Network-Agnostic Systems in a Networked World

Glenn Scott
Palo Alto Research Center, Palo Alto, CA 94304
Glenn.Scott@parc.com

Christopher A. Wood
University of California, Irvine, CA 92617
woodc1@uci.edu

ABSTRACT

Content-Centric Networking (CCN) is an emerging networking paradigm in which named content, rather than named network interfaces or hosts, are treated as first-class entities representing the end-points of communication in a network. This abstraction decouples data from its source to enable multiple, simultaneous providers of data within the network, which fosters optimized bandwidth consumption, improved availability, reduced latency, and efficient utilization of multiple network interfaces concurrently. In CCN, this abstraction layer is provided by a lightweight API called the CCN Portal. The CCN Portal API enables communication and interaction via atomic request and content response messages, though applications may often require more robust abstractions. To this end, we introduce the CCN Assembly Framework (AF). The AF provides a Create Read, Update, and Delete (CRUD) interface for reading and manipulating data "in the network." We then show how to use the AF API with the UNIX file I/O API, thereby placing file I/O and network communication under a single, standard API. The ultimate goal is to merge remote and local data to make applications *network agnostic*, which is extremely compelling in society's increasingly network-oriented ecosystem.

1. INTRODUCTION

The Internet is a *de facto* public utility used by a significant fraction of humankind who rely on it for numerous daily activities. Despite unparalleled success and unexpected longevity, applications and the traffic they generate are changing. Low-bandwidth interactive (e.g., remote log-in) and store-and-forward (e.g., email) traffic that dominated the early Internet is increasingly superseded by the web-dominated Internet of today. This change has been produced applications with a significantly larger emphasis on content, such as social media applications and video streaming services. Together, they changed the very nature by which the Internet is used; point-to-point telephony-like conversations are no longer the norm. Rather, the Internet is used by consumers who want content delivered rapidly and securely, regardless of where it comes from.

The suite of protocols that exist in the current TCP/IP model have not properly satisfied these new use cases well. All of the abstractions provided by interfaces to these protocols are inherently location-dependent and provide little more than packet transmission and receipt functionality. In an increasingly content-centric world, abstractions for content are needed to build modern applications and services. Therefore, in addition to reconsidering the Internet architecture itself, we must also consider the protocols and abstractions by which applications make use of it.

Content-Centric Networking (CCN) is an approach to (inter-)networking designed for efficient, secure, and scalable content distribution [16, 13, 18]. In CCN, named content – rather than named interfaces or hosts – are treated as first-class entities. To obtain content, a consumer issues a request, called an interest, with the name of the desired content to the network. This request is autonomously routed to the location where the content is stored and/or produced. By decoupling data from its source, content can be cached within the network to optimize bandwidth use, reduce latency, and enable effective utilization of multiple network interfaces simultaneously. For authenticity reasons, each piece of named content must be signed[1] by its producer, also known as a publisher. This allows trust in content to be decoupled from trust in an entity – a host or router – that might store and/or disseminate that content.

Aside from the aforementioned performance improvements, the content-centric approach to networking has one key benefit: an abstraction layer, in the form of requests and the corresponding content, is introduced between applications and the location where content is stored. We present a detailed description of this API, called the CCN Portal, and discuss several cases in which it is used. The Portal API enables applications to be constructed in a location-agnostic way using the atomic interest and content object abstractions. This abstraction has profound implications on future applications and systems. For example, it can be used to simplify everything from media streaming applications such as Netflix to the deployments of the Internet of Things (IoT).

Although highly useful, the CCN Portal named-content abstraction can be further extended to enable further independence from the network. Although highly useful, the CCN Portal named-content abstraction can be extended to enable further independence from the network. Many modern systems and applications rely on files as a high-level abstractions upon which other data access mechanisms are implemented. For example, video streams, data bases, messages, and key-value stores. Interests and content objects serve as a vehicle for transferring *raw* data; they cannot be treated as these various abstractions without another layer of indirection. This problem is exacerbated by the fact that content objects may carry encrypted data, and upper-level applications may not wish to perform decryption manually.

These realizations lead us to the main contribution of this

[1]Signatures are not mandated with the advent of self-certifying names. See [1] for more details.

work: the Named Data Interface (NDI). This component builds on top of the Portal API to provide a Create, Read, Update, Delete (CRUD) API for applications to read and manipulate network data. Internally, it interfaces with the operating system processes and remote services needed to compose content objects into these abstractions, decrypting their contents if necessary. To highlight the efficacy of the NDI, we show how to implement the UNIX file I/O API using the NDI API, thereby placing file I/O and network communication under a single, standard API. The net result is a layer of abstraction upon which completely *network-agnostic* applications can be built.

The rest of this paper is organized as follows. Section 2 presents the development history of today's Internet and how it led to CCN. It also provides a relevant discussion of the CCN architecture and core communication protocols upon which the rest of the paper is founded. Section 3 presents the CCN Portal API, which is then expanded to the NDI API in Section 4. Section 5 shows how to use merge remote and local data using the NDI API. Finally, we conclude a discussion of related work in Section .

2. PRELIMINARIES

The focus of this work is on the evolutionary progression of the applications and the Internet. To capture the development trajectory, we must first describe the history of networking technologies, including both the underlying networking fabric (e.g., circuit- vs. packet-switching networks) and protocol suites (e.g., NCP, TCP/IP, and CCN). This section may be skipped without loss of continuity should the reader already possess a firm understanding of these technologies.

2.1 Circuits to Packets: The History of the Internet and TCP/IP

The early public switched telephone system is an ancestor of today's Internet. Telephone calls were made by establishing a direct connection between two terminals. In order to establish a call, a user would dial the operator and ask to be connected to the intended destination. This involved physically establishing a circuit between the two terminals. While this point-to-point communication paradigm makes it easy to support minimal quality of services guarantees, it does not scale.

