated from the common network socket communication for system security consideration. Specifically in the Linux kernel, it is through Netlink socket with superuser privilege.

## 3.2 Name Resolution Architecture

In iStack, name resolution utilizes NDN-semantic names, which carry the information of forwarding and demultiplexing. In this paper, we use the term *routable prefix* to represent a prefix that can be forwarded to, and the term *application-specific prefix* to represent an extension of the routable prefix used for distinguishing applications. For example with the name: */Alice/videos/DemoA.flv/1/2*, its routable prefix and application-specific prefix could be */Alice* and */Alice/videos*, respectively. According to the two types of prefixes, the packet processing path is divided into *network-layer forwarding path* and *local delivering path* as shown in Fig. 2(b).

For the network-layer packet forwarding, iStack follows the basic design of NDN. *Interests* are pending in PIT and forwarded based on FIB and forwarding strategies. *Data* packets are forwarded along the reverse path of the pending *Interests*. The main difference is the serving scope of FIB. In previous design of NDN forwarders like NFD, a local application registers the content prefixes with its identification into FIB, hence the forwarder can deliver the corresponding *Interests* to it. In contrast, FIB in iStack serves for the network layer only, which means it only records routable prefixes rather than a vast number of application-specific prefixes.

FIB in iStack can only be updated by (i) routing protocols running in kernel threads and (ii) user-space configuration tools. The former one updates FIB with routing information learned from the network and the later one is used by superusers to register routable prefixes of providers on this host. The network layer only determines whether *Interests* should be delivered to this host based on routable prefixes in FIB and hands over these *Interests* to BPT. FIB is agnostic to application-specific prefixes and local packet distinguishing.

The local part of name resolution is situated above the network layer and provides local packet distinguishing functionality. It maintains BPT, which records the mapping between application sockets and application-specific prefixes. Indicated by its name, an application registers its application-specific prefix through socket API *bind*.

There are two rules for applications to bind their prefixes. First, in BPT, any prefix *must not* be identical to or be a proper prefix[2] of another prefix. Otherwise one application may hijack the *Interests* intended for other applications. Second, an application-specific prefix *should* be an extending string of a routable prefix that has been pre-registered in FIB. Otherwise, the application may never receive any *Interests* as the network layer does not pass *Interests* with that prefix to local host. The local name resolution provides flexibility for applications by using finer-grained and more complex prefixes without messing up FIB. As needed, an application can create multiple sockets and bind them to different application-specific prefixes which share a common routable prefix.

## 3.3 Two-Layer Face System

The face system is designed to (i) hide details of various interfaces and (ii) ensure thread safety against the system events such as interface adding/removing. It provides *Faces*, the generalization of different network interfaces.

From the layered networking perspective, different layers have different targets and communicating scopes. In iStack, for packets travelling down to the lower layer, i.e. the link layer, the semantics of name/prefix is not concerned. Whereas for packets travelling up to the upper layer like applications, the name/prefix semantics may still be concerned. Due to this difference, iStack contains a two-layer face system to provide the generalization of interfaces from different layers.

### 3.3.1 Face Upper Layer

The face upper layer targets to abstract the local handles upon the network layer, which commonly are application sockets. An upper face consists of a unique *FaceID*, a handler, and a prefix. The handler is composed of an entity pointer, which points to the upper entity represented by the face, and a function entry pointer which points to the handler function of that upper layer entity. If an entity wants to receive *Interests*, then the prefix field of its face has to be specified. Otherwise this field is null. In the network layer, PIT and FIB only use *FaceIDs* to represent network interfaces. Face upper layer maintains a face upper table to map local handles to faces.

In the network part of iStack name resolution, incoming *Interests* are filtered by FIB, and those intended for local applications are passed to BPT. In iStack network layer, there are only *FaceIDs* for interfaces and entrances of BPT is also encapsulated as faces. Face upper layer provides a permanent face, $Face_{local}$, as the local entrance. Routable prefixes that can be served by the host are registered to FIB with $FaceID_{local}$. When provider-side applications bind their sockets with application-specific prefixes, the prefixes are mapped to their faces and then the faces are put into BPT. When an *Interest* comes, it is firstly passed to $Face_{local}$ based on FIB. Then the handler of $Face_{local}$ queries BPT and returns the corresponding face. Finally the handler of the returned face delivers the *Interest* to the application socket.

### 3.3.2 Face Lower Layer

The face lower layer targets to abstract the link layer communication channels. It uses a face lower table to map link-layer interfaces to faces. A lower face consists of a unique *FaceID*,

[2] In this paper, we use *proper prefix* to indicate a component-level prefix of a name, such as */A/B* or */A* for */A/B/file*

a device, and a field for transmitting information. The device can indicate either physical hardware devices like ethernet adapters or virtual devices such as loopback device. Specifically in implementation, all link-layer devices for network accessing in the Linux kernel are abstracted as *net_device* and hence, the device of a lower face is a pointer of that.

A single device is not enough to transmit link-layer frames. A link-layer device may be connected with different remote devices at different time or even with multiple remote devices at the same time, like the wireless NIC of a WLAN access point. Besides, the link-layer transmission includes unicast, multicast and broadcast. Only broadcast needs no remote information for transmitting while the other two needs the destination address(es) of the remote host(s) or group(s). Thus, the field for transmitting information is used for recoding remote host information which typically is a destination MAC address. Obviously, a lower face represents for the channel between a local device and a remote host/group.

### 3.3.3 Thread-Safe Face Operations

The principle of face system is that the control path should not expose details to the data path and the data path processing should not block the control path operations. To achieve these, the face structure has a field of user count and thread-safe function pairs are provided: *face_register* and *face_release* for the control path; *face_hold* and *face_put* for the data path.

