Hadoop on Named Data Networking: Experience and Results

MATHIAS GIBBENS, CHRIS GNIADY, LEI YE, and BEICHUAN ZHANG,
The University of Arizona

The Named Data Networking (NDN) architecture retrieves content by names rather than connecting to specific hosts. It provides benefits such as highly efficient and resilient content distribution, which fit well to data-intensive distributed computing. This paper presents and discusses our experience in modifying Apache Hadoop, a popular MapReduce framework, to operate on an NDN network. Through this first-of-its-kind implementation process, we demonstrate the feasibility of running an existing, large, and complex piece of distributed software commonly seen in data centers over NDN. We show advantages such as simplified network code and reduced network traffic which are beneficial in a data center environment. There are also challenges faced by NDN, that are being addressed by the community, which can be magnified under data center traffic. Through detailed evaluation, we show a reduction of 16% for overall data transmission between Hadoop nodes while writing data with default replication settings. Preliminary results also show promise for in-network caching of repeated reads in distributed applications. We also show that overall performance is currently slower under NDN, and we identify challenges and opportunities for further NDN improvements.

Additional Key Words and Phrases: Named Data Networking; Large-scale systems; Hadoop; Data centers; Emerging technologies

ACM Reference format:
Mathias Gibbens, Chris Gniady, Lei Ye, and Beichuan Zhang. 2017. Hadoop on Named Data Networking: Experience and Results. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 1, Article 1 (June 2017), 21 pages.
https://doi.org/http://dx.doi.org/10.1145/3084439

1 INTRODUCTION

In today's data centers, clusters of servers are arranged to perform various tasks in a massively distributed manner: handling web requests, processing scientific data, and running simulations of real-world problems. These clusters are very complex, and require a significant amount of planning and administration to ensure that they perform to their maximum potential. Planning and configuration can be a long and complicated process; once completed it is hard to completely re-architect an existing cluster. In addition to planning the physical hardware, the software must also be properly configured to run on a cluster. Information such as which server is in which rack and the total network bandwidth between rows of racks constrain the placement of jobs scheduled to run on a cluster. Some software may be able to use hints provided by a user about where to schedule jobs, while others may simply place them randomly and hope for the best.

2476-1249/2017/6-ART1 $15.00
https://doi.org/http://dx.doi.org/10.1145/3084439

Every cluster has at least one bottleneck that constrains the overall performance to less than the optimal that may be achieved on paper. One common bottleneck is the speed of the network [1]: communication between servers in a rack may be unable to saturate their network connections, but traffic flowing between racks or rows in a data center can easily overwhelm the interconnect switches. Various network topologies have been proposed to help mitigate this problem by providing multiple paths between points in the network, but they all suffer from the same fundamental problem: it is cost-prohibitive to build a network that can provide concurrent full network bandwidth between all servers. Researchers have been working on developing new network protocols that can make more efficient use of existing network hardware through a blurring of the line between network layer and applications. One of the most well-known examples of this is *Named Data Networking* (NDN) [7] [22], a data-centric network architecture that has been in development for several years [20].

While NDN has received significant attention for wide-area Internet, a detailed understanding of NDN benefits and challenges in the data center environment has been lacking. The questions that have been researched for Internet deployment of NDN need to be answered for the data center. Will NDN offer benefits for data center applications? If so, the data center can become a new venue for potential benefits brought by NDN. A prime example of a large data center application is Apache Hadoop [18], a widely-used open-source MapReduce framework and distributed file system for use in large, massively-distributed computation clusters. It is used extensively for solving problems that can be divided into arbitrarily-sized units of independent work, such as sorting, searching, data mining and machine learning.

To a large distributed system like Hadoop, network performance, reliability, and adaptation to failures are at the core of overall performance [2]. This makes Hadoop a prime choice for evaluating NDN's performance when used by a complex, distributed system. In this paper, we will focus specifically on the distributed file system and data movement among nodes to study and understand how NDN impacts data movement in data center applications. We make the following contributions: (1) We redesign Hadoop to use NDN for all of its network communication; (2) We discuss and identify benefits and challenges to implementing Hadoop over NDN; (3) We show pilot research applying the NDN protocol in a data center-scale distributed computing and storage environment; and (4) We present a detailed evaluation of the benefits and challenges of such an implementation.

2 MOTIVATION

NDN has several features, such as in-network caching, native multicast, and in-band failure detection and recovery, which can potentially benefit data center networks significantly. Popular applications, such as Hadoop, can take advantage of the capabilities of NDN to improve their performance and resilience.

2.1 Hadoop

Hadoop is a complex system with many types of nodes, although they can be generally categorized as master or slave nodes as shown in Figure 1. A cluster can consist of only a few nodes, or even upwards of hundreds of thousands of nodes. Master nodes, such as the *JobTracker*, *NameNode*, and *Secondary NameNode*, maintain the overall state of the cluster and perform tasks such as scheduling jobs on different nodes or ensuring a consistent global view of the *Hadoop Distributed File System* (HDFS). Slave nodes, such as the *DataNode* and the *TaskTracker*, store data in the cluster or perform the actual computation. There may be any number of slave nodes, each of which receives instructions from a master node, such as reading and storing data, running a task, or reporting their status. The JobTracker and TaskTracker nodes are involved with the scheduling and execution of MapReduce tasks in the cluster, while the NameNode and DataNodes are responsible for storing data within HDFS. For the remainder of this paper,

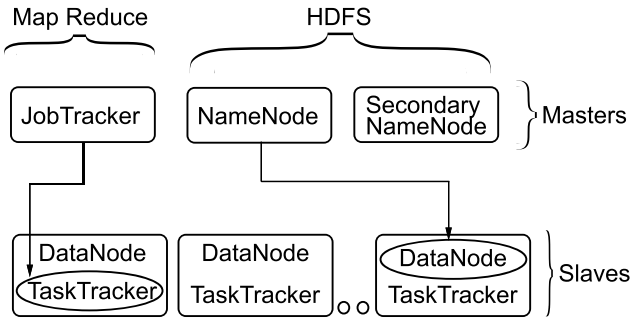


Fig. 1. Types of nodes in a Hadoop cluster.

we will focus specifically on the data storage side of Hadoop and therefore will deal primarily with the NameNode and DataNode components.

Hadoop was designed with the core assumption that failures will occur while a cluster is running [19]. Especially with a large number of nodes, even very low failure rates will still result in the regular failure of nodes across the cluster. Hadoop relies on detailed monitoring of node state using heartbeat messages, which, if not received, will trigger an automatic task restart on a different node and transparent replication of data stored on the failed node. The default replication of data to three DataNodes ensures that data is not lost if a node fails. While these mechanisms were designed to provide fault resiliency, they can have a significant impact on performance. For example, a network delay or temporary link failure can delay heartbeats and trigger unnecessary task recovery and data replication, further increasing network congestion. To account for this, there is an adjustable timeout before the recovery operation begins. However, increasing this timeout can delay recovery of actual failures. A more intelligent and adaptive network protocol could alleviate this situation.

Hadoop's ability to maintain data integrity even as nodes fail comes at a price: a significant amount of state needs to be maintained by the NameNode (excess code complexity) and a large amount of redundant data is transmitted when replicating stored data (network hot spots can easily form and overwhelm the data center network). For Hadoop to achieve maximum performance, its network must be properly configured. Hints can be given about where nodes are located in the network topology, but this is not an automatic process. The cluster administrator must manually assign each node to a rack or row before Hadoop can use the information for scheduling tasks in close proximity to existing data or other resources. Therefore, large clusters require very careful network planning to provide optimal performance. This configuration is complex and relies on many different variables being set properly [3], as misconfiguring a cluster can significantly impact its performance. Again, a more intelligent network protocol could reduce the configuration complexity and even offer adaptive mechanisms to changing network conditions that maximize performance under current network load and configuration.