Leonard Kleinrock introduced the notion of packet-switching communication in 1961 [17], and later went to DARPA to work on the first incarnation of a packet-switched network – ARPANET – standardized in 1967 and deployed in 1969 [25]. In December 1970, the Networking Working Group, led by Steve Crocker, finalized the Networking Control Protocol (NCP), which was a host-to-host protocol that ran on top of ARPANET. Using NCP, applications could now be developed on top of ARPANET. All applications and uses were, however, limited to noncommercial endeavors. In this original incarnation, there were only three layers in the networking stack: the physical layer, supporting various mediums such as radio and satellite networks, transport layer, supporting packet movement by NCP, and application layer.

As the number of nodes and types of networks connected ARPANET grew, it became clear that dealing with heterogeneous systems would be an issue. To remedy this problem, Kahn introduced the idea of open-architecture network, a concept still embodied by today's Internet. The Transport Control Protocol (TCP) emerged from Cerf and Kahn [5] in 1974 as a result of decoupling the network details from the transport semantics in NCP. The original TCP implementation only supported virtual circuits for reliable packet delivery. Such QoS guarantees are not needed by all applications, though, which led to the split of the overloaded TCP into a pair of network and transport protocols responsible for individual packet forwarding and service features such as reliabile delivery, respectively. The User Datagram Protocol (UDP) was later introduced as a separate transport protocol for applications that did not require the rich feature set of TCP.

With this protocol suite in place to support a diverse set of applications and internetworking technologies, ARPANET was set to flourish. Robert Metcalf's development of MAC-layer Ethernet at PARC [21] paved the way for increasingly connection local area networks. As networks grew, the scalability of packet routing technologies came into question. To address this problem, hierarchical routing was addressed [20]. ARPANET was segregated into regions (i.e., autonomous systems) within which the Interior Gateway Protocol (IGP) was used to distribute routing information. Routes between these regions were connected via the Exterior Gateway Protocol (EGP). Both IGP and EGP are network-layer protocols, upon which TCP relied for establishing routes that make individual packets routable.

Building on TCP/IP and ARPANET, the U.S. National Science Foundation (NSF) launched NSFNET in 1986 for uses beyond research and education. The goal of the NSF was to create a network of networks, connected to ARPANET, with less restrictions on its use. Due to these limited restrictions, use of NSFNET grew rapidly, which required equipment and technology upgrades backed by commercial partners such as IBM [27, ?]. Its continued growth led to the decomission of ARPANET in 1990, and was itself subsequently decomissioned in 1995 in lieu of a new backbone service – today's Internet.

In 1995, the TCP/IP protocol suite consisted of five layers: physical, link (MAC) network, transport, and application [7]. These layers (models) encapsulate, or at least attempt to encapsulate, the scope of the direct links to other nodes on the local network, the internetworking range, end-to-end transport connections, and the scope of the software applications. Opponents of the TCP/IP model criticzed the fact that some layers in this protocol stack had cross-cutting concerns or were duplicated [3]. An alternative is the Open Systems Interconnection (OSI) model, which was developed and released in 1980 to enforce more strict layering [29]. It consists of a physical, data link, network, transport, session, presentation, and application layer. Its goal was to more precisely define boundaries between layers for increased modularity and better layers of abstraction.

Regardless of the model chosen, the general abstractions are the same. The application layer is presented with either (a) raw sockets to send and receive data froma a specific host in the network, or (b) a raw IP packet to send data to a host. Application layer protocols and APIs such as HTTP, FTP, and Telnet were developed as a further layer of abstraction for certain applications such as web browsers and file downloading clients, but these abstractions are not universally applicable. Some applications still need to use sockets and/or raw IP packets to send or receive data. Thus, the key abstractions in the current Internet model are based on *packets and their locations*.

2.2 Packets to Content: The Emergence of ICNs

Unlike TCP/IP, which focuses on end-points of communication and their names/addresses, request-based ICN architectures ([16]) focus on content by making it named, addressable, and routable within the network. CCN [1] is incarnation of a request-based ICN; we use the terms interchangeably going forward. A content name is composed of one or more variable-length components opaque to the network. Name component boundaries are explicitly delimited by "/" in the usual URI-like representation. For example, the name of a Youtube video content might be `lci:/youtube/parc/ccn-overview.mpeg`.

CCN communication adheres to the *pull* model whereby content is delivered to consumers only upon explicit request. There are two basic types of packets in CCN: interest and content messages. A consumer requests content by issuing an *interest* message. If an entity can "satisfy" a given interest, it returns a corresponding *content* object. Each content delivery in CCN must be strictly preceded by an interest. If content C with name n is received by a router with no pending interest for that name, the content is considered unsolicited and is discarded. Name matching in CCN is based on exact match. For example, an interest for `lci:/youtube/alice/video-749.avi` can only be satisfied by content named `lci:/youtube/alice/video-749.avi`.

CCN interests messages include, at a minimum, the name of the requested content. Additionally, they may carry a payload field that enables consumers to push data to producers along with the request. Conversely, CCN content objects include several fields. In this paper, we are only interested in the following three:

- `Name` – A sequence of explicit name components followed by an implicit digest (hash) component of the content recomputed at every hop. This effectively provides each content with a unique name and guarantees a match when provided in an interest.
- `Validator` – a public key signature, generated by the content producer, covering the entire object, including all explicit components of the name. The signature field also includes a reference (by name) to the public key needed to verify it.
- `Recommended Cache Time` – an optional time for the content objects to be cached supplied by producers.

There are three types of CCN entities (roles):[2]

- *Consumer* – an entity that issues an interest for content.
- *Producer* – an entity that produces and publishes (as well as signs) content.
- *Router* – an entity that routes interest packets and forwards corresponding content packets.