In the control path, *face_register* is used to create a new face and bind a network interface to it. Meanwhile, the user count of it is set to 1. For interface removing operations, such as closing a socket or taking a network adapter offline, the control path uses non-blocking function *face_release*. The function decreases the user count value of the face and its further behaviour depends on the result. If the decreased user count is 0, which means the face is not in use, then *face_free* is called to free the face and recycle its *FaceID*. In the other case, a positive user count means that the face is still referenced by at least one thread, then *face_orphan* is called. The function *face_orphan* releases everything in the face except the shell of it and meanwhile, it set the *FaceID* to 0 and the handler to *face_output_blackhole*, which drops any packets passed to it. The release of the face shell is delayed to the last data path calling *face_put*. Hence, the data path threads do not block the control path operations such as closing a socket.

In the data path, when a *FaceID* is transformed to the face, *face_hold* is called to increase the user count of it. When the packet processing is finished, *face_put* is called to decrease its user count. Specially, if the decreased user count is 0, which means the real interface of the face has been dead, then *face_free* is called to free the shell of this face. The function pair *face_hold* and *face_put* only involves an integer increasing/decreasing operation in common cases and a memory free operation in the case of interfaces removing. Thus the performance influence from control path events is minimized.

## 3.4 Socket-Compatible Interfaces

### 3.4.1 Forwarding Configuration

As shown in Fig. 2(a), configuration of iStack is isolated from common network socket communication. iStack utilizes a specific communication mechanism between kernel and user space supported by operating systems to provide control functionalities such as configuring FIB and exporting information.

Specifically in the Linux kernel, iStack exploits two mechanisms: the *procfs* and the Netlink socket. *Procfs* is a virtual filesystem which allows the Linux kernel to export internal information to user space. Netlink socket is more powerful than *procfs*. *Procfs* is able to read/write kernel exported data as operating files, while the Netlink socket communicates with kernel using socket programming model and moreover, the later one can initiate a transmission rather than only waiting for responses to user-space requests. Hence, iStack uses *procfs* to export its running status and statistical information in form of files as convenient interfaces. Meanwhile, we develop some utility tools with the Netlink socket for configuration requirements such as FIB configuration and CS management.

### 3.4.2 Application View

The common communication between applications and iStack is entirely based on the network socket mechanism. An application uses iStack via creating a socket with the address family AF_NNET. Then, the application can send/receive NDN packets with the socket. We provide three socket types to support different application requirements (detailed in Sec. 4.3.3). Note that a socket of iStack can support both consumer-side and provider-side applications at the same time. Just for ease of understanding, we introduce them separately.

For a consumer-side application, it can send out *Interest* and receive the corresponding *Data* once the socket is created. When *send* is called for the first time, the face system allocates a *FaceID* and automatically *bind* the socket to it. Then the *Interest* goes to the network layer through that face and is inserted into PIT and then forwarded based on FIB and forwarding strategies. When the corresponding *Data* is retrieved, it matches the record in PIT and then is passed to the socket via the *FaceID*. The *Data* is put in the socket receiving queue and waits until the application fetches it with *recv*, or in other cases, the application is blocked with *recv* until the *Data* queues in the socket and triggers a software interrupt request to retrieve it. Note that an application can send out a batch of *Interests* and receive the *Data* packets one by one. The order can be implied by the segment numbers contained in names. The socket owns the face until it is closed.

For a provider-side application, it needs one more step to receive *Interests*. That is, using *bind* to bind a prefix. The face system allocates a *FaceID* and maps the binding prefix to it in BPT. When an intended *Interest* arrives, it is inserted into PIT and then finds out that the outgoing interface recorded in FIB is *FaceID*$_{local}$. After matching the application-specific prefix in BPT, the *FaceID* of that application's socket is returned and the *Interest* is delivered to that socket. Finally the application receives the *Interest* and sends out the corresponding *Data* which will go along the reverse path of that *Interest*.

iStack adopts the sockets API rather than the established NDN API. Consequently, it is presently incompatible with existing NDN applications. However, rectifying this disparity through API code modification is a straightforward process. Our forthcoming endeavor involves streamlining the transition of existing NDN applications to iStack. To achieve this, we will provide a user-space library that seamlessly translates the current NDN API into iStack socket API. Furthermore, we are proactively engaged in the development of an information-centric programming model, complemented by a dedicated set of primitives and a socket-based API tailored for the advancement of future NDN applications.

## 4 Efficient Design and Extension for iStack

### 4.1 Compact Forwarding Data Structures

#### 4.1.1 Design Rationale

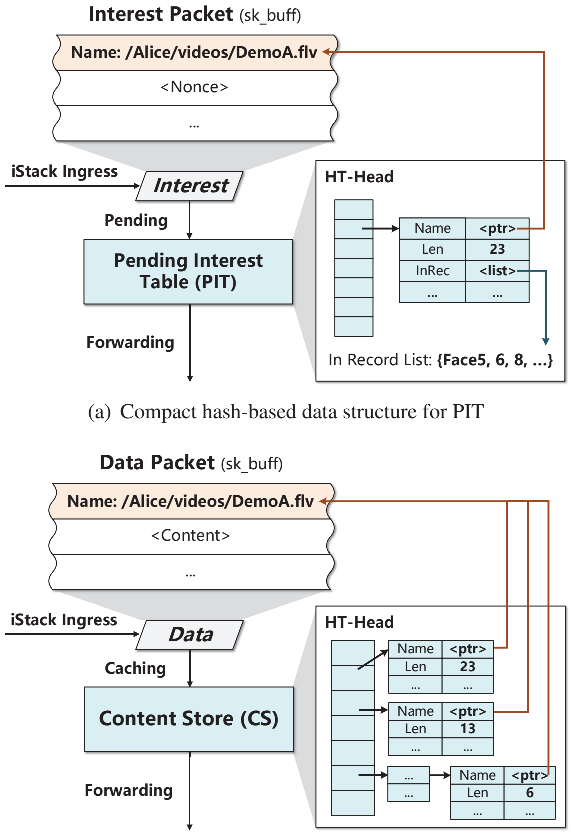
In NDN design, each *Interest/Data* carries a variable-length hierarchical name. As PIT maintains per-packet state and CS caches content in the network, it takes much more memory than traditional stateless network stacks. An in-kernel general-purpose network stack should take the trade-off between performance and resource consumption. In order to reduce the kernel-space memory consumption while keeping fast packet processing, we design compact forwarding data structures for PIT and CS in iStack.