Even a properly optimized network can pose a limit for scalability of a distributed system. Adding more nodes to a cluster will result in a near-linear increase in computational power; however, eventually a point will be reached where the network will become the bottleneck, causing overall system performance to level out, or even decrease. Mehrotra et al [11] explored the direct impact of network communication on the overall speed of two clusters, one in Amazon's EC2 and the other at NASA. Overall, NASA's cluster performed better, as it utilized a faster InfiniBand network, rather than 10GigE Ethernet. Illustrating this were results showing that adding additional nodes to the EC2 cluster began to result in worse throughput. This resulted from the communication delay becoming greater than the computation time. Switching to a

faster network alleviated this problem. While upgrading a cluster's network can reduce the bottleneck, it is an expensive undertaking, due to labor and hardware costs. A more efficient network protocol could be deployed on existing hardware to improve efficiency.

Similarly, Hadoop has events that can result in significant network congestion. These events are: (1) the beginning of each step in the MapReduce process, during which each node is sent the data it requires to perform the computation; and (2) data replication, during which data is written to the cluster, since, by default, each data block is replicated on three different nodes. A node cannot begin its task until it has all the necessary data, so network congestion and delay can have a dramatic negative impact on the performance of the cluster as a whole. Subsequently, a smarter protocol may be able to find a better path, avoiding congestion and potentially utilizizing different nodes for data retrieval and placement based on network conditions.

2.2 Named Data Networking

The NDN architecture can offer solutions to some of the network challenges encountered in Hadoop clusters. NDN is a *data-centric* architecture, meaning that it focuses on the *what* (content) rather than the *where* (IP address). In NDN, data are referred to by a unique name - if the data changes, so does the name by which it is referred. NDN data are usually versioned by including a component in the name, as demonstrated in the following example: `/foo/bar/v1` and `/foo/bar/v2`. NDN nodes request data by their names rather than directing requests to certain destination nodes, and any node that has the data can respond to the requests. Every piece of data contains a digital signature that binds the data names with its content, and will be verified by the data consumer. This architecture provides three features that can significantly benefit Hadoop:

- *In-Network caching* - The NDN Forwarding Daemon (NFD) at each node automatically caches data sent in response to a request. Data is referred to by a globally unique name that enables transparent data caching in an application-agnostic fashion.
- *In-Network de-duplication* - Each instance of NFD will only forward one copy of a request or response along a given network link. If additional requests for the same content arrive, they can be either served by the local cache, or recorded locally and responded to when the data for the first request returns. This allows data to be multicasted efficiently in the network without any additional protocols like IP does.
- *Fast failure detection and recovery* - In NDN, a request and its response take the same network path. Thus if there is a failure that prevents the response from returning, the network node can quickly detect it after the round-trip-time (RTT) expires, and can then retransmit the request and/or send it onto a different path. This intelligence can greatly benefit applications in data center networks. However, it is lacking in existing TCP/IP networks.

The above can translate to the data movement heavy Hadoop system and can potentially reduce congestion and hotspots in Hadoop networks. Data aggregation in NDN enables more efficient use of network links and in-network caching can help reduce strain on individual nodes that would otherwise be required to serve duplicate requests. Depending upon the caching policy, frequently requested data can remain cached in close proximity to the requesters, even if the original source is far away or overloaded.

NDN also differs from IP in that it uses a "pull" network model where the client always initiates transfer of data with an initial request - it is not possible for a host to "push" data to a host. When a client wants to obtain a piece of data, it expresses an *Interest* that contains the data's name into the network. Because NDN fundamentally uses content names for addressing, there is no concept of a "remote host',' since any host in the network with the corresponding data can return it to the requester. The network then routes this Interest towards host(s) that can provide the requested data. As the Interest passes

through each router, a pending-Interest entry is made so that the response can be properly routed back to the requesting host. Additionally, if more identical Interests arrive (possibly from different hosts), they will be aggregated with the existing pending-Interest entry and will not be forwarded further upstream.

Compared to traditional protocols like IP, NDN moves more logic to the network layer. This additional network logic requires more processing and memory, but provides a smarter and more robust network. Abstracting data control logic down to the network layer allows the application to take advantage of the protocol features and simplify network processing at the application layer. For example, switching to NDN for RPC calls allowed us to remove a significant amount of code related to processing heartbeats that was no longer required when using NDN. This can lead to improved performance at the compute and management nodes and simplify network programming. Both of those features offer tremendous benefits to data center application development and deployment.

2.3 NDN in a data center

To the best of our knowledge, this is the first implementation and evaluation using the NDN protocol in a software project of significant size. While several simple programs have been previously developed to use NDN [21], they have all been fairly small and served as basic proof-of-concept examples. Our work to modify Hadoop to run on NDN is significant for three reasons:

- *Significant and complex code* - Hadoop consists of tens of thousands of lines of code and is very complex. When running, it heavily stresses the network and demonstrates that NDN is capable of handling real life traffic.
- *Explore the use of NDN in a data center* - Research regarding NDN so far has not focused on the domain of the data center. Our work will allow for the identification of potential issues and further avenues of research.
- *Improve quality of NDN code* - As part of the development process, we can provide feedback to the NDN developers as we exercise their code and use it in ways that may not have received much testing in the past. This will extend beyond this project to benefit any other projects that utilizes the NDN protocol.

3 DESIGN

The goal we undertake in this paper is to enable Hadoop to work on top of the NDN network and observe the potential for benefits and further optimizations. Subsequently, we keep our changes to Hadoop minimal, and perform only necessary changes to enable NDN communication. The minimal changes allow us to observe the core benefits and challenges of NDN before we proceed with more advanced optimizations in future work. Better understanding of NDN challenges and benefits will serve as a guide for future development, and will identify the greatest benefits that can be addressed first.

3.1 Hadoop vs. NDN Sockets

Apache Hadoop makes use of traditional `Socket`s and `SocketChannel`s from Java's `Socket` class family for network communication. The NDN API is very different from the `Socket` API in Java, and after a brief bit of exploration it was determined that totally replacing all instances of these classes in Hadoop with direct NDN instances would be much too intrusive and distract us from the goal of studying core NDN functionality. As a result, we developed fairly generic `NDNSocket` and `NDNServerSocket` classes (along with the corresponding `Channel` classes[1]) that can be used as drop in replacements for most code that is

[1]We did have to patch the `sun.nio.ch.{,Server}SocketChannelImpl` classes to be properly visible outside of their package. We are unsure if this is a legitimate bug in the Java distribution or not.

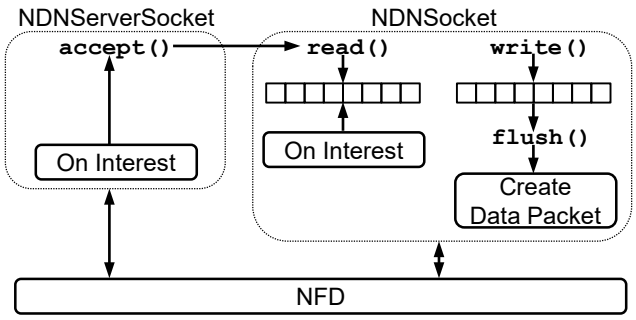


Fig. 2. Diagram of `NDNServerSocket` and `NDNSocket` operation on server side.

already using TCP/IP sockets. The core required methods for `Socket`s are implemented, although some TCP-specific operations are simply ignored in our code; one significant difference between our `NDNSocket` and the traditional implementation is that while both expose `InputStream` and `OutputStream`s, the NDN implementation is not truly a stream of bytes. Data is queued on either end and only transmitted when a call to `flush()` or `close()` occurs. Fortunately, Hadoop largely follows this convention and we only had to make minor changes to accommodate this difference.

