On Pending Interest Table in Named Data Networking

Huichen Dai[†] Bin Liu[†] Yan Chen[§] Yi Wang[†]
[†] Tsinghua National Laboratory for Information Science and Technology
[†] Dept. of Computer Science and Technology, Tsinghua University, China
[§] Northwestern University, USA
dhc10@mails.tsinghua.edu.cn, liub@tsinghua.edu.cn, ychen@cs.northwestern.edu, wy@ieee.org

ABSTRACT

Internet has witnessed its paramount function transition from host-to-host communication to content dissemination. Named Data Networking (NDN) and Content-Centric Networking (CCN) emerge as a clean slate network architecture to embrace this shift. Pending Interest Table (PIT) in NDN/CCN keeps track of the Interest packets that are received but yet un-responded, which brings NDN/CCN significant features, such as communicating without the knowledge of source or destination, loop and packet loss detection, multipath routing, better security, etc.

This paper presents a thorough study of PIT for the first time. Using an approximate, application-driven translation of current IP-generated trace to NDN trace, we firstly quantify the size and access frequencies of PIT. Evaluation results on a 20 Gbps gateway trace show that the corresponding PIT contains 1.5 M entries, and the lookup, insert and delete frequencies are 1.4 M/s, 0.9 M/s and 0.9 M/s, respectively. Faced with this challenging issue and to make PIT more scalable, we further propose a Name Component Encoding (NCE) solution to shrink PIT size and accelerate PIT access operations. By NCE, the memory consumption can be reduced by up to 87.44%, and the access performance significantly advanced, satisfying the access speed required by PIT. Moreover, PIT exhibits good scalability with NCE. At last, we propose to place PIT on (egress channel of) the outgoing line-cards of routers, which meets the NDN design and eliminates the cumbersome synchronization problem among multiple PITs on the line-cards.

Categories and Subject Descriptors

C.2.1 [**COMPUTER-COMMUNICATION NETWORKS**]: Network Architecture and Design

Keywords

PIT, Size, Frequency, Encoding

1. INTRODUCTION

The functionality of Internet has evolved substantially, though it was originally designed for host-to-host communication, it now mostly serves content-centric applications. Therefore, researchers arrive at a widely recognized agreement that content should have a more central role in future network architectures than it does in the current Internet's host-centric conversation model [5, 7]. The research community addresses this problem with a paradigmatic shift – Content-Centric Networking [10] (CCN), which focuses on content dissemination based on content identifiers rather than content hosts. Named Data Networking [15] (NDN) is an instance of the general CCN paradigm.

NDN communication is requester-driven. A requester sends out an *Interest* packet, which carries a name – the identifier – that specifies the desired data. PIT in an intermediate router remembers the Interest name and from which interface the Interest comes in. Content providers respond to the Interest by sending back a *Data* packet that carries both the name and desired content. Once Data packet arrives at an intermediate router, the router looks up its name in the PIT to obtain the interface from which the requesting Interest comes, and deletes that name from PIT. Therefore, a router inserts every incoming Interest into PIT, and removes each received Data packet from PIT. Intuitively, due to the high-speed packet arrival rate, PIT will have a large size and demand extremely high access (lookup, insert and delete) frequency, which has caused wide debate on the feasibility of PIT. The features that PIT brings (Section 2.3) makes it indispensable to NDN/CCN. Therefore, the PIT issue becomes a knotty obstacle that hinders practical and scalable implementation of NDN.

Though current researches on NDN/CCN are in full swing, the study on PIT is quite exiguous. To the best of our knowledge, we are the first to conduct measurements on exact PIT size and access frequency, and further propose solutions to address this untouched issue, i.e., shrinking PIT size and improving PIT access performance. To unfold this problem, we are faced with the following challenges:

1. How to evaluate and quantify the PIT size and frequency while NDN/CCN has NOT been really deployed?
2. Different from merely lookup a name, how to well address the insert and delete operations?
3. How to support PIT scalable to large name sets while still sustaining high PIT access performance?

In this paper, we propose a real trace-translation/mapping method to measure the size and access frequency required by PIT. We captured a one-hour trace from a 20 Gbps gateway link in the China Education and Research Network (CERNET) for our experiment. Afterwards, a Name Component Encoding (NCE) solution is put forward to reduce PIT's memory consumption and promote the access performance. At last, we also present a scheme on where to place PIT in NDN routers when actually implementing PIT. Especially, we make the following contributions:

1. We emulate NDN applications' working paradigms by transferring the existing IP applications to the NDN platform. Then, by translating/mapping our captured IP trace to NDN scenario at the perspective of applications, we quantify the size and access (lookup, insert and delete) frequencies required by PIT. Experimental results on a trace collected from

ANCS'12, October 29–30, 2012, Austin, Texas, USA.
Copyright 2012 ACM 978-1-4503-1685-9/12/10 ...$15.00.

a 20 Gbps gateway link show that the corresponding PIT has 1.5 M entries, and its lookup, insert and delete frequencies are 1.4 M/s, 0.9 M/s and 0.9 M/s, respectively. These results imply that directly storing and accessing PIT entries as character strings in commodity memories is not scalable and incurs great challenges, especially for insert and delete operations.

2. We continue to adopt an encoding-based idea (NCE) and make important improvements to shrink the PIT size and satisfy the access frequency requirement. The encoding idea was first proposed in our previous work [14]. Experimental results demonstrate that by our solution, the PIT size can be be reduced by up to 87.44%, and the lookup, insert and delete frequencies can achieve 3.27 M/s, 2.93 M/s and 2.69 M/s respectively on an Intel 2.27 GHz CPU, satisfying the access frequency requirement of the studied PIT.
3. Moreover, in combination with the router architecture, we design an ingenious PIT residence scheme that places PIT on packets' outgoing line-cards (egress channel), avoiding the cumbersome synchronization problem among multiple PITs on line-cards.

The remainder of the paper is organized as follows. Section 2 provides NDN background information and our motivation. Section 3 designs working paradigms on the NDN platform for existing IP network applications, and measures the size and access frequency of PIT. We propose an encoding-based solution to reduce the PIT size and promote PIT access performance in Section 4. Section 5 proposes a PIT residence solution on router outgoing line-cards. We evaluate our solutions in Section 6, Section 7 is the comparison of previous work, and Section 8 concludes this paper.

2. BACKGROUND AND MOTIVATION

2.1 NDN Introduction

NDN, a specific instance of the CCN paradigm, is a novel network architecture proposed by [15] recently. Different from current Internet practice, it makes content ("what") as its central role, rather than "where" content is located. A critical distinction from IP is that, every piece of content in NDN network has an assigned name, and packets are routed/forwarded by names, rather than IP addresses.

NDN names are application-dependent and opaque to the network, but they all share common characteristics – hierarchically structured and composed of explicitly delimited *components*. The delimiters, usually slash ('/') or dot ('.'), are not part of the name. The naming system is an important piece in the NDN architecture and is now still under active research. For the purpose of early exploring the properties of PIT before the naming specification finally goes to standard, in this paper, we temporarily use *hierarchically* reversed domain names as NDN names. For example, the scholar service provided by Google – *scholar.google.com* is hierarchically reversed to *com/google/scholar*, and *com*, *google*, *scholar* are three components of the name. For an HTTP URL, we hierarchically reverse its host name, and concatenate the rest part, such as absolute path, as an NDN name. For instance, URL *name.example.com/path/to/content* is transferred to an NDN name of *com/example/name/path/to/content*. The hierarchical structure, which resembles IP addresses, enables name aggregation and allows fast name lookup of Longest Prefix Match (LPM), and will be leveraged by PIT lookup.

NDN adopts a brand new data requester-driven communication mechanism. To receive data, a requester sends out an *Interest* packet, which carries a name – the identifier – that specifies the desired data. An intermediate router remembers the name and the interface from which the Interest comes in the PIT, and then forwards the Interest by looking up its name in the Forwarding Information Base (FIB). When the Interest arrives at a node that serves the requested data, a *Data* packet that carries both the name and the content is returned. Once the Data packet reaches a router, the router looks up its name in the PIT to obtain the interface from which its corresponding Interest comes in, and then forwards Data to that interface. Therefore, Data travels back to the requester by taking the same path of the Interest, but in the reverse direction, i.e., symmetric routing. Moreover, the Data packet is strategically cached by a router's Content Store (CS) to serve subsequent Interests.

2.2 Packet Lookup and Forwarding in NDN

The NDN packet lookup and forwarding processes are a bit complicated than that of IP. To better comprehend the packet lookup and forwarding process, keep in mind that PIT keeps track of *pending* Interests, i.e., received but yet un-responded Interests. The specific lookup and forwarding processes of Interest and Data packet are shown in Figure 1(a) and Figure 1(b), respectively. Figure 1(a) shows that once an Interest packet arrives at interface i of an NDN router R, R:

1. consults CS if the desired content is present and returns a copy in Data packet via i,
2. if not, looks up PIT to see if PIT has an entry for this Interest. If so, adds i to that entry, and discards this Interest packet,
3. otherwise, creates a PIT entry for this Interest and add i to this entry, and
4. forwards Interest to the next-hop interface by looking up FIB.

When Data packet returns, Figure 1(b) shows that R:

1. forwards the Data packet over all the requesting interfaces in the corresponding PIT entry and deletes this entry,
2. caches the Data packet in the CS based on policies.