The compact design ingeniously utilizes the fact that the life time of a PIT/CS entry is the same as that of an *Interest/Data* packet. In iStack, each PIT entry keeps at least one *Interest* packet pending for the potential retransmission in its lifetime and obviously, the entry name is carried by that *Interest*. For CS, it caches *Data* packets in kernel directly and each of its entry name is carried by the corresponding *Data*.

#### 4.1.2 Compact Hash-Based Data Structures

PIT needs to support both exact name matching (ENM) and longest prefix matching (LPM). Hash table is naturally fit for ENM. LPM in PIT is for the situation where an *Interest* is marked with *CanBePrefix* flag. Such an *Interest* can be satisfied by the *Data* with the same name prefix. As PIT only records *Interest*, LPM needs no modification of the hash table, but requires a particular lookup procedure for incoming *Data*.

The compact PIT structure is shown in Fig. 3(a), which is a separated chaining hash table and takes *Interest* names as keys. For an incoming *Interest* whose name has not been recorded before, PIT creates an entry and pends the packet directly. The entry uses a pointer to reference the name string in the payload of that *Interest* and an unsigned integer to record the name length. Note that an iStack PIT entry keeps one and only one *Interest*. In case that more *Interests* with the same name arrive before the corresponding *Data* is retrieved, PIT records the incoming *FaceIDs* of them in the existing PIT entry and updates the entry life time. Then the duplicate *Interests* are dropped.



(a) Compact hash-based data structure for PIT

(b) Compact hash-based data structure for CS

Figure 3: Compact forwarding data structures

When a *Data* packet arrives, PIT performs LPM in two steps. First, it queries the full *Data* name to find the matched entry and forwards it to the incoming *faces*. Then, each proper prefix of the *Data* name are queried for those*CanBePrefix*-flag marked entries. Note that forwarding *Data* to multiple *faces* is done by the kernel network function *skb_clone* which increases the packet reference count and only copies the *sk_buff* structure for management. Hence, iStack multicast has no additional overhead for duplicate *Data* packets.

CS needs to support both of ENM and all sub-name matching (ASNM) [24], which means all the proper prefixes of a *Data* name need to be indexable. CS caches *Data* packets in kernel directly and the compact structure of CS is shown in Fig. 3(b). In order to extend supporting from ENM to ASNM, CS creates a series of entries for a cached *Data* packet. Each of the entries uses a pointer to reference the name string in

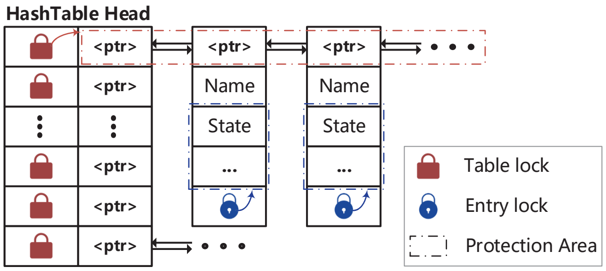


Figure 4: Fast two-level locks structure

the payload of that *Data* and an unsigned integer to record the length of the whole name or one of its proper prefixes.

With this design, CS in iStack supports NDN name discovery: an *Interest* with *CanBePrefix* flag can be satisfied by cached *Data* with an extended name. Moreover, entries in CS may be evicted based on the replacement policies. In order to evict the series of entries of a cached *Data* packet efficiently, the entry of CS is designed with another dual pointer field which forms a doubly linked list associating the series of entries for a *Data* packet. With this, evicting a cached *Data* packet from CS is simple: find the entries with the specific name and then remove all the entries in its doubly linked list and finally release the referenced packet.

The compact hash-based data structures in iStack take zero memory copy for names and prefixes and have no packet duplication, which reduce the memory consumption. Moreover, as both of the data structures and packets are in kernel space, there is also no memory copy and context switch overhead between kernel and user space.

## 4.2 Fast Locks for Multi-Threading Safety

There are forwarders which choose sharded data structures to improve multi-thread performance, whereas keeping private instance of PIT/CS per thread may break down NDN name discovery. The proposed solution is introducing *PIT token* which is the hash value of the entire name or a fixed-length prefix of *Interest* name and is carried hop-by-hop (NDN-DPDK, YaNFD [20, 23]). As a general-purpose network stack, iStack avoids involving dependency on under-layer protocols. Hence, the three key NDN data structures in iStack are shared by parallel kernel threads. In order to ensure multi-threading safety while achieving high performance, we carry out fast two-level locks structure which reduces locking overhead.

As illustrated in Fig. 4, there are two-level locks for iStack compact hash-based data structures, the table locks and the entry locks. Instead of competing for a single lock, each bucket of the separate chaining hash table is equipped with an individual table lock. Since parallel operations are naturally distributed to different chains by hashing, the blocking overhead is limited to the concurrent operations on entries with a same name and with hash collisions.

As name lookup without table modification takes a significant proportion of name-based forwarding processing, iStack adopts readers-writers locks as the table locks [25]. The critical area of reader locks is a traversal of a hash table chain and writer locks protects for entry insertion and removal. To minimize the impact on performance, the critical area only covers direct operations on the corresponding hash table chain while other processing including entry initialization and memory free are done before and after the critical area, respectively.