Like any network transport protocol, NDN has a maximum packet size. Currently, this is 8,800 bytes in the Java client library. Our implementation automatically handles the slicing of Data packets larger than this into a series of smaller 8KB packets and reassembles them on the remote end. Interests larger than the maximum size are more difficult to handle, but all observed Interests in our Hadoop cluster have been below this limit.

Figure 2 shows how the `NDNServerSocket` interfaces with NFD and client requests on the server side. During startup, `NDNServerSocket` sets up a NDN Interest handler that will queue any incoming Interests routed to it and then enters an `accept()` loop until an Interest arrives. A new `NDNSocket` object is created for the first Interest to arrive from a remote host; any subsequent Interests arriving from the same source are then handled by the same `NDNSocket`, similar to how a normal `Socket` handles a TCP stream. The `NDNSocket` class is more complex and handles most of the NDN logic. As each Interest from a single client is received, its contents are placed into a buffer for reading. Data written to the `Socket` is buffered until it is flushed or closed, at which point a Data packet is created with the response.

On the client side, the handling of Interest and Data packets is reversed: Interests are created based on bytes which are written to the `Socket` and an appropriate NDN URI is generated. Data packets which are returned are read and placed into a buffer which is then exposed for reading from. Because all networks experience some packet loss or corruption, if a Data packet is not received within a timeout a new request will be made automatically. However, after three failures an `IOException` is raised to properly inform the system that something went wrong in the network transfer.

3.2 RPC Conversion

Once we had the basic `NDNSocket` class working, we first looked at converting the RPC calls to use NDN. Fortunately, RPC calls can be mapped directly onto NDN's concept of Interest and Data packets. The RPC setup in Hadoop is built upon a DataNode sending a heartbeat approximately every three seconds to the NameNode and receiving back a response. The NameNode maintains the full state of the HDFS (data replication, write operations, etc.) based on received reports from each DataNode, but does not

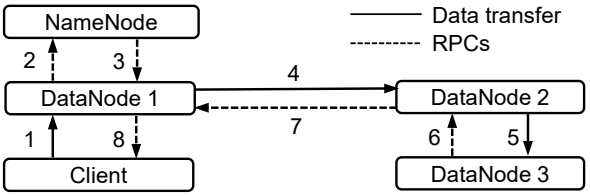


Fig. 3. Data pipelining in Hadoop under TCP.

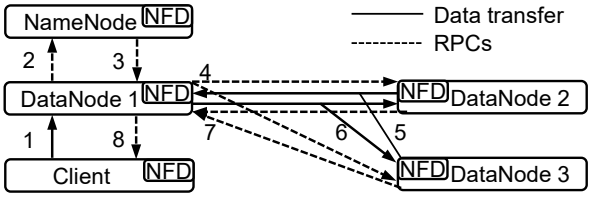


Fig. 4. Flow of data transfer in Hadoop under NDN.

initiate communication with a DataNode. The RPCs are used to send status updates as well as receive new commands from the NameNode.

Because it is critical for the correct host to receive the appropriate RPC, we need to maintain the unique mapping between client and server that exists in TCP/IP. This concept does not exist natively in NDN; we chose to map the IP address and port given to the `Socket` into the second and third NDN URI components. For example, `192.168.0.1:9000` would be mapped to a prefix of `/hadoop/192.168.0.1/9000/`. Additionally, because multiple clients may connect to a given node, each client will also tack on a nonce to the prefix used for all communication while the `Socket` is open. This allows the remote end to maintain proper state for each client, such as when the NameNode last received a heartbeat from a DataNode. Finally, we do not want to cache the results of a RPC call. Consider what would happen if a DataNode received a status of "OK" from the NameNode in response to a heartbeat it sent. Unknown to the DataNode is that the NameNode actually has new instructions for this DataNode, but because of network caching it may be quite some time until the cached response is evicted from all the caches between the DataNode and NameNode. This will cause the NameNode to erroneously mark the DataNode as non-responsive and remove it from the cluster. To avoid this, we append the current Unix time stamp to the NDN prefix. Thus, for a given RPC client, the NDN prefix will look like `/hadoop/192.168.0.1/9000/adadadad/1453480622/`.

Hadoop uses the protobuf library [5] for the encoding and decoding of RPCs. The bytes are then written to or read from the `Socket`. For our implementation, on the calling side, we append the RPC call buffer to the Interest prefix, and then express it to the NDN network. The receiver then picks up the Interest, removes the last component and passes the bytes back to Hadoop to be decoded. When the method is completed and a response is ready to be returned, the bytes are placed in a response Data packet and sent back to the originator to satisfy the Interest.

3.3 Block Transfer Conversion

While the implementation of RPCs over NDN was fairly straight forward, getting Block data transfer working was much more challenging. When writing data to the HDFS, Hadoop uses a pipeline setup between all the DataNodes that will be storing a copy of the data. The developers have spent time highly optimizing this setup to maximize the throughput and speed of the Hadoop cluster. This comes at the cost of being very specific in how Hadoop sends data as well as the sort of issues it will expect to encounter. Introducing NDN to transport data required us to change the network model employed by Hadoop and adjust internal state that is updated when data is sent.

The transfer of data in Hadoop is performed via dedicated TCP streams and optimized pipelines (refer to Figure 3). When new data is written to the cluster, the client first interacts with a DataNode (1), either locally or over the network. The data is then stored on the originating DataNode in temporary storage, but is not yet included in the HDFS. At the next scheduled heartbeat, the node will inform the

NameNode that it has new data that needs to be replicated to other nodes (2). The NameNode will look at the state of the cluster and select additional nodes that should replicate the data. The return message from the master will include a list of other nodes that the DataNode should send the data to (3). The originating node will then construct a pipeline from itself to the first node in its list (4), then that node to the next (5) and so on. The data is then concurrently sent and written to each designated node. The initial write operation does not return success (8) until after every node has confirmed that its own write has been successful (6 & 7).

3.3.1 Changing the network model. The pipeline that Hadoop establishes between DataNodes can be described using a "push" network model: the initial node that has the data to be written is told which nodes it needs to send data to for the cluster to maintain sufficient replication of the data. It then pushes the data to the first node, which then continues pushing the data to the second node, etc. However, NDN operates on a "pull" network model where the client initiates the transfer of data over a network. This change in behavior needs to be addressed.

To handle this change from pushing data to pulling, we modified the behavior of the DataNodes (Figure 4). Now, instead of using the NameNode provided list of DataNodes to setup a pipeline, the originating DataNode will express a ReadCommand Interest to each DataNode in the list it received (4). The content of the Interest is simply the Block's ID that each remote DataNode will then request and store locally. Each DataNode will then express an Interest for the Block it needs to store (5). Initially, only the first node that wanted to write the data will be able to respond (6), since the data only exists on that node in temporary storage. However, as soon as another DataNode starts receiving the Block, it too could potentially serve requests for the pieces of the Block that it has. Furthermore, as the responses make their way through the network, the data can be cached on the routers and be used to answer multiple aggregated Interests. The net result is that data is still replicated from one DataNode to another, but is done in parallel using NDN's pull model. Once all DataNodes have their copy of the Block, they inform the first DataNode (7) which then returns success to the client's write request (8).

The use of an extra ReadCommand Interest can be optimized further by having the NameNode inform each target DataNode to request the Block by using the existing heartbeat system, rather than having the originating DataNode initiate the read. However, this would require the introduction of a new HDFS command, which will be considered in our future work.