2.3 Motivation

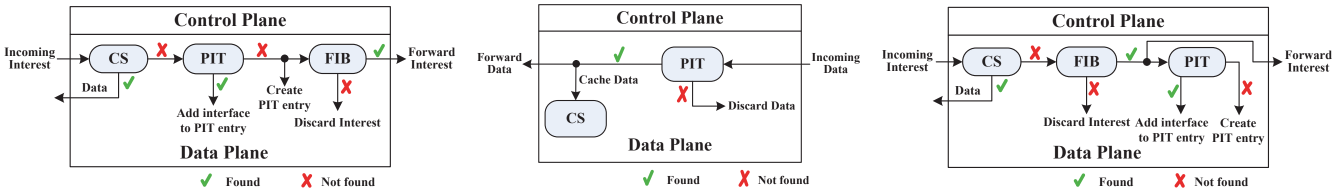
We have presented the role that PIT plays in NDN, and PIT brings NDN significant features:

1. PIT enables Interest and Data packets be routed without specifying a source or destination address, which means PIT makes NDN communication concentrates on the content itself, rather than where the content locates or who are exchanging data.
2. The feature above inherently supports anonymous communication, making attacks difficult to launch and communication more secure.
3. PIT prevents Interests loop persistently, because Interest name plus a random nonce, which is stored in the PIT entry, can effectively identify duplicates to discard. Data packets do not loop since they take the reversed paths of Interests.
4. The property above also enables NDN to inherently support multipath routing, because NDN router can send out an Interest via multiple interfaces without worrying about loops.
5. PIT supports Data packet multicast when multiple Interests received by the router apply for the same content.
6. At last, PIT can detect Data packet losses if an Interest has not been responded beyond a time threshold.

All these significant features that PIT brings to NDN enable NDN/CCN to be an information/content-centric network, and therefore motivate us to conduct a thorough study on PIT.

3. PIT SIZE AND ACCESS FREQUENCY

There has been a debate on PIT since the proposal of PIT, because each Interest looks up, inserts and updates PIT, and each Data packet looks up and removes PIT entries, intuitively resulting in extremely large size and high access frequency of PIT. Interests are triggered



(a) Interest lookup and forwarding process.

(b) Data lookup and forwarding process.

(c) Interest lookup and forwarding process when placing PIT on the outgoing line-cards.

Figure 1: Packet lookup and forwarding process.

by applications for data communication. Each application has its own way to send out Interest and Data packets, and the mechanism of sending out packets influences the size and access frequency of PIT. In other words, each application has its own way to construct and destruct PIT entries. Therefore, we measure the size and access frequency of PIT at the perspective of application, rather than at the network layer.

3.1 Applications from IP to NDN platform

Researchers propose that Internet architecture switch from IP to NDN to better satisfy the requirements of users, such as browsing web pages, sharing files, and watching videos, etc. Therefore, the network applications demanded by users will not change, but require different implementations on the NDN platform. Consequently, we design working paradigms for the major applications in current Internet – HTTP, FTP, P2P, Email, Online Game, Streaming media, Instant messaging – based on NDN communication model[1], i.e., transfer applications from IP platform to NDN platform.

Fundamentally, NDN's requester-driven communication model *pulls* data from remote host to local host – a one-way data flow service. NDN has to solve the problem of bootstrapping two-way communications on top of a fundamentally one-way service.

3.1.1 HTTP

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request Method, request-URI (Uniform Resource Identifier), etc., over a connection to a server. The server responds with a message, which includes a success or error code, followed by entity meta information and possible entity-body content.

HTTP communication is initiated by a client sending out a request to be applied to a resource on some origin server. In the simplest case, this may be accomplished via a single connection between the client and the origin server. Therefore, HTTP accords with the NDN requester-driven communication model. The HTTP requests can be divide into different categories because of the different methods they take with them. The *Method* token of an HTTP request indicates the method to be performed on the resource identified by the Request-URI. Method includes:

1. OPTIONS: allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server.
2. GET: retrieves whatever information identified by the Request-URI.
3. HEAD: identical to GET except that the server MUST NOT return a message-body in the response, but only the metainformation about the resource.
4. POST: requests that the origin server accept the data enclosed in the request as a new subordinate of the resource identified by the Request-URI.
5. PUT: requests that the enclosed data in the request be stored under the supplied Request-URI.
6. DELETE: requests that the origin server delete the resource identified by the Request-URI.
7. TRACE: invokes a remote, application-layer loop-back of the request message.
8. CONNECT: (reserved).

NDN only contains two kinds of packets: Interest and Data. Interest packet requests for a specific piece of content, i.e., *pulls* desired content, which is similar to the HTTP OPTIONS, GET and HEAD method. Thus, these three methods can be directly implemented by Interest packet, with the Request-URI being Interest name, and the method encapsulated in the Interest body. Their corresponding HTTP responses are implemented by Data packets. However, in order to implement the integrated HTTP protocol in NDN, we should extend the functionality of Interest packet to accommodate the rest methods. Let's examine them one by one.

For request with POST or PUT method, it tries to *push* data to the server, which violates NDN's pull communication model. To implement the pushing HTTP request, Interest packet should be *extended* to not only contain the name of the requested content, but also include a data block that contains the content to be pushed to the server. By this means, requests of POST and PUT method can be implemented by Interests as well, with their request-URIs also being Interest names. Their responding Data packets notifies the clients of the status of these requests, such as SUCCESSFUL, NOT ALLOWED, NOT IMPLEMENTED, etc. The rest two methods, DELETE and TRACE, neither pulls or pushes data, but invokes a function on the server. Now that we have extended the Interest from pull to push communication model, it is no harm that we further extend Interest to function calls (delete, echo-back, etc.). In this way, DELETE and TRACE can be implemented by the Interest packet as well. For DELETE, the Interest specifies the name of the to be deleted resource by Request-URI, and Data packet returns the status of the deletion. For TRACE method, the Interest is echoed-back by the server in Data packet.

It's worth pointing out that, though NDN is a brand new network-layer architecture, it is still built on current technologies. The data link layer evolves separately from network layer, and the current dominant layer-two standard is Ethernet. We believe that NDN still employs the data transmitting services that current data link layer technologies provide. Therefore, due to the Maximum Transmission Unit (MTU) of the data link layer, the HTTP response message, when being transmitted back to the client, may be segmented into multiple IP packets. Similarly, in NDN, a HTTP response message is likely to be segmented into multiple Data packets as well, rather than a single Data packet.

Subsequently, we examine the life time of PIT entries created by NDN HTTP connections. An NDN HTTP connection v is initialized by a client sending out an Interest packet. Assume that this Interest packet enters router R via line-card i and leaves via line-card j, line-card j (not line-card i, see Section 5) creates an PIT entry when receives it, and keeps this entry until all the responding Data

[1] NDN does not have a separate transport layer, thus all the NDN applications build directly on the request-driven communication model.

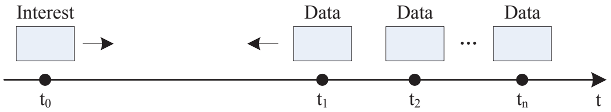


Figure 2: The life time of this PIT entry is $t_n - t_0$.

packets of this NDN HTTP response message have been returned, i.e., as the last Data packet arrives at line-card j, it removes the corresponding PIT entry. Therefore, one NDN HTTP request/response pair corresponds to one PIT entry, and the life time of this PIT entry is the time interval between the Interest packet and last Data packet observed by a router, as shown in Figure 2.

3.1.2 FTP

Similar to HTTP, we implement FTP on the NDN platform by assigning more roles to Interest and Data packet to accommodate the FTP requests and responses.

FTP is built on a client-server architecture and uses separate control and data connections between the client and the server, as depicted in Figure 3. The client's Protocol Interpreter (PI) initiates the control connection, then standard FTP commands are generated by the client PI and transmitted to the server process via the control connection. Standard replies with status codes are sent from the server PI to the user PI over the control connection in response to the commands. The FTP commands specify the parameters for the data connection (data port, transfer mode, representation type, etc.). Data connection can either be opened by the server (active mode) from its default port to a negotiated client port, or by the client (passive mode) from an arbitrary port to a negotiated server port as required to transfer file data.

The control connection is in an interactive mode, which can make direct use of NDN's request-driven communication model, with Interest packets implementing commands and Data packets acting as replies or acknowledgements. A command is encapsulated in the Interest packet, and now the Interest's name is not the name of desired data, but the FTP service name on the FTP server. Data packets are replies or acknowledgements of the commands and contain the status code of the command executing result.

IP FTP data connection may run in active or passive mode, which determines how the data connection is established. The difference between them is whether the client or the server opens the data connection. In active mode, the server initializes the data connection. In situations where the client is behind a firewall and unable to accept incoming connections, passive mode is used and the client initiates the data connection to the server. However, in whatever mode the data connection is initiated, a control connection must be initiated by the client first!

No matter established by active mode or passive mode, the data connection of IP FTP is duplex and bidirectional, i.e., the client and server can actively send data to each other over this very data connection. However, the NDN connection is unidirectional, the data can only flow from the host possessing the data to the host sending out Interests. Consequently, in NDN, the way how the data connection established alters and depends on whether the client C wants to upload file to or download file from the server S. If client C wants to download file from server S, it has to actively send out an Interest to initiate the data connection, which corresponds to the passive mode (initiated by client) of IP FTP. To implement this, client C first negotiates with server S over control connection by sending out an PASV command, then server S replies its information such as FTP service name. Next, client C uses this information to establish a data connection by sending out an Interest with S's FTP service name to S, now server S can send file to client C by sending back Data packets. This process is shown in Figure 4(a).