The entry lock is used for safely updating the state of an entry such as adding incoming faces to an existing PIT entry. Due to most of the entry related operations are mutually exclusive and likely to be finished in very short period, iStack takes spinlocks as the entry locks.

The fast two-level locks structure reduces performance degradation of parallel processing while ensures multi-threading safety with little cost of memory consumption. iStack only equips per-bucket table locks for three tables: PIT, CS, and FIB. Given the assumption that setting those table heads length to 65536, the extra memory for table locks is 1.5 MB in total. The entry locks are equipped on all table entries. Nevertheless, PIT entries are temporarily pending in the table. For CS and FIB, the size of entry lock is negligible compared with that of cached *Data* and stored routable prefix strings.

## 4.3 Miscellaneous Design

### 4.3.1 In-Network Caching

As iStack is an in-kernel stack, it naturally takes CS into the kernel. For generality, two issues should be considered, namely the kernel-space memory consumption for storage and the capacity of CS. For the former one, iStack provides two parameters for management. They are maximum number of entries and maximum size. The two limitations take effect simultaneously. The maximum size is to limit the kernel-space memory consumption. As CS in iStack creates entries for each proper prefix of a *Data* packet, the maximum number of entries is used to prevent from a large amount of entries for *Data* with an extremely long name overwhelming CS and decreasing its performance. For the latter, we design hierarchical structure for CS. The in-kernel part is used for *Data* with the most popular prefixes. The out-kernel part utilizes hard disk space to provide vast size of storage. We implement in-kernel CS with configurable parameters in our prototypes and leave the hierarchical CS implementation as future work.

### 4.3.2 Loadable Kernel Module

There are two ways to implement iStack in the Linux kernel: i) add iStack codes to the kernel source tree and recompile the kernel or ii) add iStack as kernel loadable module to the running kernel. As an in-developing novel network stack, the main part of iStack is implemented as a loadable kernel module (LKM). But some socket-like APIs remain to be completed. With the current socket APIs, the parameter of *bind* is limited to 128 bytes long, which limits the maximum binding prefix length. Besides, with Berkeley socket API, the programming model of iStack includes merely *send* and *recv*. We also design information-centric programming model and implement new socket-like APIs for iStack to be published. Practically, there is a static size array for the system call table and adding new system calls needs to modify the architecture-specific table and recompile the kernel.

In principle, NICs provide *receive side scaling* (RSS) on hardware for dispatching incoming packets to different CPU cores. Nevertheless, the typical filter used in RSS is a 4-tuple hash over IP addresses and TCP ports of a packet. Based on our observations from implementing, most of the commercial off-the-shelf NICs do not perform RSS for packets with unknown ether types and they just send all packets with unknown type to a single RX queue. Hence, we implement a software-based dispatcher to enable multi-threading packet processing in iStack with two dispatching strategies: dispatching based on hashing entire names and dispatching randomly.

#### 4.3.3 Reliability and Security

Although the raw type of socket provides maximum flexibility for applications, a general-purpose network stack should provide interfaces with different levels of granularity in terms of reliability. Hence, besides the basic socket type SOCK_RAW and SOCK_DGRAM, we also implement the type SOCK_SEQPACKET in iStack. SOCK_RAW only provides a basic interface for applications to send and receive packets. SOCK_DGRAM provides packet encapsulating and decapsulating functions with which applications can focus on names and content rather than manually deal with packet format. SOCK_SEQPACKET provides more reliability. In the *sock* part of this type of socket, we implement slice-window flow control, retransmission, and name-based reordering. To save space, more details are not elaborated here.

It is worth noting that iStack represents an in-kernel design and prototype for NDN. Its primary value lies in its robust framework. It is designed to offer fundamental NDN functionalities and facilitate integration of additional features and strategies, rather than presenting a finalized production-ready solution. Given that ICN/NDN fundamentally transforms the network communication paradigm from host-dependent to host-independent, and introduces state into the network, the required transport abstractions may differ significantly from the established end-to-end protocols in IP realm. Ongoing research has been dedicated to transport-layer services [26], congestion control [27–29], and etc. Furthermore, in NDN, each *Data* carries a unique name, resulting in distinct transport service requirements compared to traditional models. Consequently, iStack currently incorporates essential transport functionalities for application development, rather than fully piggy-back established end-to-end transport abstractions like TCP/QUIC. We will keep the evolution of iStack.

NDN secures data directly by requiring data producers to cryptographically sign every *Data* packet [1]. There are also literatures investigating security mechanisms for different scenarios [30–32]. Based on the basic *Interest/Data* communication, users can retrieve both content and the corresponding keys and perform verifications. We leave the security operations to user space and plan to carry out function libraries for the convenience of applications in the future.

## 5 Evaluation and Discussion

### 5.1 Implementation and Experimentation

We have implemented iStack in the Linux kernel 4.14 and 4.19 and tested on PCs, servers, Raspberry Pies (4B+), and edge routers. We also implemented several applications over iStack, including file transfer, streaming video player, and etc. Our prior test shows that iStack can achieve 1 Gbps line speed on laptops and low-end devices like Raspberry Pies. We built a pure NDN local area network (LAN) with iStack and applications including file transfer and video player ran in the LAN successfully. In this paper, our evaluation focus on two aspects: (i) network performance benchmark and (ii) application-level performance for real edge.

For the network performance measurement, we ran a provider-side file transfer application on a server equipped with dual 6 cores CPUs (hyper to 24 threads in total) and connected it to a packet (*Interest*) generator. As the hyper thread of a core is abstracted as a processor in the Linux kernel, we interchangeably use the terms *core* and *thread* in the following paper. We evaluated the maximum network throughput with different working threads and compared the delay of cached *Data* response between in-kernel and out-kernel CS.