Each CCN entity, i.e., not just routers, maintains at least two components [4]:

[2]A physical entity, or host, can be both consumer and producer of content.

- *Content Store* (CS) – cache used for content caching and retrieval. From here on, we use the terms *CS* and *cache* interchangeably. Recall that the timeout of cached content is specified in the freshness field.
- *Forwarding Interest Base* (FIB) – table of name prefixes and corresponding outgoing interfaces. The FIB is used to route interests based on longest-prefix-matches of their names.
- *Pending Interest Table* (PIT) – table of outstanding (pending) interests and a set of corresponding incoming and outgoing interfaces.

Notably, a CCN entity may optionally maintain a Content Store (CS), a cache used for content caching and retrieval. From here on, we use the terms *CS* and *cache* interchangeably. The timeout of cached content is specified in the Recommended Cache Time field specified by the content producer.

When a router receives an interest for name n, and there are no pending interests for the same name in its PIT, it forwards the interest to the next hop(s) according to its FIB. For each forwarded interest, a router stores some amount of state information, including the name in the interest and the interface from which it arrived. However, if an interest for n arrives while there is already an entry for the same content name in the PIT, the router collapses the present interest, and any subsequent interests for n, storing only the interfaces on which it was received. When content is returned, the router forwards it to all of the corresponding incoming interfaces and deletes the corresponding PIT entry. Since no additional information is needed to deliver content, an interest does not carry any *source address*.

Upon receiving an interest, a router first checks its cache to see if it can satisfy this interest locally[3]. Producer-originated digital signatures allow consumers to authenticate content that is received, regardless of the entity that serves this content.

In addition to a name, CCN interest messages may also contain a KeyId and/or ContentObjectHash. The former is the hash of the public key used to verify the signature of the content, whereas the latter is the actual hash digest of the content object returned in response. These are useful for trust management purposes [12]. For example, the CCN protocol stipulates that if content objects satisfied from the cache of a router, then that router must either (a) verify the signature of the content or (b) check that the hash digest of the content is as expected. In case (a) the router checks the interest KeyId, if present, against the key used to verify the signature. In case (b) the router computes the hash of the content object and compares it against the provided hash digest. Equality in either case means the content can be trusted and forwarded downstream as a satisfying response to the corresponding interest.

2.3 Manifest-Based Content Retrieval

In CCN, manifests are a special type of content object which serve to *encapsulate* and *collate* groups of content objects with some set of associated metadata. Each content object contained in the manifest is coupled with its associated hash digest value. To illustrate their utility, consider

[3]This is why CCN lacks any notion of a *destination address* – content can be served by any CCN entity.

the following example. A consumer first issues an interest for a content objects and retrieves a manifest as the response. The manifest is then parsed to obtain content object names and hash values. Using these names and hash values, interest messages with non-null ContentObjectHash fields are created and issued to the network sequentially or simultaneously to obtain all content objects encapsulated by the manifest.

Aside from reducing the overhead involved in retrieving fragmented messages, manifests are useful in that they circumvent the need to individually sign and verify all content objects encapsulated within a manifest. Since each component content object is obtained via an interest with a hash digest, only the signature of the manifest needs to be verified to ascertain the trustworthiness of the content object. This benefit can be exploited by creating and leveraging manifest trees, in which the name of a component content object within a manifest is itself the name of another manifest. If the signature of the root manifest is verified and the associated verification key is trusted, then all content objects encapulsated by said manifest are also trusted by the properties of self-certifying names.

3. THE CCN PORTAL

The CCN transport stack is a set of components, each of which is focused on a specific task. It adheres to the chain-of-command pattern: each component has an outbound queue to move messages from the application toward the network, as well as an inbound queue to move messages from the network toward the application. Since each component focuses on a single purpose, the design promotes a strong separation of concerns between elements in the stack and enables a plug-and-play approach to stack compositions for different needs.

The transport stack is composed of a set of optional components and two required components: the API adaptor and the forwarder adaptor. Messages are pipelined through the transport stack components on their way between the upper-level API and the **forwarder**. The forwarder is the component that contains, updates, and uses the FIB, PIT, and CS when processing inbound and outbound messages. Typical stack instantiations include components for protocol management (e.g., flow control and chunking), outbound message signing, inbound message signature verification, packet format encoding and decoding, and communication with the local forwarder. This extensible and modular architecture supports a spectrum of possible stack configurations, including instances with only a very minimal set of features necessary for datagram-style messaging to instances with highly sophisticated components providing optional features such as an in-order stream of content objects with arbitrary rewind and fast-forward within the stream.

Interest and content objects are the primary elements of discourse in CCN[4]. Consumers issue interests for data and receive content objects in response; they do not need to know the details of the transport stack to communicate. The actual contents, semantics, and representation of both interests and content objects within the network stack are entirely dependent on its internal components. Consequently, a natural abstraction for interfacing with CCN is a single interface through which discrete interest and content object messages flow. In CCN, this interface is called the **Portal**. The Portal is minimal interface used to communicate with the **transport stack**, and therefore, the network. An instance of the Portal encapsulates a library of functions that perform operations between the **Portal API** and the transport stack.

[4]Control messages also exist, but as alternative means of sending data and information between parties.

The Portal API provides a simple interface to the transport stack allowing the application to compose, use, and maintain transport stacks and to perform message operations through the stack. It is a low-level API providing a simple interest and content object interface upon which applications and other interfaces are implemented. A single transport stack is associated with a single Portal API instance.

Within the context of a single system, all instantiated transport stack and Portal API instances are contained within the **transport framework**, which also has its own API (see Figure 1). The transport framework is responsible for supplying the runtime environment for a transport stack. It provides an interface for composing and decomposing transport stacks and interfaces with the necessary system resources (e.g., the operating system, communication interfaces, etc.). While an application may start multiple Portal instances with associated transport stacks, it will only have one transport framework to manage all stacks. Furthermore, each pair of Portal and transport stack instances will have its own encapsulated identity assigned to it by the application. This identity, which is associated with a set of public and private keys, is used when signing messages within the network stack that are issued by Portal.