However, if client C wants to upload file, which corresponds to the active mode (initialized by server) of IP FTP, client C first sends its information (FTP service name) to server S, and server S acknowledges with a Data packet. Then server S sends client C an Interest to establish a data connection. After receiving the Interest, client C can upload file to server S, which is illustrated in Figure 4(b). Till now, we have designed the working paradigm on NDN platform for both FTP control connection and data connection.

It's known that, each Interest creates a PIT entry and the PIT entry lasts until the responding Data packet returns. Because the control connection is interactive, a corresponding PIT entry is inserted and removed over and over again. At each snapshot, there can be at most one PIT entry that corresponds to the control connection, but control connection brings high PIT access frequency. For data connection, because the size of a file can be relatively large and will be segmented, the PIT entry will last until all the Data packets are returned, which is very like that of an HTTP connection.

3.1.3 P2P

P2P is no doubt a major application in current Internet and contributes vast traffic. To implement P2P on the NDN platform, we should know how it works. Fortunately, it also obeys the request-driven paradigm.

P2P network has two constructing ways: unstructured and structured. In unstructured P2P networks, a node sends out query packets by flooding or smarter algorithms [11, 12] to search for nodes serving desired files, and those nodes possess the requested files will send content back to the requester. In structured P2P networks, a node first consults its local Distributed Hash Table (DHT) about which nodes have the desired content, and then directly send requests to those nodes. No matter unstructured or structured P2P network, P2P's working paradigm is the closest to that of NDN. The query packets or the requests can be directly implemented by Interest packets "as-is", as well as the replies implemented by Data packets without any modification.

The corresponding PIT entries are created and removed in the way similar to that of HTTP.

3.1.4 Email

Implementing Email service in NDN will encounter some difficulties and calls for an a new mechanism. Because hosts *push* emails out to servers, which is fundamentally different from NDN's communication *pull* model.

Email servers and other email transfer agents use Simple Mail Transfer Protocol (SMTP) to send and receive email messages, user-level client email applications typically only use SMTP for sending messages to a email server for relaying. For receiving messages, client applications usually use either the Post Office Protocol (POP) or the Internet Message Access Protocol (IMAP).

Firstly, we consider sending emails. In IP, email is submitted by a mail client (MUA, mail user agent) to a mail server (MSA, mail submission agent) using SMTP. The messages can be directly sent to the server without any knowledge of the server in advance (except the email address). In however, in NDN, we need a "initiating" stage. Assume that a Gmail client (logged in with email address example@gmail.com) tries to send an email to "van@parc.com". It initiates the transaction by sending out an NDN Interest with the name: "parc.com/email_service/van/example@gmail.com/123". ("parc.com/email_service/" is a routable prefix for the email service of PARC, and 123 is a nonce.) After receiving the Interest, the PARC server responds to the client's Interest with an "OK" Data packet as acknowledgement. This process is called the "initiating" stage. Then the PARC server then sends out an Interest with the name: "gmail.com/email_service/example/van@parc.com/456" ("gmail.com/email_service" is the routable prefix to the Gmail server, and 456 is also a nonce) to pull down the incoming email. By this means, the Gmail mail client can send out an email via

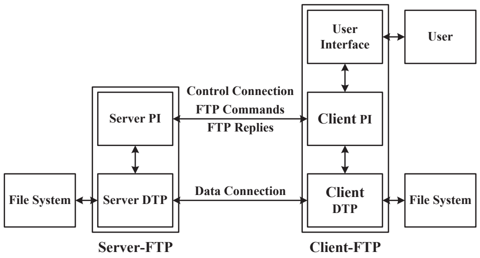
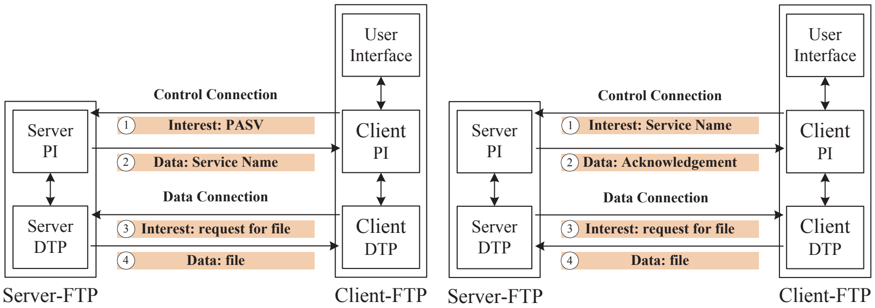


Figure 3: Model for FTP.



(a) NDN FTP: download file.

(b) NDN FTP: upload file.

Figure 4: Upload and download file process of NDN FTP.

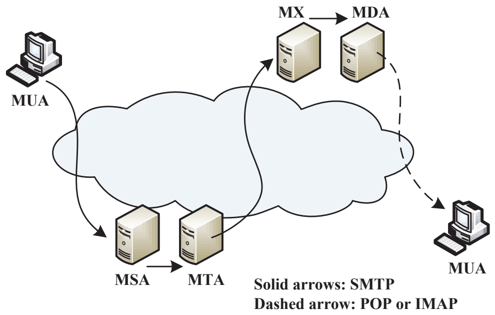


Figure 5: Email.

the responding Data packets. After the email is delivered to the MSA, it is forwarded to the Mail Transfer Agent (MTA) and Mail Delivery Agent (MDA) in the same way (Solid arrows in Figure 5). Therefore, for the NDN email sending process, it takes a little longer than that of IP for about a RTT time due to the initiating step.

Secondly, we consider the mail delivery procedure (Dashed arrows in Figure 5). Once delivered to the local mail server, the mail is stored for retrieval by authenticated email clients (MUAs), using POP or IMAP. When using POP, clients typically connect to the email server briefly, only as long as it takes to download new emails. Therefore, for POP, email clients actively send Interests to email servers when they want to receive emails, and mail server responds with emails by Data packets.

For IMAP, clients often stay connected as long as the user interface is active and download emails on demand. In other words, IMAP pushes emails to mail clients. Therefore, we adopt the same method for the email sending procedure. Let mail server send an Interest to mail client, then mail client acknowledges this Interest by a Data packet, and next sends another Interest to pull down the emails.

3.1.5 Online games

In current Internet online games, there must be a consistent connection between the client and server since an account logs in until it logs out, which is demanded by the application requirements. In NDN, this requirement does not alter, which reveals bidirectional communications between game client and game server. Therefore, the game client sends out an Interest with the user name and password to login to the server. (The Interest's name is set to a routable prefix of the game server.) This Interest is extended to a special kind of Interest that its corresponding PIT entries will not be removed while the user is logged in, in order to support the consistent connection requirement. By these PIT entries, a consistent channel from the server to the client is created. The server can arbitrarily send Data packets to the client. However, this unidirectional channel is not enough since the client also needs to send messages such as mouse clicks and keystrokes to the server. Consequently, after receiving the Interest from a client, the server sends an Interest to the client as well, which is also a special Interest that its PIT entries will not be removed until the client logs out. In this way, a channel from the client to the server is also established, and the two unidirectional channels form a bidirectional connection between the game client and server.

3.1.6 Streaming media

Due to the entertainments brought by on-line videos, streaming media has become one of the major Internet applications, such as the Youtube website. To watch a video, a client (e.g., Adobe flash player) sends an Interest to Youtube website, and then Youtube sends back the video segments to the client in continuous Data packets, like a stream. If P2P accelerating techniques are used, the Interest is sent from a peer to another, and another peer sends back the video encapsulated in Data packets. Refer to *HTTP* and *P2P* for how the corresponding PIT entries are created and removed.

3.1.7 Instant messaging

Instant messaging (IM) has been integrated into people's lives. It also *pushes* out messages like sending emails, which violates the the NDN's communication model. In current Internet, IM mostly uses UDP to transport packets and thus provides best-effort service. Therefore, we design a mechanism for IM in NDN that consumes no PIT entry, which offers best-effort service as well.

In our design, an IM client logs into the IM server by sending an Interest to the server. The Interest's name is the IM service name of the server (a routable prefix to the server), and payload includes an account's user name, password, and local IM service name (a routable prefix to the client). The server acknowledges with a Data packet, which also includes the remote IM service names of the account's friends, indicating which friends are online. If the user wants to initiate a conversation with a friend, he sends out an Interest, with the friend's remote service name as the Interest's name, and the message that he wants to deliver to his friend as the Interest's payload. This kind of Interest is classified as another special kind of Interests that intermediate routers will NOT create PIT entry for it. Moreover, when the Interest reaches the destination client, the client will NOT return a Data packet either, providing best-effort service.

By now, we have designed working paradigms for the most major Internet applications on the NDN platform. In next subsection we will evaluate and quantify the size and access frequency of PIT.

3.2 Measuring PIT size and access frequency

We captured a one-hour trace from a 20 Gbps link in China Education and Research Network (CERNET). By mapping or translating this IP-generated trace to an NDN trace at the perspective of application, we quantify the PIT size and access frequency. Videlicet, to transmit the same data encapsulated in the payload of the IP trace via NDN (with the data link layer technologies remain the same), how many PIT entries and how much access frequency of PIT is required. For example, for a pair of HTTP request and response, a PIT entry will last since the arrival of the Interest packet until the arrival of the last Data packet of the response message. For an SMTP connection, we should add an initiating stage right before it starts, contributing a PIT entry for about a RTT time. We first parsed out the bifows[2] (connections for TCP and bidirectional flows for UDP) of each application discussed above. The parsing process takes advantage of multiple tools, including TIE [3, 6], Tstat [4, 8], l7-filter [2], etc., and we parsed our 88.76% of the total trace. The numbers of biflows of each application at each snapshot[3] are shown in Table 1 (except the second column is the # of yet un-responded HTTP requests).