To evaluate application-level performance for real edge, we deployed multiple consumer-side file transfer applications to request different files from the server. All applications use the same format of content names and an example is */Net-A/prdc-01/0/PubFiles/ABCD/1* which consists of three distinct parts:

1) A prefix of the provider (*/Net-A/prdc-01*).
2) A provider application defined prefix part (*/0/PubFiles*).
3) A part of application-specific information including a segment number (*/ABCD/1*, */ABCD/2*).

As the consumer applications run with socket APIs in user space, this scenario can obtain "goodput", application-layer throughput, and monitor CPU usage and memory consumption as the applications are running. For fair comparison with existing forwarders, the throughput and the forwarding rate in the evaluation only accounts for *Data* packets. Hence the total packet forwarding rate (including both *Interest* and *Data*) is at least twice[3] of the results in this paper.

[3] There is potential of *Interest* packet loss and retransmission which makes the total packet forwarding rate may higher than twice of the reported results.

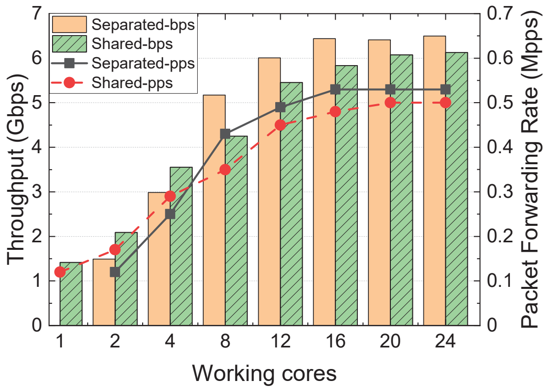


Figure 5: Throughput with different processor affinity settings

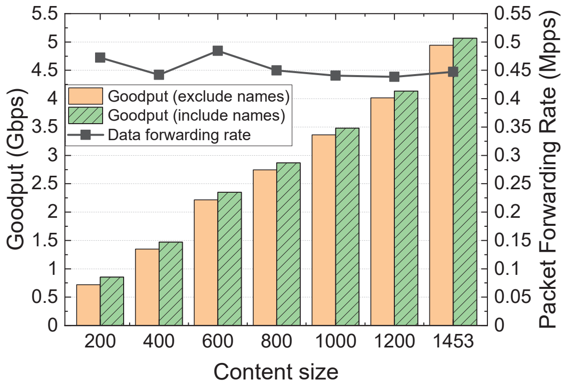


Figure 6: Goodput with different content-payload sizes

Table 1: Delay comparison between in/out-kernel CS

| CS size | Avg Delay for Cache Hit (*us*) | | Reduction Ratio |
|---|---|---|---|
| | **In-Kernel** | Out-Kernel | |
| 10 MB | 3.85 | 7.14 | 46.08% |
| 100 MB | 4.17 | 7.69 | 45.77% |
| 1 GB | **4.55** | 8.33 | 45.38% |

## 5.2 Throughput and Delay

As described in Sec. 4.3, the commercial off-the-shelf NICs do not perform RSS for the packets with unknown ether types. All these packets are passed to a single RX queue and handled by one thread. Hence, in our current implementation, there is only one thread running the dispatcher which distributes all incoming packets to different CPU cores. In the experiments, we used a random dispatching strategy, in which packets are dispatched to different kernel threads randomly and evenly.

Fig. 5 shows that both of the throughput and the *Data* forwarding rate grow almost linearly up to 8 or 12 forwarding cores. After that, the growing slows down and finally achieves 6.50 and 6.13 Gbps with 24 cores. We also evaluated the performance influence of whether using a separated core for the dispatcher. Note in Fig. 5 that the *separated-** represents that the dispatcher never dispatches packets to the core it is running on. The core (separated or shared) for dispatcher is also counted in the number of working cores, hence the result of *separated-** starts from two.

In Fig. 5, we observe that the performance of shared core is higher than that of the separated core with up to 4 cores. This is because the software based dispatcher has not achieved its performance bottleneck yet and separating a core for it means decreasing the number of cores for packet forwarding and finally leads to performance decreasing. However, when there are more than 8 cores, the forwarding throughput is relatively sufficient and the single core dispatcher becomes the bottleneck. In this case, sharing a core decreases the throughput of the dispatcher and finally limits the maximum throughput of iStack. Hence, with a separated core for the dispatcher, the input bottleneck comes later (12 instead of 4 cores as shown in Fig. 5) and the maximum throughput of iStack achieves 6.50 Gbps. Given that the number of cores for packet ingress depends on how many NICs are installed on the system and the hardware RSS of the NICs, we plan to eliminate the dispatching bottleneck in the future work.

In order to evaluate the overhead reduction of taking CS into kernel space, we also implemented an out-kernel version which communicates with iStack through Netlink sockets. CS in user space needs to copy packet buffer from/to kernel space via system calls for *Data* insertion/retrieval. It has significant overhead comparing with the in-kernel version. Table. 1 presents the forwarding delay with capacity varying from 10 K to 1M. The forwarding delay increases slightly with capacity as larger capacity means higher workload. Table. 1 also shows that the in-kernel CS of iStack reduces over 45% forwarding delay for the case of cache hit.

## 5.3 Application Performance

Fig. 6 shows that varying content size, the goodput grows up linearly from 762.4 Mbps to 5.07 Gbps. Meanwhile the *Data* forwarding rate keeps at about 0.45 Mpps, which proves that the payload length has limited impact on iStack forwarding rate but is important for application goodput. Due to the applications imports additional overhead, the measured maximum *Data* forwarding rate (0.48 Mpps) is slightly lower than that measured with the *Interest* generator, which achieves 0.53 Mpps. Another observation is that comparing with an iStack router, an edge host running applications with iStack can achieve 90.5% of the maximum performance.