3.3.2 Serving Blocks via NDN. Each DataNode registers a special NDN prefix that will be used to transport Block data: `/blocks/`. Unlike the RPC handler, we wish for any node that can provide a Block's data to respond, so the generic prefix is used by all DataNodes. Furthermore, since each Hadoop cluster has a unique HDFS blockpool ID that is included as the second component, in case there are multiple Hadoop clusters on the same NDN network. Finally, the Block ID itself is appended to any Interest requesting that specific data. For example, the full NDN path for a Block is represented as `/blocks/BP-2005120925-192.168.1.11-1444104333081/1073741825`.

The first Interest to request a Block will receive back a Data packet that contains solely the total size of the Block. Based on this information, the DataNode will determine how many 8KB slices are required to fully transfer the data. Multiple Interests are then expressed, each containing the unique slice number appended as the fourth element of the URI. Each DataNode that receives a request will return the slice of data, if it exists locally. If a pending Interest times out, the code will retry up to three times. If it still fails, Hadoop will then leverage its existing error handling code to deal with the failure to write the Block.

When Hadoop writes a Block, it is not marked by the NameNode as "finalized" until all the DataNodes have successfully received the data. Normally this would not be a problem, but since the NDN serving code queries the local DataNode for information about the Block such as size and underlying file name, this

information is not available when needed during an initial Block write. This causes a circular dependency, since writing the Block requires information that cannot be queried until after the Block is written. To handle this, when a DataNode sees that it is creating a brand new Block, it will provide a copy of the bytes in the Block to the NDN serving code before the Block is marked as "finalized" by the NameNode after the necessary DataNodes have replicated the Block. This allows the requests from other DataNodes to complete before the initial write is completed.

The original data pipeline works in a synchronous fashion while the pull model replicates data in parallel. However, Hadoop is still able to maintain its data integrity guarantees. Under normal operation, the write call does not return success until every node along the pipeline has forwarded back its own success to the initial DataNode. Under the new approach, the initial DataNode still waits to receive a success message from each DataNode that was to replicate a Block. If this does not happen for some reason, Hadoop handles the failure just as it would if a DataNode failed during a normal pipeline replication.

One surprising discovery made while converting Block transfer to NDN was that the `DFSOutputStream` class contained a subclass that is used to divide up a Block into individual packets, each of which contained a checksum header and could contain one or more chunks of the Block. Because Hadoop uses TCP for data transport, we were unsure why at the bottom most layer of data transfer Hadoop would be performing its own packeting and computing checksums over the data it was sending. This is redundant, since TCP guarantees the integrity of data being transmitted, and Hadoop has already verified the checksum of the entire Block when reading it. We speculate this may be legacy code that has been carried through from a time when perhaps UDP was an option for data transfer. We removed this redundant layer of code from Hadoop, and instead rely on NDN and our `Socket`s to ensure data packeting integrity during transport.

3.3.3 Transport layer and data integrity. At the moment, NDN only provides basic packet-level transport of data. This can be fine for basic use, but to achieve full utilization of the network's bandwidth a transport layer is required. Adding a transport layer brings many benefits: multiple concurrent Interests can be in flight at once; Data packets being returned can arrive out of order and be properly ordered then assembled before being returned to the application; and a transport layer can help hide some of the low-level details that would otherwise be exposed to the user. One of the major challenges when multiple Interests are in flight concurrently is how to handle network congestion to ensure the best possible performance for the application. NDN is still under active development and a solution to network congestion under different network conditions is proposed in [13]. Once it is implemented in NFD our transport layer will be able to take full advantage of it. However, at the moment we have created a basic transport layer that supports up to 100 concurrent Interests for a given Block transfer. This number was configured to adapt to our cluster's environment and chosen benchmarks as we attempted to strike a balance between throughput and the possibility of seeing network congestion.

Hadoop takes great care to ensure that data is not corrupted, either during transport or when at rest on disk. TCP is used to provide this guarantee, in addition to a packeting system as previously described. Our implementation maintains the same guarantee of data integrity. We leverage NDN data integrity mechanisms that use cryptographic hashes to verify data at any point in the network. If for some reason there is a mismatch, the data can be requested again to ensure that the requesting host receives the correct data. Writing data to disk is unchanged under NDN: a cumulative checksum is computed for the entire Block when it is written and verified when reading the Block in the future.

4 METHODOLOGY

We constructed two clusters for our investigation: a small local testbed to run various benchmarks comparing the performance of regular Hadoop versus Hadoop over NDN, and a 128 node cluster utilizing

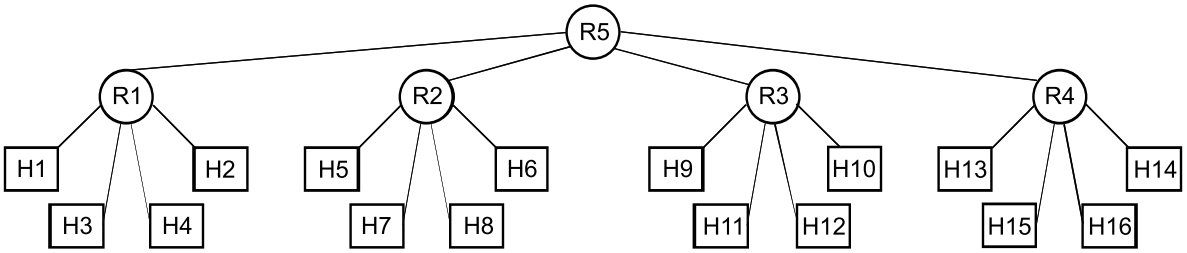


Fig. 5. Testbed topology of 16 VMs (H1 – H16), with network routers (R1 – R5) providing connectivity.

Benchmark	**Dataset size [GB]**		**Runtime [min]**	
	16 nodes	**128 nodes**	**16 nodes**	**128 nodes**
TestDFSIO	10	1000	3.5	273.6
TeraSort	10	1000	7.62	106.2
WordCount	3.3	60	17.85	342
PageRank	0.38	1.21	3.67	60
MahoutBayes	0.19	35	10.28	184.2
K-Means	0.59	66	6.85	200.4

Table 1. Benchmark details for 16 and 128 node clusters.

Amazon's EC2 cloud service so we could explore the possibility of caching in a more realistic setup. The local cluster consisted of a total of 4 physical machines / 16 virtual nodes that were arranged in a basic two-level tree topology, as shown in Figure 5.

Each physical server in our lab hosted 4 identical Xen guests, each of which ran Ubuntu 14.04 LTS with two 3.5Ghz vCPUs, 3.5GB RAM and a dedicated Gigabit NIC via PCI pass-through. Physical backing for each Xen guest's storage was a RAID-0 SATA disk array. Hadoop version 2.4.0 and NFD version 0.4.1 were installed on each Xen guest; only basic configuration changes were made, such as specifying the IP address of the NameNode and other changes required to successfully start up a Hadoop cluster. The jndn client library version 0.13 was used to interface Hadoop with the NDN protocol. An isolated network with dedicated Gigabit Ethernet links between each Xen guest and Linux machines acting as network routers with bridged network connections was configured; no other external traffic was present. Each Linux router machine had five NICs bridged at the MAC layer and ran Ubuntu 16.04 LTS with NFD version 0.4.1. The default routing policy for Interests was set to be "broadcast" and the virtual NDN network configuration followed the physical network topology. The EC2 cluster ran Hadoop version 2.7.0 and similar to the smaller cluster only basic configuration for cluster configuration was modified.