4. THE NAMED DATA INTERFACE

Raw interests and content objects facilitate the development of network-agnostic applications. They do not, however, serve as rich abstractions upon which network-agnostic applications and systems can be built. This is a well-known lesson in the Unix world; while it is possible to perform basic I/O operations to read and write data to disk, files are a much simpler interface to data stored on disk. The UNIX File API provides an interface to files. The File type identifies a stream and contains the information needed to control it, including a pointer to its buffer, its position indicator and all its state indicators. Issues such as user permissions and concurrent modification are handled by the operating system; the File API operates *within the context* of the operating system and *under the auspices of a particular user.*

UNIX files are an extremely powerful abstraction upon which many different applications (e.g., grep, sed, awk), APIs (e.g., Apache Zookeeper [14]), and large systems (e.g., databases) have been built. We borrow from its success by introducing the CCN **Named Data Interface** (NDI), a centralized service that collates services such as network communication, access control and permissions, and concurrency synchronization to provide a single, simplistic, file-like API. The framework, shown in Figure 2, is an *extension of the transport stack* that uses external and local operating system services to perform a variety of useful functions and provide a lightweight file-like API to upper-level layers in the stack. The API follows the CRUD (create, read, update, delete) model with ACID guarantees. A high-level description of this API is summarized in Figure 1.

Note: Although the word "file" is used, it is not meant to

Table 1: The Named Data Interface API.

Function Signature	Description
`create(lciName, ACS)`	Make a file available with the given LCI encoded name and access information.
`buffer = read(lciName, offset=0, numbytes=0)`	Read data (of numbytes size) from the specified name starting at the given offset.
`update(lciName, buffer, offset=0)`	Update the file with the given name and offset (default 0) with the specified data.
`delete(lciName)`	Make a file that was previously available, unavailable.

imply persistent storage. Applications may create files that are only transmitted across the network and are consumed without being stored. However, the NDI does impose semantics that require a `create()` to provide a file that is not already in existence, and update to provide data to a file that must already exist. This permits cooperating applications – using the NDI – to establish what it means to create a file versus updating a file.

The NDI API empowers clients to create content with specific names, which ultimately determines the *manner and location* at which they are published. Clients can then open handles to files in read or write mode, and interact with them using the appropriate read or write functions. Figure 3 shows a sample use case in which a consumer reads content from a file provided by the producer. In this case, the producer is in charge of creating, updating (writing), and deleting the file as needed.

Note that, depending on the name, the file may not be stored locally. In this case, the producer creates files stored on remote machines using the same CRUD functions. However, depending on the relationship between the producer and remote system, some form of producer authentication protocol may be required in order to validate all CRUD requests on the remote system. This distributed behavior is shown in Figure 4.

4.1 Encapsulated Behavior

From the application's perspective, the NDI is meant as a more useful abstraction beyond interest and content objects. For example, applications may instantiate file instances using the framework and then read and write to them, much like any normal file. From the system's perspective, the NDI coordinates many working services to give further meaning to the atomic interest and content object messages provided by CCN. Examples of internalized behavior performed on behalf of the framework are elaborated upon in the following sections.

4.1.1 Object Reconstruction

As discussed in Section 2.3, manifests may be used to create collections of data. These manifests will either be created by the NDI on behalf of producers, or producers will manually create and inject them into the local NDI instance. If a client application wants to view the data represented by a manifest as a single source of data, e.g., a file, the NDI will (1) recursively obtain the data for all content objects pointed to by a root manifest, and (2) use a predefined policy for concatenating or merging these data partitions into a single piece of data over which clients can iterate. Additionally, the assembly manifest may decrypt the data in content objects referenced within a manifest, if it can obtain the appropriate cryptographic keys needed to do so. If content is encrypted, the AccessControlSpecification (ACS) field of the manifest will contain all of the information necessary to (1) identify the access policy controls enforced for the content and (2) retrieve the (possibly personalized) cryptographic keys tied to the client [19]. Access control is an application-layer policy decision that must be made by producers and then encoded into manifests and their ACS fields. Consequently, if a piece of content that is represented by a manifest is under access control, the producer must provide an ACS to specify the access policy.

4.1.2 Environment Interaction

As shown in Figure 2, the NDI uses information and services in its environment for operation. For example, the NDI interfaces with the local operating system to access physically attached devices. The name-based CRUD interface does not impede one from enabling device interaction via these simple commands. For example, a user may write content to a USB drive named `lci:/localhost/media/usb0` attached to the local machine by invoking the following command:

```
PARCBuffer *emptyBuffer =
parcBuffer_AllocateCString(Secret String');
update('lci:/localhost/media/usb0', emptyBuffer);
```

Since the offset was not provided, a handle to the local device is opened and the buffer elements are appended to the device.

Other information that can be retrieved from the information include file permissions and policies, storage space, etc. Applications on the system are also named, and can therefore also be invoked by the NDI. For example, if the user wanted to run `top` to view a snapshot of process information on the system, the user may invoke the following command:

```
PARCBuffer *procStatusBuffer =
read('lci:/localhost/proc/top');
```

4.1.3 Access Control and Identity Management

Data contained in a content object may be encrypted. Rather than return encrypted data to the application, the NDI can decrypt the content object payload, if given the appropriate cryptographic keys or the ability to retrieve them from an external service or the local operating system. In a similar vein, if content needs to be decrypted or the identity of a particular consumer needs to be resolved to be included in an interest for authorization purposes, the NDI may communicate with the local operating system or an external identity service, e.g., Kerberos, to obtain the necessary cryptographic keys or identity information.