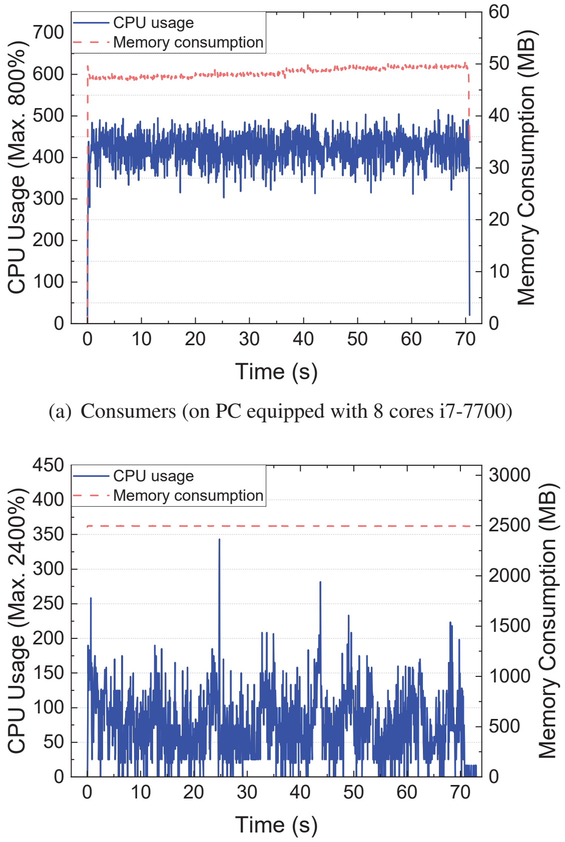
In Fig. 6, the differences between the results of excluding and including names are similar with different content size, whereas it is worth noting that with the content size growing, the proportion of the difference decreases. Hence, if the name does not carry application information, using larger content size and shorter name is more efficient.

## 5.4 CPU Usage and Memory Consumption

As iStack targets to deploy on edge hosts and routers for general purpose, we evaluated CPU usage and memory consumption of it. The result is measured from the entire system instead of some specific applications. Hence it can better indicate the resource overhead of an end host. In this measurement CS was disabled on end hosts to avoid its influence on memory consumption. All consumer applications continuously ran for 30 times after a warm-up running.

The result is illustrated in Fig. 7. Note that the two hosts were equipped with different cores of CPU and hence the celling of CPU usage in Fig. 7(a) and Fig. 7(b) are 800% and 2400%, respectively. The CPU usage of consumer applications is relatively high, because the consumer-side applications used sockets with the type SOCK_SEQPACKET, each of which ran a TCP-like sliding window algorithm and took care of retransmission based on timeout. The average goodput in this measurement is about 4.97 Gbps. Fig. 7(b) shows that such scenario is an easy task for providers as most of the time the CPU usage is under 200% (maximum 343.3%).

We measured initial system memory consumption before running consumer/provider applications and the result in Fig. 7 is the real-time measurement minus the initial value. Fig. 7(a) shows that the total memory consumption with 15 consumer applications concurrently running is less than 50 MBytes. The memory consumption of both the server and the host vary slightly, which indicates that excepting CS, iStack has little memory consumption and is suitable for edge usage.



(a) Consumers (on PC equipped with 8 cores i7-7700)

(b) Provider (on server equipped with 24 cores Xeon E5-2620)

Figure 7: CPU usage and memory consumption

## 5.5 Discussion

Our evaluations show that iStack is a promising in-kernel network stack for general purpose. iStack achieves up to 6.50 Gbps throughput and meanwhile keeps low CPU usage and memory consumption. A comparison among iStack and other NDN forwarders is shown in Table 2. NDN-DPDK has the highest throughput. Nevertheless, DPDK-based solutions have limitation of hardware depending and polling-based I/O is not suitable for edge usages, especially for low-end devices.

Among all event-driven forwarders, iStack outperforms in terms of the throughput/goodput with a factor of 6.26x-16.25x. Besides, our in-kernel CS eliminates the overhead of memory copy between kernel space and user space for caching and retrieving *Data*, respectively. The evaluation result shows that in-kernel caching reduces upto 46.08% forwarding latency in case of cache hit. It is worth noting that currently iStack has not been fully optimized and the single-thread dispatcher also limits its multi-threading performance.

iStack is not another NDN forwarder but an ICN protocol stack. The other implementations in Table 2 are all user-space forwarders. In contrast, iStack is designed and implemented integrated with OS kernel and our prototypes can be compiled with the source code of Linux kernel. iStack provides system-level NDN functionality supporting. Applications can access the named network through OS network sockets directly.

## 5.6 Lessons Learned

Our experiences constructing iStack have revealed important insights into putting NDN into OS kernel. First, we have discovered that from a system perspective, the namespace of NDN should be carefully organized. Allowing applications to register arbitrary prefixes into FIB results in unnecessary overhead of the in-kernel network stack. Hence, iStack classifies prefixes into routable ones and application-specific ones. In the network layer of iStack, FIB only deals with routable prefixes. Applications can binding more specific prefixes which are handled by BPT. Hence, iStack decouples local demultiplexing from network-layer routing and forwarding, while keeping NDN name-based communication model.