There are several widely-used benchmark suites for evaluating the performance of a Hadoop Cluster. One of the most well-known benchmark is Intel's HiBench [6]. Hadoop also comes with several example programs can be used to compare performance of a cluster. Table 1 shows benchmarks we have selected to evaluate the performance of our system as well as the dataset size and runtime when executed on both the 16 node local and 128 node EC2 clusters. These benchmarks were picked because of their wide spread use in existing literature and their ability to generate realistic workloads that model real life use of a Hadoop

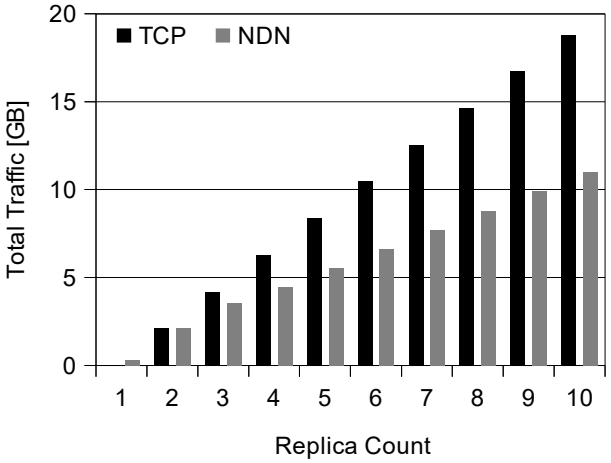


Fig. 6. Total network traffic seen with varying amounts of HDFS replication when writing 1GB on the 16-node cluster.

cluster. TestDFSIO and TeraSort ship with Hadoop, while the last four are part of HiBench. *TestDFSIO* is a distributed read/write benchmark that tests the HDFS. *TeraSort* is an I/O-bound benchmark that generates large random datasets and then sorts them. *WordCount* is a CPU-bound benchmark that counts the number of words in a dataset. *PageRank* is an implementation of the PageRank algorithm whose input is generated according to a Zipfian distribution. *MahoutBayes* is a benchmark that uses the Naïve Bayesian algorithm for simple distributed machine learning over a generated data set. Finally, *K-Means* is an implementation of the K-means clustering algorithm.

In addition to the above benchmarks, we have used raw HDFS commands to gather basic performance measurements on our in-lab cluster using Hadoop's `fs` command to write and read files from HDFS. A basic direct comparison of the performance of raw TCP sockets and NDN data transfer was also performed.

5 RESULTS

This section presents the results of the Hadoop-NDN implementation and compares the differences between the use of TCP versus NDN for network communication.

5.1 Benefits of NDN when replicating data

We begin our analysis by looking at one of the most common cases in Hadoop: writing new data to the HDFS and the subsequent replication among nodes as Hadoop ensures sufficient redundancy in case of node failure. Figure 6 shows the impact on overall network traffic as the number of replicas configured for HDFS increases while writing a 1GB file on the 16 node cluster. The default number of replicas in Hadoop is 3, but this can be configured as desired. A replica count of 1 essentially disables the redundancy provided by HDFS, so in practice it will never be used on a real cluster. For both TCP and NDN with the number of replicas set to 1, we see very little network traffic, as data is stored on the local node. NDN does have slightly higher amount of traffic than TCP – this is additional broadcast traffic seen in the current NDN implementation, as every node in the network will receive a copy of the broadcast, even if it cannot respond to the request.

Node	TCP [MB]	NDN [MB]	Change
H3	137.61	140.24	+1.9%
H5	268.21	139.36	-48%
H13	130.88	139.39	+6.5%
H12	0.02	2.81	+14,380%
TOTAL	536.72	421.81	-21.4%

Table 2. Broadcast traffic comparison while replicating a single 128MB Block.

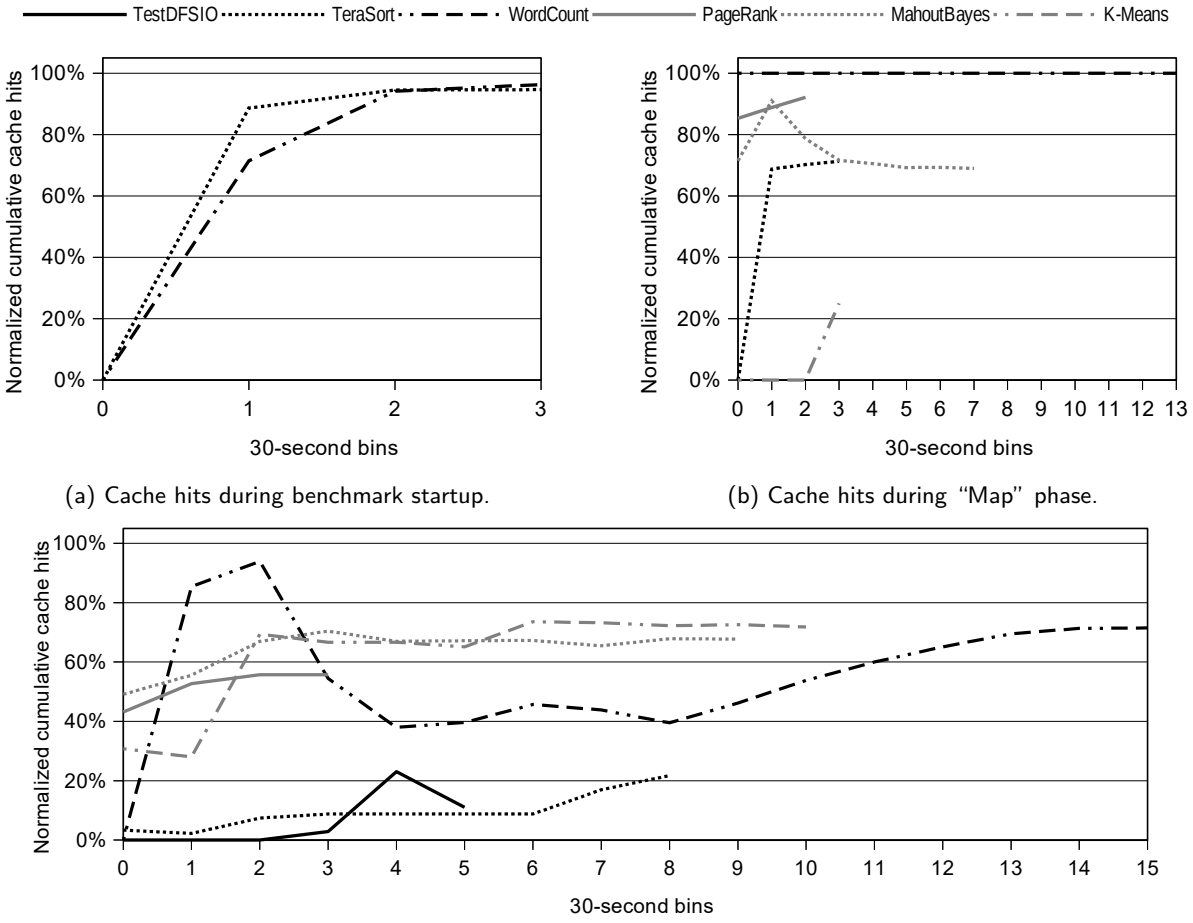
In a default block replication of 3, the additional traffic due NDN broadcasts is only 1.4%. NDN developers are currently working on a way to reduce the amount of erroneous broadcast traffic that needs to be processed by NFD [17] which will make NDN perform more like TCP/IP. While the broadcast traffic is higher, its overall impact amongst all data traversing the network is minimal. With 2 replicas, we see the total amount of traffic is approximately equal under both TCP and NDN, as sending only one copy of the Block to another DataNode does not offer any opportunities to aggregate or cache the data in the network. The default of 3-Block replication in Hadoop results in 16% less traffic in NDN than in TCP. As the number of replicas increases, so does the gap between NDN and TCP. With 10 replicas, a total of 18.8GB of data is seen compared to 10.9GB under NDN – a reduction of 42%!