From Table 1, as well as our designed working paradigms of NDN applications, we translate this trace from the IP scenario to the NDN scenario. E.g., the second column "Un-responded HTTP requests" presents the number of HTTP requests that have not been responded. Because HTTP/1.1 [1] adopts consistent connection and multiple HTTP packets can be sent via the same connection in a pipelined manner, but in NDN, each un-responded Interest packet has an associated entry in PIT, thus the number of PIT entries is related to the un-responded requests at a snapshot, rather than the number of HTTP connections. Another example is Online Game, as aforesaid, two unidirectional NDN channels realize the functionality of an IP connection, therefore, each Game connection corresponds to 2 PIT entries. In this way, by translating all the major applications, we derive that the number of PIT entries is around 1.5 million. We only consider the traffic of network applications, thus the traffic of network management (e.g., ICMP) and services (such as DNS, which is no more needed in NDN) are not counted (the last two columns in Table 1). IM belongs to the Conference group, which is not counted, either.

Moreover, the read frequency of PIT depends on how many Data packets (excluding the last Data of a request/response pair) arrive per second, the insert frequency is determined by the number of emerging biflows (new Interest arrive), and delete frequency the number of disappearing biflows (last Data packets arrive). These statistics of the first 10 seconds (out of 3,600 seconds) of the studied trace are provided in Table 2. It's worth pointing that we ignore all the packets without payload when calculating the PIT access frequency. From Table 2, the read frequency of PIT is around 1.4 M/s, and the insert and delete frequencies are both around 0.9 M/s.

4. ENCODING-BASED FAST PIT ACCESS AND SIZE SHRINKING

In this section, we propose an encoding-based approach to shrink PIT size, and accelerate the lookup, delete and update performance.

As the NDN names are composed of components and are relatively long compared to IPv4/v6 addresses. Directly storing them as character strings in a table and search for a match is not a wise idea. The hierarchical structure of NDN names enable some of them to share the same name prefixes, thus the names can be organized in Name Prefix Trie (NPT), which is very likely to the IP Prefix trie [9, 13] structure. NPT (middle part in Figure 7) makes the PIT organized and more manageable, but it does not help much in shrinking PIT size and accelerating PIT access frequency.

We propose to assign a code (an integer) to each component of the name, i.e., the Name Component Encoding (NCE) method, and use the codes to build the NPT, called the Encode Name Prefix Trie (ENPT). The codes are utilized to conduct lookup, delete and update on ENPT (rightmost part in Figure 7) to make PIT access faster. We should keep in mind the following problems when adopting NCE:

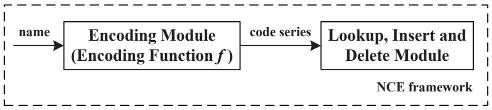


Figure 6: The NCE framework.

1. High-speed Longest Prefix Match (LPM) lookup, insert and delete, which are the major objectives of the NCE mechanism.
2. Fast component encoding. When a packet arrives, the name's components must be assigned codes before starting the longest prefix matching. Therefore the speed of component encoding should be no slower than the lookup, insert and delete speed.
3. Low memory cost. An effective encoding-based method should reduce the total memory cost of PIT as well.

The encoding-based solution was first introduced in our previous work [14]. In this paper, we only adopt the *encoding* idea, the ways to allocate codes, lookup, insert and delete is totally different. Based on our solution, when an NDN name arrives, its components are extracted and each of them is assigned a code. Then the code series are used to lookup, delete and update the PIT, as shown by Figure 6.

4.1 Comparing With Pattern Matching

Intuitively, our problem is to look for a string among a set of strings, which is very similar to the pattern matching problem at the first glance. However, there are some fatal differences.

1. Pattern matching checks if a string contains a pattern (or more than one patterns), beginning at arbitrary position of the string. However, we lookup the NDN names according to the LPM rule, which begins at the first character of the input string. Moreover, pattern matching resolves at character granularity, while our problem is of component granularity.
2. The patterns in conventional pattern matching problems are fixed or update at a very low frequency. While in our problem, the names in the PIT are all patterns, and the patterns are inserted and removed at a very high frequency, which will lead to frequent reconstruction of the Deterministic Finite Automaton (DFA) if adopted.

Therefore, traditional pattern matching solutions may not apply here.

4.2 Name Prefix Trie for Name Lookup

An NDN name is composed of explicitly delimited components, its hierarchical structure inspire us that it can be represented by Name Prefix Trie (NPT), a data structure very similar to IP Prefix Trie, but it is not necessarily a binary tree. A sample PIT with 9 names is shown in the leftmost part Figure 7, its corresponding NPT is illustrated by the middle part of Figure 7, with its edges standing for name components and nodes representing lookup states. The NPT is of component granularity, rather than character or bit granularity, since the longest name prefix lookup of NDN names can only match a complete component at once, i.e., no match happens in the middle of a component.

Name prefix lookups always begin at the root. When an Interest packet arrives, its name is extracted and the Longest Prefix Match lookup starts on NPT. The process is as follows: we first check if the name's first component matches one of the edges originated from the root node, i.e., the level-1 edge. If so, the transfer condition holds and then the lookup state transfers from the root node to the pointed level-2 node. The subsequent lookup process proceeds iteratively. When the transfer condition fails to hold or the lookup state reaches one of the leaf nodes, the lookup process terminates and outputs the index that the last state corresponds to.

[2] Each bifow is identified by the five tuple of $< src\_ip, dst\_ip, src\_port, dst\_port, protocol >$.

[3] We take 60 snapshots and show the first 20 of them.

Table 1: Trace Statistics.

Snapshot	Un-responded HTTP requests	P2P (TCP)	P2P (UDP)	Multimedia (TCP)	Multimedia (UDP)	Email (SMTP, POP, IMAP)	Online Game	FTP	Conference	Network Management	Services
1	296964	290259	802510	24124	10981	41680	23815	505	32160	1383	245229
2	297903	300391	802879	24159	10920	42447	23382	510	32214	1363	244204
3	295908	306238	806185	23602	10959	42102	22314	504	31872	1416	233500
4	293035	306394	803680	23719	11073	39462	22128	508	31246	1383	234829
5	289425	313038	814347	24478	11238	39529	23014	505	30771	1422	239554
6	296794	314074	802497	24540	11241	40849	23206	505	30528	1383	238147
7	290364	315705	796327	24280	11094	42456	22051	505	30970	1405	238524
8	292213	317899	794001	24691	11227	43039	22536	507	30711	1350	229399
9	289825	318622	796744	23895	11275	42036	22099	504	31290	1414	227809
10	289100	319768	792193	24174	11113	41097	21579	507	30385	1407	227809
11	291704	320269	800449	24066	11295	41310	22426	507	30120	1375	233416
12	284372	315984	808876	24435	11199	39768	22207	510	30375	1401	226281
13	281394	316593	800140	23904	11259	38079	21801	511	28992	1339	221280
14	285902	317544	785419	24022	11140	37252	20238	511	28993	1321	226810
15	282457	317049	798048	24118	11376	37377	21867	519	29445	1365	229740
16	281691	315820	789139	24081	10525	39309	22384	510	29296	1396	223627
17	284215	318627	794436	24280	10450	38898	22311	517	29481	1389	217819
18	283341	321657	809278	24046	10674	39459	22500	511	29532	1434	225202
19	283813	324276	817171	24126	10788	39729	22462	517	29967	1471	225826
20	281163	322878	800040	24111	10702	38428	22774	513	30255	1503	210875

Table 2: Lookup, insert and delete frequency.

Time (s)	1	2	3	4	5	6	7	8	9	10
Read frequency	1,409,082	1,409,477	1,411,578	1,406,273	1,409,961	1,408,013	1,409,447	1,409,731	1,410,433	1,409,555
Insert frequency	90,278	88,513	89,561	90,597	90,376	88,821	90,457	89,828	90,856	91,079
Delete frequency	91,422	92,264	90,296	89,132	90,280	90,614	90,805	90,303	90,475	90,394

4.3 Name Component Encoding (NCE)

NPT makes names in the PIT well organized and manageable, but it cannot contribute much to reducing PIT size or greatly improving PIT access performance.

In this section, we propose the NCE solution to solve the challenges. Each name component is encoded as an integer (code), and the bits allocated for a code is dependent on the amount of integers used. In this paper, a 32-bit integer for each code is sufficient. Therefore, the NPT will be encoded as Encode Name Prefix Trie (ENPT), whose edges stand for codes of name components, as shown by the rightmost part of Figure 7. ENPT is a logical structure and we then construct the State Transition Arrays (STA) – a data structure similar to Adjacency List that stores a graph, but more concise – to implement the ENPT, which significantly shrinks the size of NPT and enables fast lookup, insert and delete. Once an NDN names arrives, it is encoded to a series of codes, and the name is looked up against, inserted to, or deleted from PIT based on these codes. Moreover, a Code Allocation Mechanism is also designed to assign each component a (dynamic) code, which eliminates the potential size explosion problem of PIT.

As Figure 7 shows, the given 9 names can be organized as an NPT with 14 nodes. Different components (edges) leaving the same node should be encoded differently to distinguish themselves. A straightforward method is to assign unique codes to all the components in NPT, ranging from 1 to N, N is the number the edges in the NPT. However, in our solution these unique codes will not help accelerate the PIT access speed. Moreover, since the amount of edges in an NPT can be very large, unique codes will lead to codes of large numerical values and require more bits to store them.

We define the edges in the NPT leaving from the same node as a Code Allocation Set (CAS). which are illustrated by the dotted ellipse on the NPT in Figure 7. We propose that we allocate continuous unique codes within each CAS separately, as depicted by RULE 1.