Second, in contrast to IP's stateless forwarding design, NDN maintains per-packet state at the network layer. Crafting a robust stateful network stack introduces additional complexity. One challenge arises from the potential for network state to create blockages with certain OS interfaces. Events such as socket closed or network cable unplugged can occur while

Table 2: Comparison with other forwarders

| Forwarder/Stack | Type | I/O Driven | Programming Model | Net. Throughput | App. Goodput | Language |
|---|---|---|---|---|---|---|
| NFD [33] | App-level | Event | Specific API | 0.4 Gbps | N/A | C++ |
| NFD-Opt. [33] | ditto | ditto | ditto | 0.9 Gbps | N/A | ditto |
| YaNFD [23] | App-level | Event | Specific API | N/A | 0.81 Gbps | Go |
| MW-NFD [22] | App-level | Polling | Specific API | 4.26 Gbps | N/A | C++ |
| NDN-DPDK [20] | Bypass kernel | Polling | Specific API | N/A | 22 Gbps* | C & Go |
| **iStack** | **In kernel** | **Event** | **Network socket** | **6.50 Gbps** | **5.07 Gbps** | **C** |

* NDN-DPDK achieves 108 Gbps throughput with Jumbo frames (8000 B). For fairness, we list results that are based on standard Ethernet frames (1500 B).

corresponding Interests are still pending in PIT. iStack mitigates such potential blockages through its thread-safe face system, elaborated in Section 3.3.3. Another consideration is the need for judicious management of overhead. iStack employs streamlined forwarding data structures and avoids packet duplication to minimize the overhead with stateful network processing. Additionally, iStack adopts an interruption-based packet reception approach rather than a polling-based one. Note that in cases of extremely high throughput demands, such as those in data-center server environments, a polling approach like DPDK might be a viable alternative. However, such solutions often lead to elevated CPU usage even during idle periods, potentially monopolizing resources required for non-networking processes. Consequently, iStack ultimately embraces an interruption-based approach.

Third, as NDN enables in-network caching, an in-kernel network stack for it involves storage into the OS kernel. As mentioned in Sec. 4.3.1, taking storage into kernel should consider two issues: memory consumption and capacity. As a packet certainly exists in the network stack before it is cached, an in-kernel stack can directly cache *Data* packets in kernel-space memory for fast response. However, consuming too much memory for fast in-network caching hinders other tasks running on the host. Hence, iStack carefully limits the in-kernel memory usage for CS and utilizes external storage to extend CS capacity. The Linux kernel disapproves operating files directly and from our developing experience, there are two promising methods to achieve this. One is building a user-space CS and exchange packets with in-kernel stack through socket interfaces. Another one is designing a new storage mode for the rising in-network caching involved by NDN.

Network stacks in current OS is built to support the existing node-centric networking model, with the intrinsic notion of socket-connection-interface mapping. An in-kernel stack communicates with user-space applications through system calls. Specifically for networking, there are socket-based system calls. Although presently the socket API of iStack is able to support essential NDN communication functionalities, it is a send/receive programming model and lacks the aware of symmetrical flow, the unique naming of each packet, and other features included by NDN. Hence, designing an information-centric programming model and the corresponding API for NDN applications is an attractive research topic.

## 6 Conclusion and Future Work

This paper describes iStack, a general-purpose information-centric network stack, taking NDN into OS kernel, for practical deployment and usage. Our work contains both intellectual and practical contributions. On the intellectual side, iStack shows how to integrate a stateful, entirely name-based protocol stack into OS kernel and be compatible with the socket mechanism. On the practical side, iStack demonstrates itself as a practical kernel-level protocol stack for NDN.

We implement iStack prototypes on different four kinds of platforms from high performance servers to low-end devices. Evaluation shows that iStack achieves 6.50 Gbps throughput and meanwhile keeps reasonable CPU usage and memory consumption, which indicates that iStack is sufficient for the requirements of general usage and practical deployment.

To the best of our knowledge, iStack is the first in-kernel stack for NDN and achieves both generality and high performance. In terms of generality, iStack supports multiple applications that have different purposes and requirements with network socket mechanism implemented by operating systems directly. In terms of performance, iStack outperforms the NDN-testbed forwarder by a factor of 16.25x and even faster than the polling-based MW-NFD by a factor of 1.52x.

Our future work includes the following aspects. First, we also design a series of information-centric socket primitives and the corresponding programming models for iStack to be published. Second, we are working on porting iStack to other platforms including Andriod and Windows. Third, with further developing and extensive testing in more cases, we will provide a well-polished version for community as well as long term support and evolution in the near future[4]. We believe that iStack is not just another forwarder for NDN, but a step forward for the development of ICN.

## Acknowledgments

This work was supported by the National Natural Science Foundation of China under grant No. 92067203. We would like to thank anonymous reviewers and our shepherd James Mickens for thoughtful feedback.

[4] We provide long term support through the website: *name-ip.cn*

## References

[1] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, KC Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. Named Data Networking. *ACM SIGCOMM Computer Communication Review*, 44(3):66–73, 2014.

[2] Yating Yang, Tian Song, and Beichuan Zhang. Open-Cache: A lightweight regional cache collaboration approach in hierarchical-named ICN. *Computer Communications*, 144:89–99, 2019.

[3] Tian Pan, Xingchen Lin, Enge Song, Cheng Xu, Jiao Zhang, Hao Li, Jianhui Lv, Tao Huang, Bin Liu, and Beichuan Zhang. NB-Cache: Non-Blocking In-Network Caching for High-Performance Content Routers. *IEEE/ACM Transactions on Networking*, 29(5):1976–1989, 2021.

[4] Zhiyi Zhang, Yingdi Yu, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang. An Overview of Security Support in Named Data Networking. *IEEE Communications Magazine*, 56(11):62–68, 2018.

[5] Tian Song, Yating Yang, and Tianlong Li. Rethinking Caching Security of Information-Centric Networking: A System Recovery Perspective. *IEEE Communications Magazine*, 57(10):104–110, 2019.

[6] Yu Zhang, Hongli Zhang, and Lixia Zhang. Kite: A Mobility Support Scheme for NDN. In *Proceedings of the 1st ACM Conference on Information-centric Networking*, pages 179–180, 2014.

[7] Xavier Mwangi and Karen Sollins. MNDN: Scalable Mobility Support in Named Data Networking. In *Proceedings of the 5th ACM Conference on Information-Centric Networking*, pages 117–124, 2018.