Table 2 explains the benefit seen under NDN by showing a detailed accounting of the traffic sent and received from three active DataNodes as well as an idle one when writing a single 128MB Block to the Hadoop cluster. The three active nodes (H3, H5, and H13, shown in Figure 5) are the three nodes that Hadoop chose to use when establishing the pipeline write for the new Block . The idle node (H12) is simply a node that was not directly involved during the write of this specific Block and serves as a baseline for nominal network activity. Under normal Hadoop, nodes H3 and H5 must transmit the data over the pipeline, while H5 and H13 receive the data. (If Hadoop is configured to make N replicas of the data, then DataNodes 1 through $N-1$ will mirror that data during the write to DataNodes 2 through N.) However, as NDN can leverage content aggregation, we see that DataNode H5 sees half of the total traffic under NDN as compared to TCP since it does not have to both send and receive the Block data as part of the pipeline. The other two active nodes see slightly increased traffic, but not a significant amount. Overall, the amount of traffic seen between these nodes is reduced by 21.4%. The increase in broadcast traffic for the idle node (H12) does not introduce a significant amount of load on the network as compared to the overall data transferred.

5.2 Caching potential in NDN

NDN caching can offer benefits during any phase in addition to the data replication phase seen earlier. Figure 7 presents a caching potential in the 16-node cluster. Each trace is normalized to the total number of cache hits seen up to that point in time and is separated into three distinct phases: a startup phase, before the benchmark fully begins; the "Map" phase where data is transferred to the necessary nodes; and the "Reduce" phase where the computation is performed and a final result returned. The x-axis represents the execution time in 30-second bins, i.e. the time elapsed from 2 - 3 on the x-axis represents the 60-90th seconds of execution time. Since the benchmarks did not all take the same amount of time to run, some lines end sooner than others in each graph.

Only two benchmarks, TeraSort and WordCount, had a small amount of reads that could potentially be served from a cache during startup as seen in Figure 7a and Table 3. The other benchmarks did not have significant startup phases and data transfers as seen in Table 3. Even in case of TeraSort and



(a) Cache hits during benchmark startup.

(b) Cache hits during "Map" phase.

(c) Cache hits during "Reduce" phase.

Fig. 7. Normalized cumulative cache hit rates for each benchmark from the 16 node cluster.

WordCount, the startup phase was not significant as compared to the overall execution. Figure 7b shows the Map phase of each benchmark and the corresponding data activity is presented in Table 3. At one extreme, TestDFSIO has no reads in its Map phase, as its data is already present in each node. At the other extreme is WordCount, where virtually all reads can be served from the cache once the startup phase is completed as every node requests a complete copy of the data that will be needed when they count the number of words in their data. The remaining benchmarks show more variation in when and how many reads can be cached.

The Reduce phase has the most diverse activity as shown in Figure 7c. PageRank completes the most quickly, but has a good potential for benefiting from caching. TestDFSIO, which performs all its work in the reduce stage, does not have much opportunity for caching until near the end of its execution, as for most of its time TestDFSIO is reading unique Blocks. We will see that cumulatively, TestDFSIO is the least cacheable of the selected benchmarks, and provides a good example of workloads that mostly

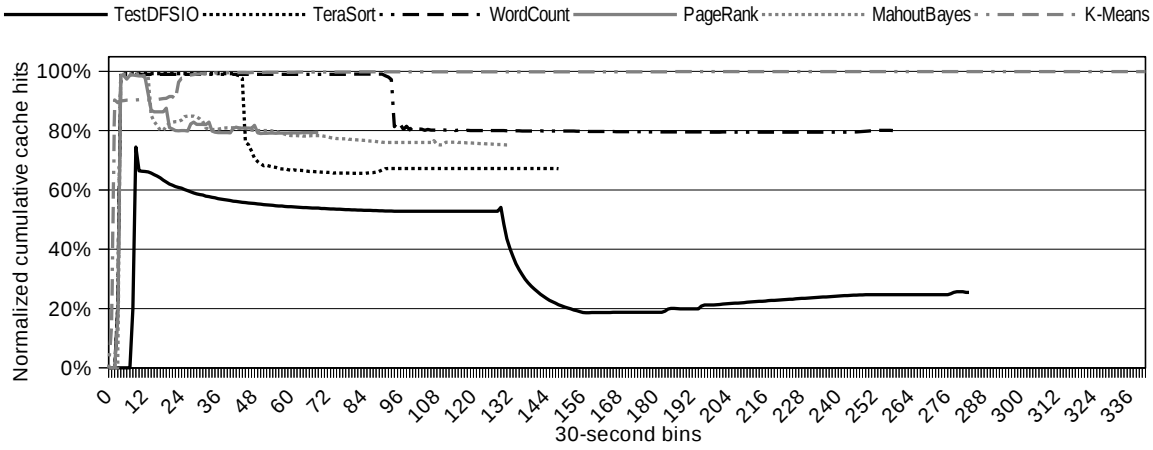


Fig. 8. Normalized cumulative cache hit rates for each benchmark from the 128 node cluster.

Benchmark	Setup [GB]	Map [GB]	Reduce [GB]	Total [GB]
TestDFSIO	0	0	9.98	9.98
TeraSort	0.01	8.71	5.41	14.13
WordCount	0.01	7.88	50.14	58.02
PageRank	0	0.20	0.71	0.91
MahoutBayes	0	1.33	5.46	6.79
K-Means	0	0.34	3.99	4.33

Table 3. Network data transfers in the 16-node cluster.

deal with unique data. WordCount shows a significant spike near the beginning where a large number of reads could be cached, followed by a period of lower cache potential as individual nodes perform their counting task, before once again reaching a period of higher caching as results are collected before the benchmark completes. MahoutBayes and K-Means remain bounded between 60% and 80% cache hits after the 30 second mark, as they are both distributed computations which must send results to their neighbors as the computation progresses. Finally, TeraSort is similar to TestDFSIO as its workload is largely self-contained and data communicated is mostly unique to each node.

Figure 8 presents the caching potential for the 128-node EC2 Hadoop cluster. Due to space limitations, we show an aggregated figure for all phases. The corresponding dataset statistics are shown in Table 1. The overall results shows a significant potential across all phases. Larger network size and datasets also show a larger potential for in-network caching that is offered by NDN. This behavior is most notably seen in K-means, where a larger dataset and communication pattern resulted in a 99% hit rate after about 14 minutes of execution. This hit rate is approximately equal to the ratio of reads to writes observed in the trace. Figure 9 presents a summary view of the potential for caching in each trace as executed on the 128 node cluster. All caches are infinite to show the maximum potential for caching and the unified cache shows the theoretical best upper bound.

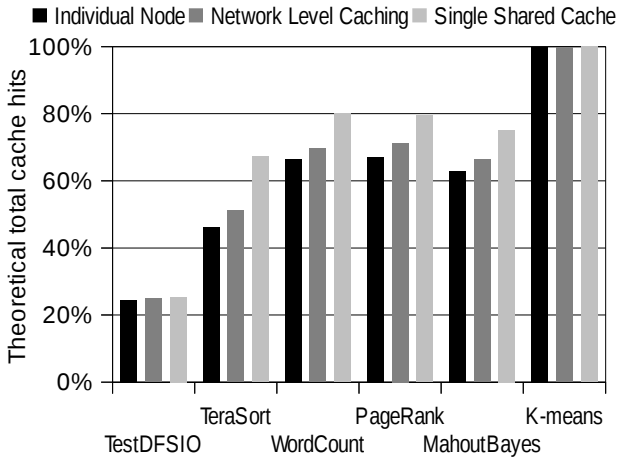


Fig. 9. Caching potential in benchmarks run on the 128 node cluster.