4.1.4 Content Publication

The name of the content specified in the `create()` function determines the location at which the data is actually persisted. For example, the forwarder on a Dropbox host may be configured to route all content with the prefix `lci:/dropbox`

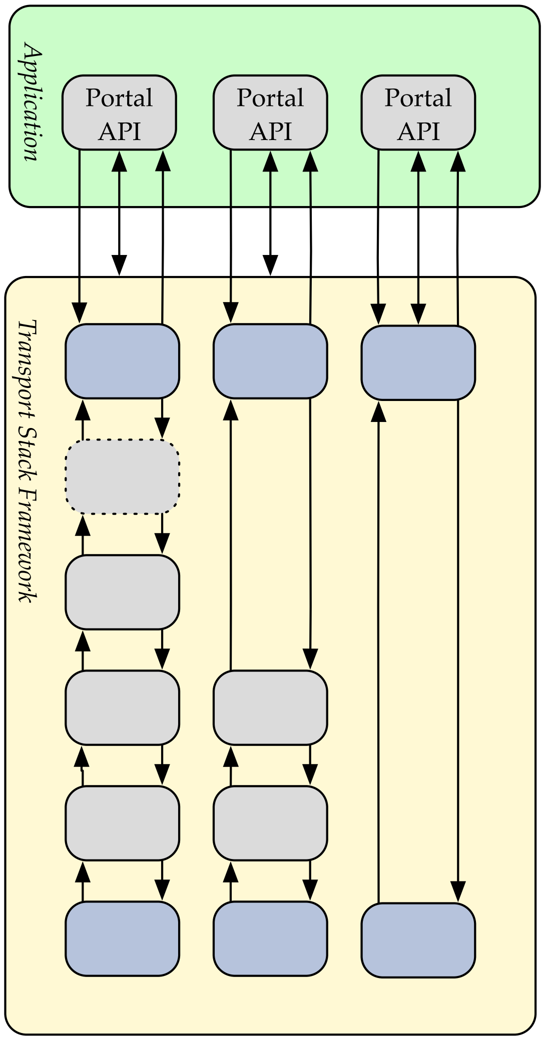

Figure 1: A Transport Framework with multiple, different transport stacks.

to the local machine. Invoking the `create()` function with a name with the same prefix would, naturally, cause the data to be persisted locally. For external hosts, such as remote clients using the Dropbox service, invoking the `create()` function with the same prefix would cause all requests to be forwarded to the appropriate Dropbox host for persistence. Hiding the location behind the name brings applications one step closer to network-agnostic settings.

4.1.5 Remote Authentication and Access Control

When reading data from or writing data to a remote machine, security (authentication and authorization) is a concern. The NDI will implement the name-based access control (NBAC) scheme outlined in [11] for protecting access to remote content. Let C be the client machine initialization a request for a remote read or write operation on content object X, and let P be *any host where the data is located.* If C is authorized to access content X, then all requests (interests) generated via the CRUD API will have signed payloads proving their ability to access the data. P will verify this signature before performing any operation.

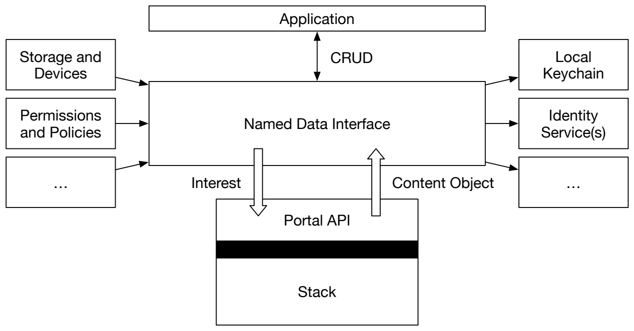

Figure 2: Component-level diagram of the NDI and the interfaces it relies upon, as well as the interfaces it exposes to upper-level applications and APIs. All components above the solid red line are completely network-agnostic.

If NBAC is not employed, then the NDI must authenticate itself or each request so that the target machine can verify the request before reading or writing data. Any CCN-friendly authentication protocol can be used and implemented for this purpose, but we advocate one based on the standard challenge-response paradigm to avoid replay attacks.

4.1.6 State Management

The primary intent of the NDI is to provide a lightweight, simple, and familiar API for creating, reading, and manipulating possibly remote content. The CRUD interface provides users with a way to extract or write data of arbitrary size to content objects. Since the NDI is built on top of the Portal API, which only provides a discrete message interface, the NDI needs to maintain state about all content it manipulates. This state is stored in a content state block (CSB). In the UNIX File API, this is analogous to maintaining a file pointer offset so that calls to `fseek` and `fread` behave as expected. A CSB is maintained for all "open" content handles, i.e., a new CSB is allocated for each request that corresponds to a previously unseen content name. To keep the memory footprint small for each CSB, the only additional state that is maintained beyond the content offset is the access mode (e.g., read, write) and a persistent authentication token, if per-request authenticators are not used. All other information, such as the identity associated with the NDI, can be obtained from the application, local environment, or underlying Portal API instance.