RULE 1. *Assign each name component in a CAS a unique code. The codes should be as small and continuous as possible within each CAS.*

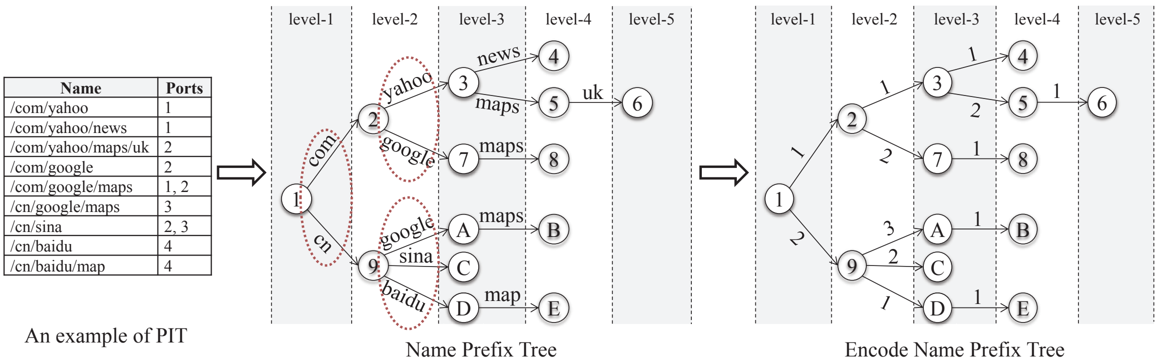
By default, the codes start from 1 within each CAS. After encoding each CAS, we arrive at the ENPT, which is shown by the rightmost part of Figure 7. Suppose that a CAS is composed of edges originated from node i, we denote this CAS as CAS i. By this method, components of the same level but in different CASes may have the same code, e.g., component "yahoo" in CAS 2 and "baidu" in CAS 9 share a common code 1. And the same component in different CASes may be assigned different codes, e.g., CAS 2 and CAS 9 both contain the component "google", but "google" in CAS 2 is assigned code 2 while "google" in CAS 9 is encoded as 3. These two cases, which can be called code assignment collisions, will not bring any negative effects to the name lookup, however. The latter case is in fact how to allocate codes to components, and will be discussed in Section 4.6. Now we assume that each component has been assigned a code based on RULE 1. We then prove the first case will not lead to lookup conflict. (A conflict arises when matching different level-i components of two different names, the lookup states transfer to the same state/node.) The proof is by contradiction.

PROOF. Given two names $C_1C_2 \cdots C_i \cdots C_n$ and $C'_1C'_2 \cdots C'_i \cdots C'_n$ with n level components, and they are encoded to $E_1E_2 \cdots E_i \cdots E_n$ and $E'_1E'_2 \cdots E'_i \cdots E'_n$, respectively. Their corresponding lookup paths are $N_0N_1N_2 \cdots N_i \cdots N_n$ and $N_0N'_1N'_2 \cdots N'_i \cdots N'_n$ (N_i represents a node in the ENPT, and N_0 is the root). Component $C_i \neq C'_i$, and are assigned the same component, i.e., $E_i = E'_i$. Assume that there is a conflict after matching C_i and C'_i, i.e., the lookup states both transfer to the same state/node, thus $N_i = N'_i$. And because (allowing for) $E_i = E'_i$, then $N_{i-1} = N'_{i-1}$. Therefore component C_i and C'_i belongs to the same CAS $i - 1$. According to our code allocation algorithm, each component within a CAS is assigned a unique code, due to $E_i = E'_i$, we get $C_i = C'_i$, which contradicts previous assumption $C_i \neq C'_i$. □

4.4 State Transition Arrays for Encoded Name Prefix Trie

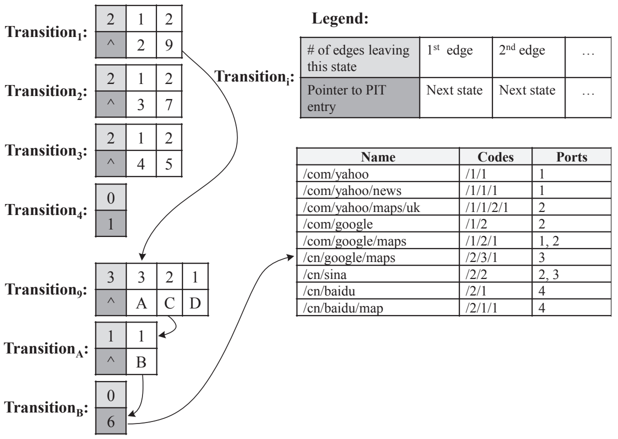
ENPT is a logical data structure, and is implemented by State Transition Array (STA), as depicted in Figure 8. (CAS i corresponds to Transition$_i$ of STA.) STA is composed of Transition arrays, each array of them, say Transition$_i$, stands for a state i and its children in the ENPT, as well as edges originated from state i. If there is a name prefix match at state i, a pointer to the corresponding PIT entry is included in Transition$_i$.

Figure 8 also shows the process of looking up "cn/google/maps", which is encoded as "2/3/1". The lookup process always begins at the root node, i.e., Transition$_1$. Searching in Transition$_1$ for code 2, the next state, 9, is obtained. Then we transfer to Transition$_9$ and continue to search for the second code – 3. The lookup process iterates like this and finally reaches Transition$_B$, where the pointer to the PIT is stored. By now, we successfully finds the PIT entry and the lookup process terminates. This lookup process make use of the codes (integers) to find a valid transfer from one state to another, compared to matching a component (character string) of variable length, matching an integer is much more easier.



Name	Ports
/com/yahoo	1
/com/yahoo/news	1
/com/yahoo/maps/uk	2
/com/google	2
/com/google/maps	1, 2
/cn/google/maps	3
/cn/sina	2, 3
/cn/baidu	4
/cn/baidu/map	4

Figure 7: PIT, NPT and Encode NPT (ENPT).

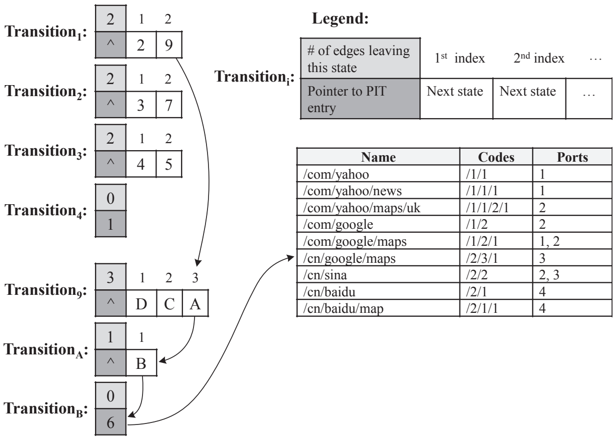


Transition$_1$: 2 1 2 / ^ 2 9
Transition$_2$: 2 1 2 / ^ 3 7
Transition$_3$: 2 1 2 / ^ 4 5
Transition$_4$: 0 / 1
Transition$_9$: 3 3 2 1 / ^ A C D
Transition$_A$: 1 1 / ^ B
Transition$_B$: 0 / 6
Legend:
Transition$_i$:

# of edges leaving this state	1st edge	2nd edge	…
Pointer to PIT entry	Next state	Next state	…

Name	Codes	Ports
/com/yahoo	/1/1	1
/com/yahoo/news	/1/1/1	1
/com/yahoo/maps/uk	/1/1/2/1	2
/com/google	/1/2	2
/com/google/maps	/1/2/1	1, 2
/cn/google/maps	/2/3/1	3
/cn/sina	/2/2	2, 3
/cn/baidu	/2/1	4
/cn/baidu/map	/2/1/1	4

Figure 8: The STA data structure.



Transition$_1$: 2 1 2 / ^ 2 9
Transition$_2$: 2 1 2 / ^ 3 7
Transition$_3$: 2 1 2 / ^ 4 5
Transition$_4$: 0 / 1
Transition$_9$: 3 1 2 3 / ^ D C A
Transition$_A$: 1 1 / ^ B
Transition$_B$: 0 / 6
Legend:
Transition$_i$:

# of edges leaving this state	1st index	2nd index	…
Pointer to PIT entry	Next state	Next state	…

Name	Codes	Ports
/com/yahoo	/1/1	1
/com/yahoo/news	/1/1/1	1
/com/yahoo/maps/uk	/1/1/2/1	2
/com/google	/1/2	2
/com/google/maps	/1/2/1	1, 2
/cn/google/maps	/2/3/1	3
/cn/sina	/2/2	2, 3
/cn/baidu	/2/1	4
/cn/baidu/map	/2/1/1	4

Figure 9: The Simplified STA (S^{2}TA) data structure.

However, this matching method still requires linearly searching a code in a Transition$_i$, which brings low frequency. To relieve this problem, we further propose to directly locate the next state by the code, videlicet, taking code as an index of the Transition array. By this scheme, the STA data structure is simplified to what is shown by Figure 9, and we name it Simplified STA (S^{2}TA). The codes are no longer stored in the STA, but act as the indexes to locate the next lookup state. The lookup process now can simply be implemented by sequentially accessing four Transition elements: Transition$_1[2]$, Transition$_9[1]$, Transition$_A[1]$ and the *Pointer to PIT entry* field of Transition$_B$. Significant advantages of this scheme include: 1) no need to move data when inserting and deleting names, 2) no complicated memory management involved. By this means, the required storage by S^{2}TA is reduced compared to the STA in Figure 8, and simultaneously the access frequency is markedly improved! Therefore, PIT is conceptually transferred to ENPT, and eventually implemented by S^{2}TA.