Individual node caching represents an infinite cache at the individual node and is independent of the network protocol. Hadoop nodes do have the ability to cache Blocks, if they have space and until told to discard the cached Block by the NameNode. Adding network level caching requires support from the network protocol, which NDN can provide. The additional gains shown in Figure 9 would depend on additional logic for caching placed in each router, replacement policy, and caching hierarchy. Utilizing NDN would make this process significantly easier than TCP. Single shared cache represents an upper bound on caching ability and corresponds to a single router with infinite cache for all nodes.

All benchmarks, except for TestDFSIO, show the potential for caching a significant number of reads, even if done only on individual nodes without the aid of a network-based cache. Adding the network cache improves cache hits by 23%, on average. For all benchmarks except WordCount, the difference between caching due to the network topology and a unified cache is similar; WordCount sees a larger difference due to its heavy amount of inter-node communication. The TestDFSIO benchmark is different from the others, since it is testing just reads from the HDFS, but does not repeat a read of a Block within the same benchmark test. Thus, the overall potential for caching reads is not very high in this particular benchmark. K-means again shows a very high cache hit rate, due to the very high level of repeated reads during the execution.

5.3 Socket-level comparison of TCP vs NDN

We implemented a basic transport layer over NDN to improve observed performance as we discussed in Section 3.3. Our code was deployed on our 16-node cluster in the lab where we had complete control over the setup. Figure 10 studies the optimal static window size for in-flight Interests. We sent various amounts of data, ranging from 8MB to 128MB, while varying the window size from 1 to 128 Interests. The time required dramatically drops as the window size is increased, but then quickly levels out and remains largely consistent after the window size approaches 100. Thus, when configuring Hadoop for testing, we selected a window size of 100. While this is an insufficient amount of traffic to saturate the network, we instead hit a throughput limit imposed by the speed of the CPU that is running NFD. This

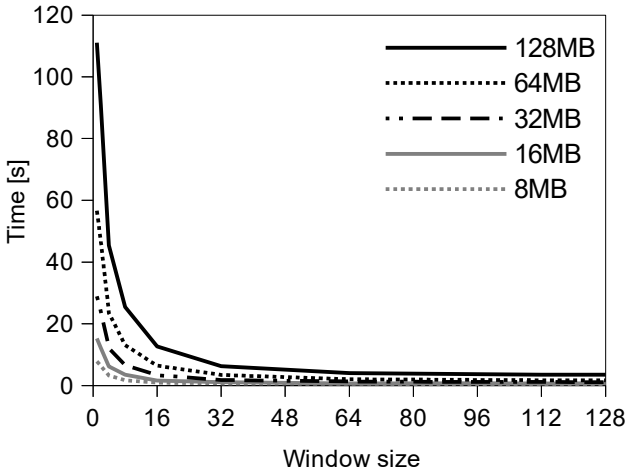


Fig. 10. Time to transmit data using NDN with varying transport layer window sizes.

occurs because NFD currently operates as a user space program as the current network interfaces only have hardware support for IP.

In contrast, TCP uses a dynamic window size for its transport window and has hardware optimizations. Because we quickly hit a CPU bottleneck when queuing around 100 Interests, we cannot fully explore the performance of NDN with a window size that begins to saturate the network's bandwidth. While this is currently an issue, as the NFD code base matures we expect the performance to increase as the code is further optimized, kernel support added, and hardware level optimizations become available.

After determining the best feasible window size for our transport layer implementation, we performed basic socket-level transfer tests as shown in Figure 11 to measure the average amount of time required to transfer buffers of different sizes while adjusting the NDN transport window size for in-flight Interests. In NDN's case, we set the transport window size to 1, 8, 16, 32 and 100 (our previously determined best feasible) concurrent Interests. We see that for each window size selection, the overall growth of NDN's transfer time is linear with the size of the buffer, as expected. For very small buffer sizes of up to a couple of MB, the growth is sub-linear, since there is a fixed startup time for setting up the NDN routes that was a significant amount of the total time required to send the data. With the best feasible window size, NDN remains within 7.4 - 21x the speed of TCP.

While a larger transport window significantly improves performance, NDN does not have the capability to efficiently handle congestion now, which can easily occur if the network is saturated with traffic, such as when Block replication is occurring. Additionally, for simplicity our transport layer implementation uses a fixed window size. A proper transport layer that can dynamically adjust its window size and handle congestion will result in performance that more closely tracks that of TCP.

Another significant impact on the overall network throughput is the total round trip time (RTT) experienced by each packet. We performed an experiment to measure the average amount of time it took a simple program to transmit and receive TCP and NDN packets through the host's network interface or NFD, respectively. We discovered that it took, on average, 0.03 milliseconds from the time a TCP packet was sent to the time it was received, while NDN took 21 milliseconds, a factor of 700. A significant amount

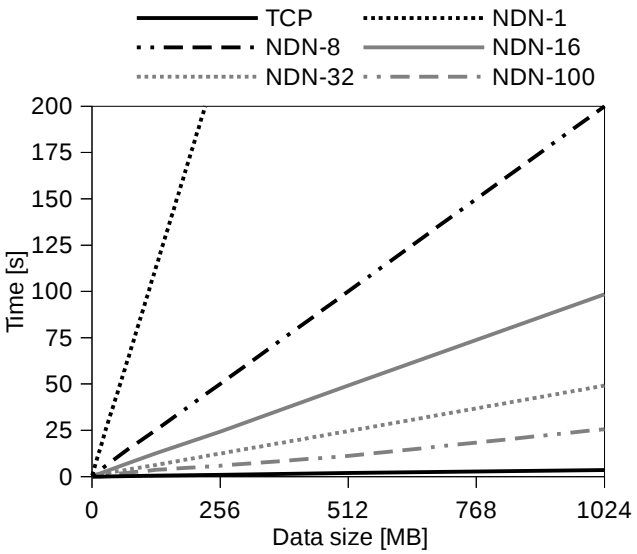


Fig. 11. Transfer time comparison of various NDN window sizes to native TCP.

of this time is due to processing time in NFD, which must be done in user space software. Advances in NDN design and hardware will result in significant performance gains.

5.4 HDFS-level comparison of TCP vs NDN

Figure 12 compares the average amount of time it took to write or read a given sized file in HDFS, using TCP or NDN, with two Block replication levels: 3 (default, "-3") and 6 ("-6"). We used Hadoop's existing HDFS utility to interact with the file system: `./bin/hadoop fs {-cat,-put} <file>`. When interacting with the HDFS, there is some amount of overhead as the DataNode initiating the operation must contact the NameNode and wait for the response before beginning the actual task as described in Section 3.3. This can become a significant amount of time for small amounts of data in NDN as it is slower than TCP now. As larger amount of data becomes dominating the transfer time, we are able to see a more realistic performance of the HDFS and observe that both TCP and NDN behave in a linear manner in terms of the amount of data transfers, as would be expected.

Writing 1GB of data with a normal replication of three via TCP took about 25 seconds, while NDN took 50 seconds. For small amounts of data, NDN is about 7 times slower than TCP. As the amount of data increases, the relative overhead of NDN decreases. Measured write speeds using TCP range from 7-41MB/sec and NDN speeds range from 1-21MB/sec. Reading is faster for larger files, achieving a maximum throughput of 66MB/sec for TCP and 47MB/sec for NDN. When using a replication level of six, writing takes longer under TCP, since there are more DataNodes involved in the replication process; NDN also takes slightly longer. Reads, however, are faster except for very small amounts of data, since higher replication levels mean there are more potential nodes that could send Blocks. Currently, NDN does have a higher performance overhead than TCP/IP. This is partly due to the design of NDN as well as the fact that the software is still under heavy development and not as optimized as the TCP/IP implementations.