5. NETWORK-AGNOSTIC CONTENT

Network programmers, web application developers, and distributed system designers, to give a few examples, have long been burdened with the task of being acutely aware of the details about network communication. With the evolution of the transport stack to include the NDI, which provides a *similar interface* to that of the UNIX file system, applications no longer have to be concerned with the location of their data (Figure 5 shows the evolution of the network stack towards this "network-agnostic" level of abstraction). Instead, they can focus on the contents and meaning of their data. The fact that the local file system or an external web application provides the data is, and should be, independent of the manner in which said data is consumed. Furthermore,

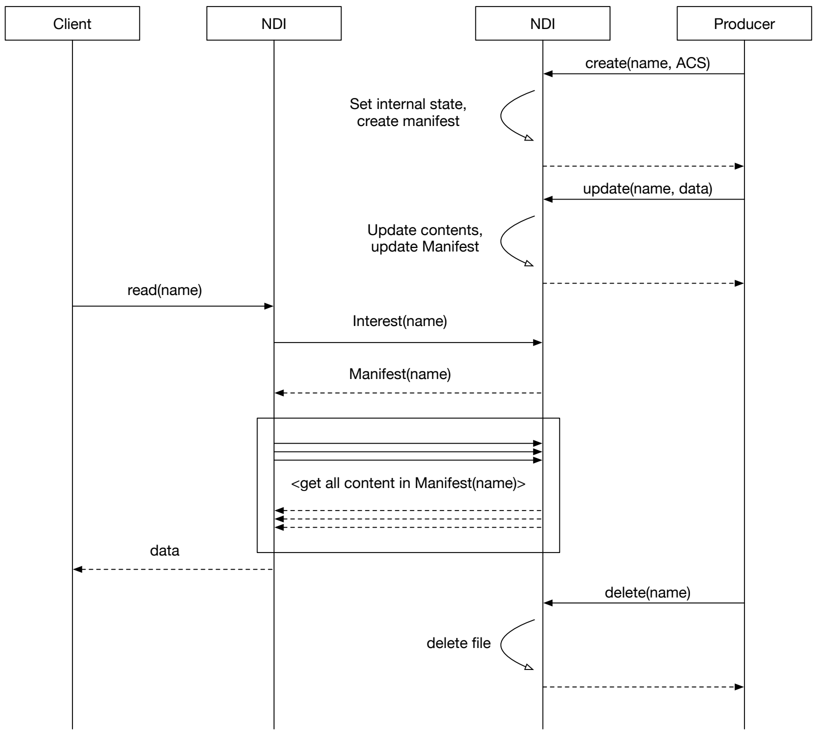

Figure 3: A normal producer-consumer interaction in which the producer performs local CRD operations.

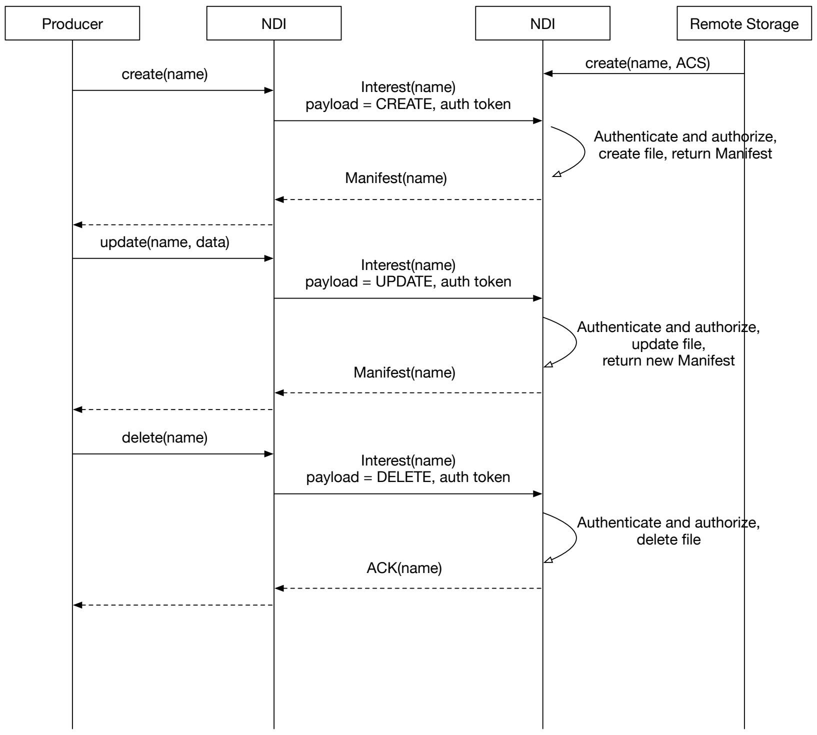
Producer
NDI
NDI
Remote Storage
create(name, ACS)
create(name)
Interest(name) payload = CREATE, auth token
Authenticate and authorize, create file, return Manifest
Manifest(name)
update(name, data)
Interest(name) payload = UPDATE, auth token
Authenticate and authorize, update file, return new Manifest
Manifest(name)
delete(name)
Interest(name) payload = DELETE, auth token
Authenticate and authorize, delete file
ACK(name)

Figure 4: A producer interacting with a remote file store via the NDI CRUD interface.

the API used to interact with this data should be location-agnostic.

The NDI CRUD API enables clients to interact and treat network data as if it were *persistent* – a paradox in today's world. It enables an application to switch – on the fly – from local file based access to network based access without through the same interface. As previously mentioned, persistent data access has traditionally been driven by the UNIX File API, shown in Figure **??**. A vast spectrum of applications from databases to web servers use this API to store, modify, and retrieve data located on persistant storage mediums. Since the NDI provides a layer of abstraction for location-agnostic persistent data, it may also be used by the very same UNIX File API. To leverage the NDI API using the standard UNIX file API, we need a shim between the these layers - an adapter [10]. The adapter, shown in Figure 6, would map the UNIX calls to the assembly CRUD operations as shown in Table **??**. Note that, in all cases, the name of a file can be obtained by the NDI; it just maintains a map from fids to string names.

This adapter would be responsible for "routing" all requests, such as an `open` invocation, to either the local filesystem or Portal API depending on the file name. This has multiple benefits. We elaborate upon a few of these below.

- Notions of storage locations and servers are immediately abstracted away behind this common API.

- Existing applications can migrate to CCN with minimal development effort. Software that currently relies upon the UNIX File I/O API can use the same function signatures for accessing both local and network files without ever having to deal with sockets or IP addresses.

- Richer APIs can be built upon the concept of random access files, including: key-value store, messaging, and data streaming APIs. These would enable greatly simplified development of applications such as databases, email systems, and media streaming services. Simplified software is subject to significantly less bugs, is easier to analyze for correctness concerns and security vulnerabilities, and eases future maintainability.