[8] Peter Kietzmann, Cenk Gündoğan, Thomas C Schmidt, Oliver Hahm, and Matthias Wählisch. The Need for a Name to MAC Address Mapping in NDN: Towards Quantifying the Resource Gain. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, pages 36–42, 2017.

[9] Tianlong Li, Tian Song, Yating Yang, and Jike Yang. iCast: Dynamic Information-Centric Cross-Layer Multicast for Wireless Edge Network. In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, pages 137–147, 2022.

[10] Ran Zhu, Tianlong Li, and Tian Song. iGate: NDN Gateway for Tunneling over IP World. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2021.

[11] Safdar Hussain Bouk, Syed Hassan Ahmed, Dongkyun Kim, and Houbing Song. Named-Data-Networking-Based ITS for Smart Cities. *IEEE Communications Magazine*, 55(1):105–111, 2017.

[12] Yating Yang and Tian Song. Local Name Translation for Succinct Communication Towards Named Data Networking of Things. *IEEE Communications Letters*, 22(12):2551–2554, 2018.

[13] Yating Yang, Tian Song, Weijia Yuan, and Jianping An. Towards reliable and efficient data retrieving in ICN-based satellite networks. *Journal of Network and Computer Applications*, 179:102982, 2021.

[14] Chavoosh Ghasemi, Hamed Yousefi, and Beichuan Zhang. Internet-Scale Video Streaming over NDN. *IEEE Network*, 35(5):174–180, 2021.

[15] NFD: Named Data Networking Forwarding Daemon, 2014. https://named-data.net/doc/NFD/current.

[16] Alexander Afanasyev, Junxiao Shi, Beichuan Zhang, Lixia Zhang, Ilya Moiseenko, Yingdi Yu, Wentao Shang, Yi Huang, Jerald Paul Abraham, Steve DiBenedetto, et al. NFD Developer's Guide. *Technical Report NDN-0021, NDN*, 2014.

[17] NDN Project. NDN Testbed, 2017. http://ndndemo.arl.wustl.edu/.

[18] Wentao Shang, Alex Afanasyev, and Lixia Zhang. The Design and Implementation of the NDN Protocol Stack for RIOT-OS. In *2016 IEEE Globecom Workshops*, pages 1–6. IEEE, 2016.

[19] Zhiyi Zhang, Edward Lu, Yanbiao Li, Lixia Zhang, Tianyuan Yu, Davide Pesavento, Junxiao Shi, and Lotfi Benmohamed. NDNoT: A Framework for Named Data Network of Things. In *Proceedings of the 5th ACM Conference on Information-Centric Networking*, pages 200–201, 2018.

[20] Junxiao Shi, Davide Pesavento, and Lotfi Benmohamed. NDN-DPDK: NDN Forwarding at 100 Gbps on Commodity Hardware. In *Proceedings of the 7th ACM Conference on Information-Centric Networking*, pages 30–40, 2020.

[21] DPDK Project. Data Plane Development Kit (DPDK), 2015. http://www.dpdk.org.

[22] Sung Hyuk Byun, Jongseok Lee, Dong Myung Sul, and Namseok Ko. Multi-Worker NFD: an NFD-compatible High-speed NDN Forwarder. In *Proceedings of the 7th ACM Conference on Information-Centric Networking*, pages 166–168, 2020.

[23] Eric Newberry, Xinyu Ma, and Lixia Zhang. YaNFD: Yet another Named Data Networking Forwarding Daemon. In *Proceedings of the 8th ACM Conference on Information-Centric Networking*, pages 30–41, 2021.

[24] Zhuo Li, Yaping Xu, Beichuan Zhang, Liu Yan, and Kaihua Liu. Packet Forwarding in Named Data Networking Requirements and Survey of Solutions. *IEEE Communications Surveys & Tutorials*, 21(2):1950–1987, 2018.

[25] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. Concurrent Control with "Readers" and "Writers". *Communications of the ACM*, 14(10):667–668, 1971.

[26] Mauro Sardara, Luca Muscariello, and Alberto Compagno. A Transport Layer and Socket API for (h)ICN: Design, Implementation and Performance Analysis. In *Proceedings of the 5th ACM Conference on Information-centric Networking*, pages 137–147, 2018.

[27] Giovanna Carofiglio, Massimo Gallo, and Luca Muscariello. Joint Hop-by-hop and Receiver-Driven Interest Control Protocol for Content-Centric Networks. *ACM SIGCOMM Computer Communication Review*, 42(4):491–496, 2012.

[28] Milad Mahdian, Somaya Arianfar, Jim Gibson, and Dave Oran. MIRCC: Multipath-aware ICN Rate-based Congestion Control. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*, pages 1–10, 2016.

[29] Sichen Song and Lixia Zhang. Effective NDN Congestion Control Based on Queue Size Feedback. In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, pages 11–21, 2022.

[30] Yanbiao Li, Zhiyi Zhang, Xin Wang, Edward Lu, Dafang Zhang, and Lixia Zhang. A Secure Sign-On Protocol for Smart Homes over Named Data Networking. *IEEE Communications Magazine*, 57(7):62–68, 2019.

[31] Tianyuan Yu, Hongcheng Xie, Siqi Liu, Xinyu Ma, Xiaohua Jia, and Lixia Zhang. CertRevoke: A Certificate Revocation Framework for Named Data Networking. In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, pages 80–90, 2022.

[32] Davide Pesavento, Junxiao Shi, Kerry McKay, and Lotfi Benmohamed. PION: Password-based IoT Onboarding Over Named Data Networking. In *ICC 2022-IEEE International Conference on Communications*, pages 1070–1075. IEEE, 2022.

[33] Yaoqing Liu, Anthony Dowling, and Lauren Huie. Benchmarking Network Performance in Named Data Networking (NDN). In *2020 29th Wireless and Optical Communications Conference*, pages 1–6. IEEE, 2020.