However, this method calls for strict requirements on the codes, which should be as continuous as possible and starting from a code of a numerical value as small as possible. Otherwise the memory consumption of PIT can be enormous. We address this problem in next subsection.

4.5 Dynamic Code – A solution to potential PIT explosion

We have demonstrated the benefits that NCE and S^{2}TA bring, but there is still one problem before the actual deployment of PIT. Because PIT is quite dynamic, though the number of PIT entries is relatively stable, the names are inserted (Interest arrives) and deleted (Data arrives) at a high frequency, which has been shown by the evaluation results in Section 3.2. Names arrive disorderly, therefore as well as the name components since names are composed of components. RULE 1 implies that, within each CAS, the code assigned to a specific name component is unique, as well as *consistent*. Assume that a name component C_m of CAS i arrives, and C_m has a consistent code 1, then Transition$_i$[1] in the S^{2}TA will be occupied by the *next state* of C_m. Immediately following C_m, another component C_n in CAS i also arrives, which has a consistent code 1000, and Transition$_i$[1000] will also be occupied by the *next state* of C_n. Therefore, elements from Transition$_i$[2] to Transitions$_i$[999] are all wasted. The worst case is, names are all composed of new components, if we assign a consistent and unique code to each component, the numerical value of codes will increase to be extremely large. Because we utilize codes as indexes of the Transitions in S^{2}TA, after a specific name is deleted, the corresponding elements in S^{2}TA can not be used by other names, cumulatively the actual memory consumed by PIT will be quite huge!

Due to the above situation, though PIT contains 1.5 M valid entries, the memory actually consumed by the S^{2}TA can be extremely large. Therefore, only using RULE 1 is impractical to deploy. To address this problem, we propose assigning *dynamic* codes to components, which is summarized as RULE 2.

RULE 2. *Assign each name component in a CAS an available code. An available code means this code has not been assigned to a name component, or is freed by a leaving name component.*

RULE 2 also ensures the code assigned to each name component in a CAS is unique. RULE 2 proposes that we assign dynamic (and maybe different) codes to the same component at same level in the ENPT while they arrive at different time. In fact, we can view each CAS as a code pool. When an Interest packet arrives, we encode its name by selecting the available codes in the CAS. Assume that the encoding function is f, which takes component as one of its parameters and returns a code. (f will be discussed in detail in Section 4.6.) Each time f is called for each component of the **Interest name**, it picks up the smallest available code for this component

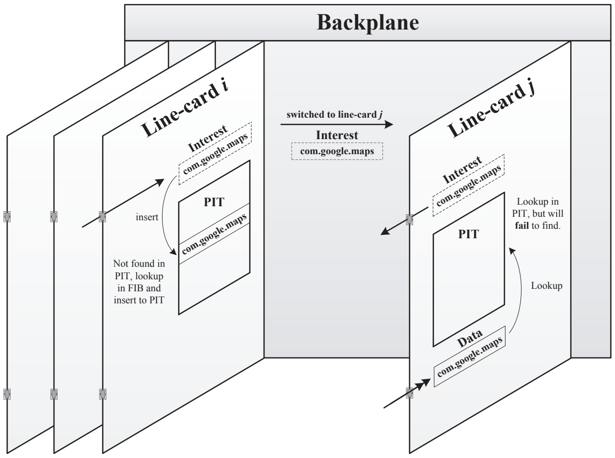


Figure 10: Place PIT on incoming line-cards.

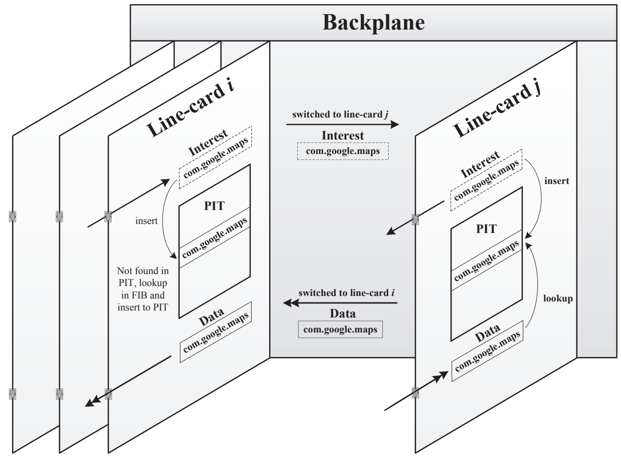


Figure 11: Place PIT on both incoming and outgoing line-cards.

and writes the <component, value> pair to a hash table. As the responding **Data** packet returns, f returns the same code series for its name. After the Data packet name is looked up and deleted from the S^{2}TA based on its code series, all the codes are freed and denoted as available. These codes will be reused for subsequent Interest and Data packets. An example is as follows (refer to Figure 7): an Interest with name "cn/sina" comes, and its name is encoded to "2/2". After the responding Data packet returns, the second level code (2) in CAS 9 is freed (assume that the first level code, 2, is occupied by another names, such as "cn/google/maps"). When another Interest name comes, e.g., "cn/yahoo", "cn" is still encodes as 2. For "yahoo", if consistent code allocation is adopted, it should be encode to 4. However, by dynamic code method, we find that in CAS 9, code 2 is an available code, and consequently assign 2 to "yahoo". It's worth pointing out that, a freed code also indicates that the corresponding element of the CAS in S^{2}TA is vacant, and thus this element can be reused to save memory consumption. Evaluation results show that the dynamic code method effectively reduces the amount of codes within each CAS and makes the codes as continuous as possible. Therefore, the numerical value of the largest is reduced, as well as the memory actually consumed by the STA.

Adopting RULE 2 involves searching for an available code for a component within a CAS, which will bring extra time cost compared to consistent code allocation. Assume that the largest code of a CAS is N, then the worst time complexity of searching a CAS is $O(N)$. However, evaluation results show that 75% search operations successfully return after only one try (edges to leaf nodes in ENPT), and 10% search operations successfully return with less than 5 tries. At last the average time complexity is approximately $O(\frac{N}{4})$.

4.6 Code Allocation Function *f*

Previously, we assume that the codes, either consistent or dynamic code, are correctly assigned to components. In Section 4.3, we have found that identical component may be assigned multiple codes. However, given a specific component in a specific name, only one code is correct for that component. In this subsection, we will present how to allocate correct codes to components, videlicet, how to implement the function f mentioned in Section 4.5.

Based on the fact that components of domains are separated by special delimiters, we can get which level a given component belongs to. We define a function $f(component, level, preceding\_code)$ that maps a component to its appropriate code by a hash function, which is borrowed from Python. f takes three parameters, the first is the component that is to be assigned a code, the second is its level in the whole name, and the third is the code of its preceding component. f returns the correct code of current component by hashing. If current component is the first component, $preceding\_code$ is set to 0. Note that f behaviors differently for Interest name and Data name. For Interest name, f assigns available codes to components within each CAS separately. As a result, identical component in different CASes may have different codes. For Data name, f is responsible for finding the correct code for a component that has multiple codes. An example of code assignment is as follows. Suppose that the looked up name is still "cn/google/maps" (refer to Figure 7). First we take "cn/google/maps" as an Interest name. "cn" is encode as 2 by invoking $f(\text{“}cn'', 1, 0)$, then $f(\text{“}google'', 2, 2)$ is invoked to encode "google", the second argument, 2, indicates that this is a second level component, and the third argument, also 2, indicates the branch or subtree that current component belongs to, and further figures out which CAS this component belongs to. In this example, "google" belongs to CAS 9. Assume that the code 3 is available within CAS 9, thus f returns 3 as the code of "google", and writes 3 to the appropriate entry of the hash table. Similarly, "maps" is encoded to 1 by invoking $f(\text{“}maps'', 3, 3)$. Assume that the arrival of another Interest name, say "com/google/maps", makes "google" be assigned another code 2 in CAS 2, the code 2 is appended to the appropriate hash table entry. Consequently, as a level-2 component, "google" has two corresponding codes – 2 and 3.

Next we take "cn/google/maps" as a Data name. Obviously, $f(\text{“}cn'', 1, 0)$ encodes "cn" to 2. $f(\text{“}google'', 2, 2)$ first hashes "google" to the appropriate entry of hash table and finds that "google" has two corresponding codes (2 and 3), then f recognizes that this "google" belongs to CAS 9 and returns code "3". (Of course, information about which CAS each code belongs to is also maintained.) At last, $f(\text{“}maps'', 3, 3)$ returns 1.

Though at level 2, "google" has two corresponding codes (2 and 3), but this will not bring any negative interference when selecting a correct code for "google". A potential drawback is, the code of the i-th component depends on the code of the $(i-1)$-th component, which makes the encoding process of each component in a name sequentially executed and may degrade the throughput. However, evaluation results show that, by a four-module accelerated hash-based encoding function f, this is not the bottleneck of the system.

5. WHERE PIT RESIDES?

Figure 1(a) and Figure 1(b) illustrate the Interest and Data packet forwarding process within a router, from which the PIT lookup and update process can be derived as well. NDN takes PIT as a global table and conceptually assume that all the Interest and Data packets can access that table, which is, however, impractical to implement in current router architecture. The router architecture is illustrated in Figure 10, with multiple line-cards plugged in the backplane.