- Security concerns such as encryption and access control are handled beneath the API. If the requesting client does not have access, whether that means the proper permissions on the local operating system or appropriate set of credentials for some remote service, the file can neither be opened nor used. The ease by which file access control is enforced in modern operating systems is naturally extended to network data.

6. RELATED WORK

Moiseenko and Zhang proposed the first robust API for producer and consumer interation for NDN [22]. It is identical in nature to the Portal API with the caveat that functions are different based on whether an entity is a producer or consumer. An entity is given a set of functions to write data to the network; consumers write interests and retrieve content, and producers write (publish) content. Both interfaces could be implemented using the NDI. Gallo et al. [9] also proposed an abstraction layer, and corresponding ICN network stack, tailored to separate consumer and producer roles for process-to-content communication via the NDN interest-content messaging paradigm. However, their implementation is based on sockets. The CCN Portal was intended to replace the legacy socket interface due to XXX [26]. If this socket-like behavior is truly desired, it can easily be implemented using the CRUD operations exposed by the NDI.

Outside of ICN designs, Zhu et al. [28] proposed a content-centric transport protocol to enable content-oriented messaging on top of existing TCP/IP architectures. Their API is fundamentally identical to the CRUD interface exposed by the NDI; consumer requests are location-agnostic and directed to the appropriate handler (i.e., the local file system or a remote machine) via a proxy request dispatcher, and producer requests are written to local storage. The key difference is that the NDI determines the write location based on the content name, rather than the origin of the publication (i.e., create) request. In a similar vein, APIs [2, 23] and optimizations [24] for I/O forwarding in highly distributed and compute-invensitve systems are similar in nature to the goal of the NDI. The common goal is a set of APIs which enable clients to execute I/O tasks, e.g., `open`, `sync`, `resize` on data without first knowing where the data is stored. That is, clients are not required to fetch data to perform an I/O task – the API serves the dispatch I/O requests to the appropriate node. A similar problem was also studied by Foster et al. in [8], where the goal was to provide an API for remote file manipulation that exploited characteristics of the network. Struggles implementing this sophisticated behavior on the current TCP/IP network stack exemplify the need for more modern network stacks [15]. Accordingly, this location-agnostic property is similar to what the Unix File API shim provides on top of the NDI. As we have argued, this functionality can be implemented with relative ease with the CCN network stack. This ease of implementation is due in part to the hierarchy of APIs upon which the UNIX FIle API shim is built; it embodies clear functional decomposition, which is a driving design goal for modern APIs [6].

7. CONCLUSION

In this paper we presented the CCN Portal and Named Data Interface. As we saw, in addition to the CRUD API exposed by the NDI, the services provided by internally, such as access control (e.g., autonomous content decryption) and object reconstruction, provide a rich level of abstraction for upper-level APIs who need not be concerned by such details. We then showed how this hierarchy of APIs can be used to create a network- and location-agnostic API for I/O, thereby merging local and remote data under a single, standard API. This is critically important as applications become increasingly content-centric and computer systems become increasingly connected; patchwork solutions such as CDNs, which entail complicated overhead and management, will cease to scale as the need for content grows faster than the performance of the underlying network.

8. REFERENCES

[1] CCNx 1.0 Protocol Specications Roadmap. `http://www.ietf.org/mail-archive/web/icnrg/current/pdfZyEQRE5tFS.pdf`.

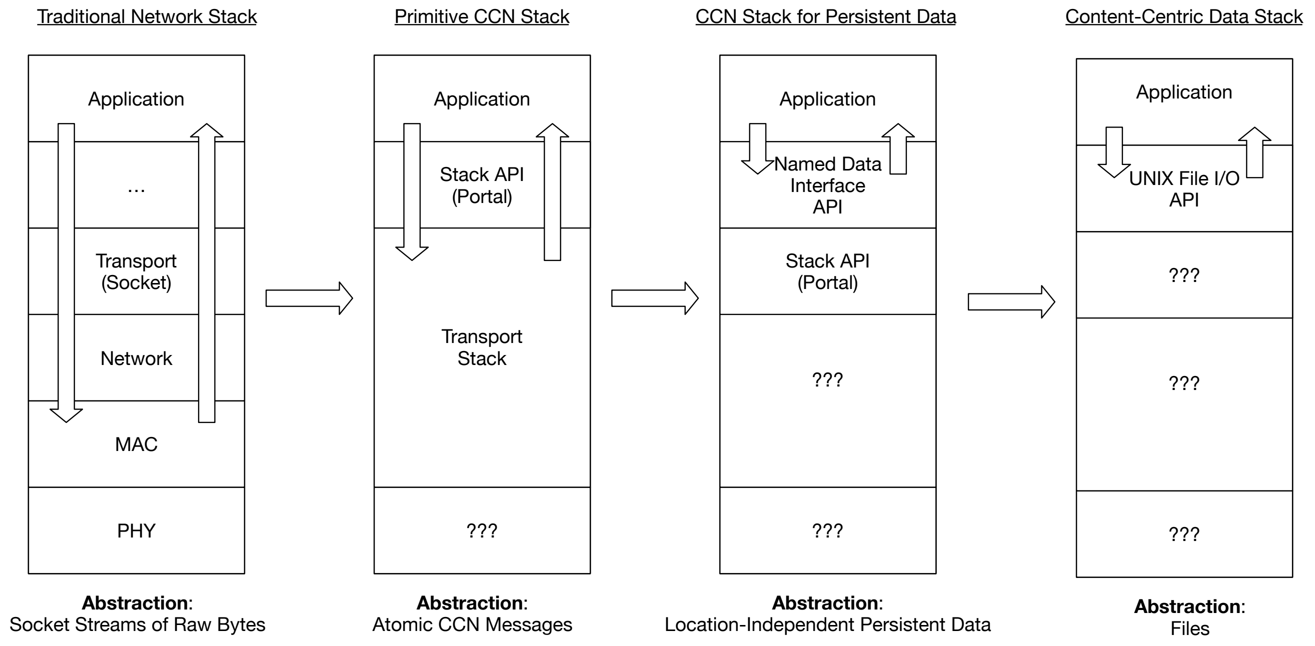

Figure 5: Evolutionary stack progression.

Table 2: The UNIX File API and a mapping to the NDI API.

Function	**Description**	**Mapping**
`open(name, flags, mode)`	Open the specified file for access according the the `mode` paramter.	Setup internal NDI state and return `fid` if mode is READ or WRITE, else `create(name, manifest(mode))` and return `fid`.
`close(fid)`	Close the file associated with the given `fid`.	Destroy internal NDI state.
`read(fid, buffer, numbytes)`	Read the specified number of bytes into the buffer from the file associated with the given `fid`.	buffer = `read(name, numbytes, offset)`
`write(fid, buffer, numbytes)`	Write `numbytes` from `buffer` into the file associated with `fid` and move the file pointer ahead by `numbytes` bytes.	Invoke `update(name, buffer, numbytes, off` and if the file offset is nonzero, then start update from that position.
`lseek(fid, offset, base)`	Move the file pointer associated with the `fid` file by `offset` bytes from the `base` address.	Update internal file pointer.
`creat(name, mode)`	This is synonomous to `open(name, CREATE | TRUNC | WRONLY, mode)`.	Invoke `create(name, manifest(mode))` return `fid`

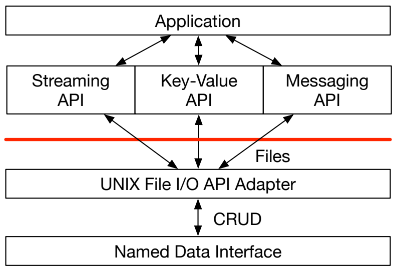

Figure 6: The shim between the UNIX File and NDI API, merging remote and local data under a single, uniform interface.

[2] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and P Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[3] Gustavo Carneiro, José Ruela, and Manuel Ricardo. Cross-layer design in 4 g wireless terminals. *IEEE Wireless Communications*, 11(2):7–13, 2004.

[4] CCNx Node Model. `http://www.ccnx.org/releases/latest/doc/technical/CCNxProtocol.html`.

[5] Vinton G Cerf and Robert E Icahn. A protocol for packet network intercommunication. *ACM SIGCOMM Computer Communication Review*, 35(2):71–82, 2005.

[6] David D Clark and David L Tennenhouse. Architectural considerations for a new generation of protocols. *ACM SIGCOMM Computer Communication Review*, 20(4):200–208, 1990.

[7] Behrouz A Forouzan. *TCP/IP protocol suite*. McGraw-Hill, Inc., 2002.

[8] Ian Foster, David Kohr Jr, Rakesh Krishnaiyer, and Jace Mogill. Remote i/o: Fast access to distant storage. In *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 14–25. ACM, 1997.

[9] Massimo Gallo, Lin Gu, Diego Perino, and Matteo Varvello. Nanet: socket api and protocol stack for process-to-content network communication. In *Proceedings of the 1st international conference on Information-centric networking*, pages 185–186. ACM, 2014.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[11] Cesar Ghali, Marc A Schlosberg, Gene Tsudik, and Christopher A Wood. Interest-based access control for content centric networks. In *Proceedings of the 2nd International Conference on Information-Centric Networking*, pages 147–156. ACM, 2015.

[12] Cesar Ghali, Gene Tsudik, and Ersin Uzun. Network-layer trust in named-data networking. *ACM SIGCOMM Computer Communication Review*, 44(5):12–19, 2014.

[13] M. Gritter and D. Cheriton. An architecture for content routing support in the internet. In *USENIX USITS*, 2001.

[14] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[15] Florin Isaila, Javier Garcia, Jesús Carretero, Rob Ross, and Dries Kimpe. Making the case for reforming the i/o software stack of extreme-scale systems. *Preprint ANL/MCS-P5103-0314, Argonne National Laboratory*, 2014.

[16] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, pages 1–12. ACM, 2009.

[17] Leonard Kleinrock. Information flow in large communication nets. *RLE Quarterly Progress Report*, 1, 1961.

[18] T. Koponen et al. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM*, 2007.

[19] Jun Kurihara, Christopher Wood, and Ersin Uzuin. An encryption-based access control framework for content-centric networking. *IFIP Networking 2015*, 2015.

[20] James F Kurose. *Computer networking: a top-down approach featuring the Internet*. Pearson Education India, 2005.

[21] Robert M Metcalfe and David R Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[22] Ilya Moiseenko and Lixia Zhang. Consumer-producer api for named data networking. In *Proceedings of the 1st international conference on Information-centric networking*, pages 177–178. ACM, 2014.

[23] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Delegation-based i/o mechanism for high performance computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2):271–279, 2012.

[24] Kazuki Ohta, Dries Kimpe, Jason Cope, Kamil Iskra, Robert Ross, and Yutaka Ishikawa. Optimization techniques at the i/o forwarding layer. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 312–321. IEEE, 2010.

[25] Lawrence G Roberts. Multiple computer networks and intercomputer communication. In *Proceedings of the first ACM symposium on Operating System Principles*, pages 3–1. ACM, 1967.

[26] Joel M Winett. Definition of a socket. 1971.

[27] Robert H Zakon. Hobbes' internet timeline. 1997.

[28] Yuncheng Zhu and Akihiro Nakao. Content-oriented transport protocol. In *Proceedings of the 7th Asian Internet Engineering Conference*, pages 104–111. ACM, 2011.

[29] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.