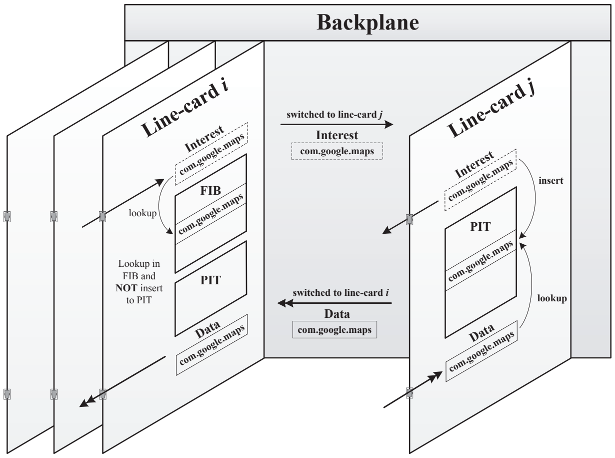


Figure 12: Place PIT on outgoing line-cards.

Packets are received by line-cards and are switched to another line-card via the backplane and switch fabric (not shown). Conceptually, a router is composed of two parts: data plane and control plane. Data plane is simply the line-cards, and the control plane is a CPU that draws network topology and computes FIB. For brevity, control plane is not shown in Figure 10.

When implementing PIT, a fundamental problem is where and how to reside PIT in current router architecture.[4] Obviously, it's impossible to place PIT on router's control plane since delivering every Interest and Data packet into control plane is not a wise idea and costs a fortune. Therefore, we place the PIT on the data plane (line-cards).

The most straightforward way is that each line-card maintains its own PIT – creates a corresponding PIT entry for every incoming Interest packet and removes the associative PIT entries for received Data packets, as shown by line-card i in Figure 10. However, this means will encounter serious problems. The first problem is, suppose that an incoming Interest packet I arrives at line-card i, and line-card i looks it up in its PIT first. If not found, an entry is inserted into i's PIT for this Interest. Then the Interest is switched to line-card j and further forwarded to the downstream routers. When the responding Data packet returns, due to the symmetric routing property, the Data packet will be received by line-card j. However, line-card j fails to find the Data packet in its PIT because the corresponding PIT entry is on line-card i, thus this Data packet does not know where to go and will be discarded. The second is problem, imagine that immediately after Interest I arrives at line-card i (I is not responded yet), an identical Interest I' arrives at line-card k for the first time, and line-card k looks it up in its PIT, and will certainly not find it since line-card k has never received an Interest identical to I' before. But according to the design of NDN, Interest I' should be found in PIT since an identical Interest I has been received by this router and is not responded yet. If we really want this, line-card i has to send I's corresponding PIT entry to all the other line-cards for synchronization after the entry is created, which will definitely consume a lot of resources and incur extra burden on the router.

A possible solution is to create entries for an Interest on both incoming and outgoing line-cards, as illustrated in Figure 11. When an Interest packet arrives at this router, line-card i and j both insert its carried name into their own PITs. When the responding Data packet comes back to line-card j, line-card j looks up the name of the Data packet in its PIT, obtains the proper destination interface and forwards it to line-card i. Thus this method solves the first aforementioned problem. But for the second problem above, Interest I' still cannot be found in line-card k's PIT, and will be forwarded to line-card j. However, if we lookup I' in line-card j's PIT, I' is there! This fact inspires us to lookup Interest names against the PITs on the outgoing line-cards. Therefore, the PIT entries on the incoming line-cards are over-provisioned. In Figure 11, the PIT entry for the incoming Interest "*com.google.maps*" on line-card i is redundant since it will never be looked up.

Consequently, we propose to only place the PIT entries on (egress channel of) the outgoing line-cards. This means each line-card only have to create PIT entries for outgoing Interests (in the egress channel) that are switched to itself from other line-cards, rather than the incoming Interests (in the ingress channel) from the outside, as depicted by line-card j in Figure 12. By this design, the Interest lookup and forwarding process slightly changes, as illustrated in Figure 1(c). (For brevity, the CS, which is a global buffer shared by all the line-cards, is not shown in Figure 12.) For a Interest packet that comes in through interface x of line-card i and goes out through interface y on line-card j:

1. line-card i checks if CS has cached the desired data chunk, if so, returns a copy by a Data packet,
2. otherwise, looks up the Interest against FIB for the outgoing interface y, and switches it to line-card j,
3. line-card j checks if PIT has an entry for this Interest, if so, appends x to this entry and discards this Interest,
4. otherwise, creates a PIT entry and fills it with Interest name and the arrival interface x. Then forwards the Interest to the interface y.

Data packet is looked by by the line-card that receives it, and the lookup and forwarding process does not change.

6. EVALUATION

6.1 Experimental Setup

We measure the performance of NCT and ENCT on the platform: Intel Xeon E5520, 2.27 GHz, 15.9 GB RAM.

The one-hour trace is captured from a 20 Gbps gateway link in CERNET, 17:00∼17:59, Dec. 21st, 2011. The domain names are collected from ALEXA, DMOZ and our web crawler, and 9,834,747, about 10 million, domain names are collected in total. We also extracted 7,624,393, around 8 million, real URLs from the HTTP GET and HEAD requests in the trace. For brevity, we refer to these two name sets by 10M Name Set and 8M Name Set, respectively. The statistics of these two Name Sets are presented in Table 3.

The evaluation can be generally divided into two parts: 1) The size and access frequency of PIT, which have been previously illustrated by Table 1 and Table 2 in Section 3; 2) The performance of our proposed encoding-accelerated PIT access scheme (NCE and S^{2}TA), such as memory consumption, access frequency, and comparison with other methods.

6.2 Evaluation Results

6.2.1 Memory Usage

For the two Name Sets, we first measure their memory consumption of the: 1) original size, i.e., directly store the names as character strings in a table, 2) NPT, 3) ENPT (S^{2}TA) + hash table. (Hash table is required by function f.)

The overall results are shown in Table 3, more detailed results are given in Figure 13 and Figure 14. Some facts can be derived from these statistics. By comparing the *# of edges/components* in the NPT with the *# of total components* in Table 3, we find that 8M Name Set is more aggregatable than the 10M Name Set. Moreover, besides

[4] Though NDN is a clean-slate network architecture, we believe that it will not make significant modifications to the router architecture.

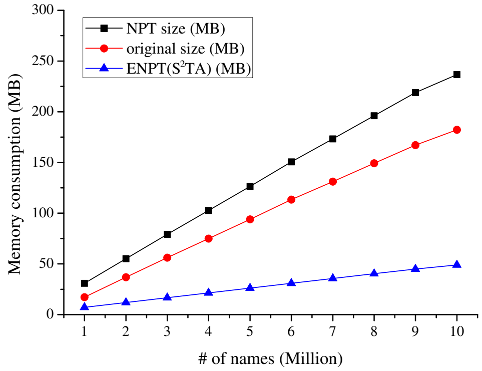


Figure 13: 10M Name Set memory consumption–original size, NPT size, ENPT size.

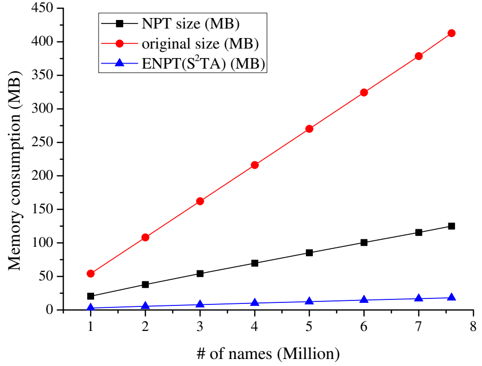


Figure 14: 8M Name Set memory consumption–original size, NPT size, ENPT size.

the name components, each NPT node stores additional information, such as state number, pointer to children, pointer to parent, etc. Therefore, the NPT of 10M Name Set is even larger than its original size. But these additional information makes NPT easier to manage than storing names directly.

As mentioned in section 4.6, we assign codes to components by invoking function f, which in turn resorts to a hash function. Thus a hash table is required, for the 10M Name Set, the size of hash table is approximately 67.11 MB, and around 33.55 MB for 8M Name Set. At last, the compression ratio of the 10M Name Set and 8M Name Set is 63.66% and 12.56%, respectively.

Please keep in mind that, ENPT is a logical data structure and we do not implement it directly. However, it is implemented by S^{2}TA. We can think of operations on the ENPT, but we actually operate the S^{2}TA to carry out those operations.

6.2.2 Lookup, insert and delete performance

Subsequently, we measure the PIT access (lookup, insert and delete) performance by ENPT. We do not measure the *update* performance, because updating PIT means appending interfaces to a specific PIT entry, whose performance it very close to that of lookup.

Deleting a name involves backtracking a path, which is easy to implement by NPT since NPT has parent pointers. The deleting process in ENPT is: first go straight to the leaf node, delete the leaf, then backtrack to see if the parent node still needs to be removed, so and so forth. (If a parent node's all the children are deleted, and it does not have a pointer to the PIT entry, it needs to be removed too.) Because there is no such pointer to parent node in S^{2}TA, we keep track of the node information along the path to a specific leaf when deleting a name. We do not actually delete the node and free the space of ENPT, in fact, deleting a node means the code of its preceding edge/component is freed, and that code can be reused within its CAS, as well as the corresponding space in S^{2}TA. (Refer to function f.)

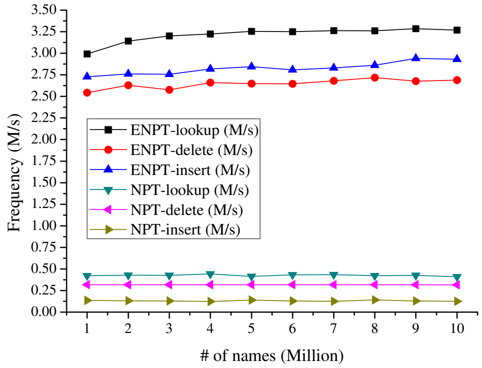


Figure 15: Lookup, insert and delete performance for the 10M Name Set.

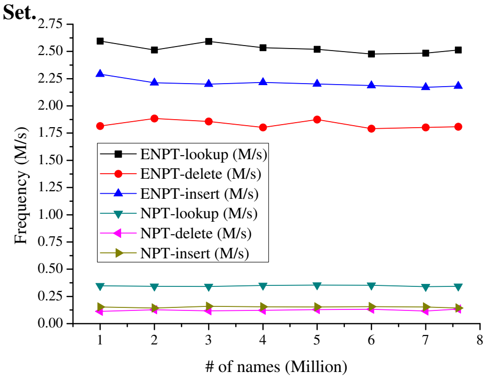


Figure 16: Lookup, insert and delete performance for the 8M Name Set.

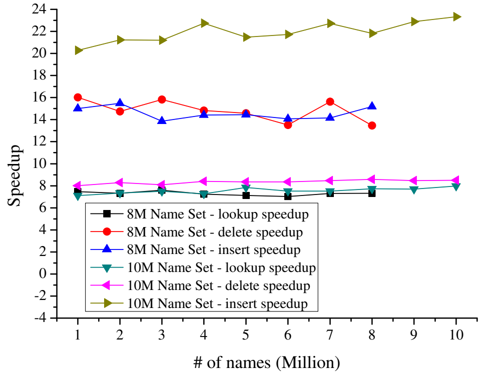


Figure 17: PIT access frequency speedup based on ENPT.

The lookup, insert and delete performance for the two Name Sets are illustrated in Figure 15 and Figure 16, respectively. The lookup, insert and delete performance on the 10M Name Set can achieve 3.27 M/s, 2.93 M/s and 2.69 M/s, respectively, and that on the 8M Name Set achieve 2.51 M/s, 1.81 M/s and 2.18 M/s, respectively.

Obviously, the PIT access frequency based on ENPT is phenomenally promoted compared with that of the NPT, which is more clearly depicted by the speedups in Figure 17.

6.2.3 Code allocation function f

Each incoming name will be decomposed to multiple name components and each component will be assigned a code by invoking function $f(component, level, preceding\_code)$, which in turn calls a hash function. The hash function is borrowed from Python. The way how f works has been discussed in Section 4.6. Because this step of encoding components is just right before the step of lookup, insert or delete, the encoding performance should be no less than the lookup, insert or the delete performance. Otherwise, the encoding step will be the performance bottleneck. We utilize 4 parallel modules to improve the encoding throughput, and the performance

Table 3: Name Component Statistics of Two Name Sets.

URL set	# of names	# of total components	average component length (Byte)	average # of components per name	original size (MB)	# of components /edges in NPT	NPT size (MB)	ENPT size (MB)	ENPT+Hash Table size (MB)	compression ratio
10M Name Set	9,834,747	24,808,603	7.35	2.52	182.26	12,228,081	236.57	48.91	116.02	63.66%
8M Name Set	7,624,393	26,882,827	15.35	3.53	412.78	4,570,563	125.03	18.28	51.83	12.56%

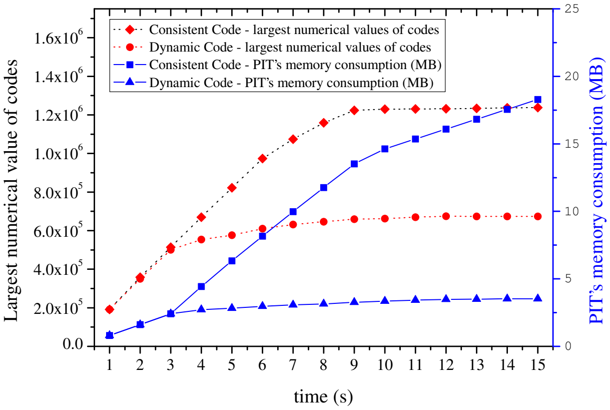


Figure 18: Largest numerical values of codes and PIT's memory consumption.

is 20.67 M components per second and 18.96 M components per second for the 10M Name Set and 8M Name Set, respectively. Divide them by the average number of components per name, we further compute the encoding performance: 8.20 M names per second and 5.37 M names per second for the 10M Name Set and 8M Name Set, respectively. Therefore, the encoding performance is better than the lookup performance and will not be the performance bottleneck.

6.2.4 Results of dynamic code

The drawback of assigning consistent codes to components has been discussed in Section 4.5. To demonstrate the effects of dynamic code, we replay the HTTP packets in the captured trace to mimic the packet (name) incoming and outgoing process, which will lead to a PIT of around 300 K (refer to Table 1) entries, and measure how large the numerical value of the codes will be. For comparison, both consistent and dynamic code method will be measured, as well as the PIT's actual memory consumption. The result is shown by Figure 18, the dotted curves represent the largest code of all the CASes, which show that as times goes on, names keep coming and going, the largest code increases. For consistent code, the largest keeps increasing at a high rate after the PIT reaches 300 K valid entries, thus the consumed memory of PIT increases as well. However, for dynamic code, after the PIT reaches 300 K valid entries, the largest code greatly slows down its increasing pace, making PIT's memory consumption remains stable (solid curves). In fact, the largest code by RULE 1 at each snapshot is the number of total components observed by a CAS until this snapshot, while the largest code by RULE 2 is the amount of components a CAS contains at each snapshot. The PIT's memory consumption exhibits similar growth law of the codes. The hash table size (not shown in Figure 18) is almost the same for both consistent and dynamic code methods, since the received names are the same, and thus the name components. The hash table size is 33.55 MB (for the 8M URL Name Set). Therefore, with NCE and dynamic code, PIT exhibits good scalability.

7. RELATED WORK

This section compares our NCE solution to our previous work in [14], and we name it Original NCE. In fact, this paper only continues the encoding idea, but the ways to assign codes, lookup, insert and delete are different. We conclude three major distinctions: 1) The data structure to implement the ENPT in Original NCE involves complicated memory management, such as data movement, fragment management; 2) Original NCE allocates consistent codes to components and does not allow identical components be encoded to different, dynamic codes, which fundamentally contradicts with the code allocation function f in this paper; 3) Original NCE only achieves lookup speedup, but does not exhibit good support for insert and delete operations. However, in this paper, NCE not only significantly accelerates lookup, but also insert and delete operations.

8. CONCLUSION

NDN/CCN propose that PIT caches yet un-responded Interests, when the responding Data packets return, the names are removed from PIT. PIT brings significant features to NDN/CCN. However, none has conducted a measurement study to show the size and access requirements of PIT. Without these knowledge, we have no data to support the design of NDN routers or the actual deployment of NDN. In this paper, we are the first address three problems associated with the PIT: 1) the size and access (lookup, insert, delete) frequency of PIT; 2) how to address the large size and high access frequency problem with a scalable solution; 3) where does PIT reside within a router.

We emulate NDN's application-layer working paradigms by transferring the existing IP applications to the NDN platform. By mapping/translating a captured 20 Gbps gateway trace from IP to NDN scenario at the application perspective, we quantify the size and access frequency of PIT, which demands an efficient and scalable solution. Therefore, NCE is proposed to accelerate the access throughput of PIT, as well as to reduce its size. Moreover, the dynamic code allocation technique makes the NCE solution practical, and further keeps the actual memory consumption of PIT stable. At last, we propose to place PIT on the packets' outgoing line-cards (egress channel) when actually implementing PIT, which meets the PIT design in [15] and eliminates the cumbersome synchronization problem among multiple PITs on line-cards.

9. REFERENCES

[1] HTTP/1.1 RFC. http://www.ietf.org/rfc/rfc2616.txt.

[2] l7-filter. http://l7-filter.clearfoundation.com.

[3] TIE. http://tie.comics.unina.it.

[4] Tstat. http://tstat.tlc.polito.it.

[5] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the internet. In *Proc. of SIGCOMM*, 2004.

[6] A. Dainotti, W. de Donato, and A. Pescapé. TIE: A community-oriented traffic classification platform. In *TMA'09*, May 2009.

[7] C. Esteve, F. L. Verdi, and M. F. Magalhaes. Towards a new generation of information-oriented internetworking architectures. In *Proc. of ACM CoNEXT*, 2008.

[8] A. Finamore, M. Mellia, M. Meo, M. M. Munafò, P. di Torino, D. Rossi, and T. ParisTech. Experiences of internet traffic monitoring with tstat. *IEEE Network*, 25(3), May-June 2011.

[9] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, Sep 1960.

[10] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, and R. Braynard. Networking named content. In *Proc. of ACM CoNEXT*, 2009.

[11] H. Jiang and S. Jin. Exploiting dynamic querying like flooding techniques in unstructured peer-to-peer networks. In *Proc. of IEEE ICNP*, 2005.

[12] A. Kumar, J. J. Xu, and E. W. Zegura. Efficient and scalable query routing for unstructured peer-to-peer networks. In *Proc. of IEEE INFOCOM*, 2005.

[13] S. Nilsson and G. Karlsson. IP-Address Lookup Using LC-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, JUNE 1999.

[14] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen. Scalable name lookup in ndn using effective name component encoding. In *Proc. of IEEE ICDCS*, 2012.

[15] L. Zhang, D. Estrin, V. Jacobson, and B. Zhang. Named Data Networking (NDN) Project. In *Technical Report, NDN-0001*, 2010.