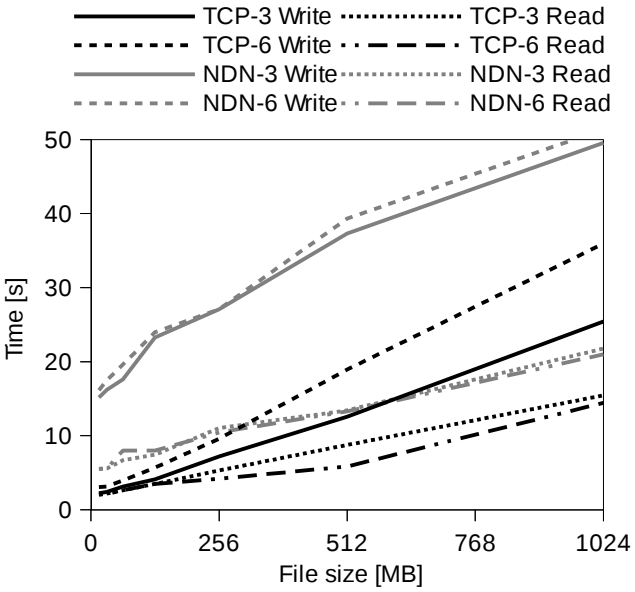


Fig. 12. Time to read and write files in HDFS.

The overhead can be measured in two metrics: the number of bytes transferred and the time required doing so. We have already considered the overall time, and will examine overhead in terms of bytes. In most networks today, a packet sent via TCP/IP has a minimal header size of 40 bytes, with a maximum packet size of 1,500 bytes, although jumbo frames can support up to 9,000 bytes. Thus, the minimal overhead of TCP/IP header per packet is between 2.7% and 0.4%. In contrast, a NDN packet header is variable in size, but is minimally 50 bytes with a maximum packet size of 8,800 bytes. Comparing with TCP's traditional maximum size of 1,500 bytes up to NDN's maximum size gives an overhead of between 3.3% and 0.6%. Subsequently, we can expect NDN to have a slightly higher overhead when transferring data. After a detailed measurement of transfer overheads with varying file sizes, we found that the total overhead when sending data with NDN was approximately 1.5% greater than when using TCP.

5.5 Code Reduction / Simplification

Taking advantage of NDN functionality allowed us to reduce the amount of code required in Hadoop to transmit data. This has made the code easier to understand as well as reduced the chance of bugs being introduced in the future. Most of the changes have been eliminating error handling that will no longer happen under NDN, such as failures during a pipeline write of a Block. As mentioned earlier, we were also able to eliminate the `Packet` class, since it no longer served any purpose when using NDN in our implementation.

Table 4 lists some of the most important classes which were modified to make Hadoop operate over NDN, along with the change in lines of code and whether or not that class was specific to Hadoop or more general in nature. The `ipc.Server` class forms the base for Hadoop's RPC communication and required slight modification to use the NDN `Socket` wrappers that had been developed. The `DataXceiver` and `DFS` classes deal specifically with the transfer of HDFS Blocks over the network, and another 28 classes

Class	Classification	± Lines of Code	Description
ipc.Server	Hadoop-specific	+12	RPC server
DataXceiver		-100	Core class for data transfer
DFSInputStream		-4	Core input class
DFSOutputStream		-126	Core output class
DFSOutputStream.Packet		-150	Removed
All other classes		-482	28 other classes modified
NDNSocket	Glue code	+476	Generic wrapper class
NDNServerSocket		+137	Generic wrapper class
NDNBufferConsumer	NDN transport layer	+159	Base NDN data receiver
NDNBufferProducer		+186	Base NDN data sender

Table 4. Hadoop classes with major changes to support NDN.

relating to the HDFS code were modified. The final four classes listed in the table are the most significant parts of our NDN "glue" code to interface Hadoop with NDN.

Overall, we were able to eliminate over 800 hundred lines of code and many edge cases by switching to NDN rather than IP. As this was just an initial attempt where we bolted on NDN, we tried not to make too many significant changes. In our future work, we will attempt an implementation that is designed to use NDN from the ground up, rather than relying on wrapper classes, reducing the amount of code needed to interface to NDN.

6 RELATED WORK

Named Data Networking (NDN) is a general Internet architecture that can be applied to different network environments. Most existing work focuses on the wide-area Internet. There has been some work done on applying or adapting NDN to data center networks. IC-DCN [9] uses an information-centric data plane to get the benefits of caching, while using a centralized control plane to coordinate multipath routing and cache-aware routing. CCDN [23] works as a shim layer between applications and HDFS to provide a caching benefit from the storage at the end hosts. These works focus only on the network aspect of the system, but do not have empirical performance evaluation of applications on top of the network system.

The effort most relevant to this paper was attempted in 2011 at Stetson University by Sherwood [15, 16] using CCNx [12], an earlier project from which NDN was forked. The initial intent was to rewrite the networking-related components of Hadoop with CCN-based counterparts. This was abandoned in favor of creating an adapter that would appear as a traditional socket interface to Hadoop, while utilizing CCN for data transport, in a manner similar to our approach. However, Sherwood was unable to successfully get Hadoop working over a content-centric network.

A significant amount of research has been done looking to improve the performance of Hadoop, and distributed computation in data centers in general. Shafer et al in [14] performed an analysis of Hadoop clusters trying to determine where performance bottlenecks exist, and what could be done to solve them by taking advantage of the underlying system. Various attempts have been made to add more and faster caches to servers in a transparent manner, such as Mercury [4]. While these can be useful, they suffer from being generic and unable to leverage their full potential without knowing details about the applications currently running.

Software-Defined Networking (SDN) [10] is an emerging architecture that decouples the network's control plane from its forwarding hardware. It enables the control plane to be programmable and offers

fine-grained network traffic control. For example, SDN has been deployed in Google's network to provide dynamic and accurate traffic engineering across different data centers [8]. While SDN empowers the *control plane* of any type of networks, NDN provides an upgrade of the *data plane* from address-based IP architecture to data-centric. Both of them can improve the performance of networks and applications, and they can complement each other to achieve maybe greater benefits.

7 CONCLUSION AND FUTURE WORK

In this paper, we showed through our first-of-its-kind work that it is possible to run Hadoop, a large, complex piece of software over a NDN network. While currently not as fast as TCP/IP, using NDN in a data center shows great potential to assist in better utilization of the network and a reduction in the aggregate amount of traffic seen. This can help provide better performance for compute clusters while at the same time requiring less direct administration in terms of network topology or compute job placement.

From experiments we found that the total observed network traffic in a Hadoop cluster between nodes can be reduced by over 21% and the overall aggregate network traffic with default replication settings can be reduced by 16% by blurring the separation between network and application layers. A strong potential for additional caching during actual execution of jobs is also present. Leveraging the features of NDN allowed us to simplify parts of Hadoop's code and promises future benefits.

There are still several areas of improvement and exploration that we will investigate, including: caching optimizations to further reduce transmission of duplicate data; optimization of NDN performance to address the issues identified that are negatively impacting performance; more native use of NDN, rather than using `Socket` wrappers; and leveraging NDN to decentralize some of the cluster state to improve reliability.

8 ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1629009, 1551057, 1345142, 1064963.

REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[2] Arista. Networking in the Hadoop Cluster. https://www.arista.com/assets/data/pdf/TechBulletins/NetworkingInTheHadoopCluster.pdf, 2016.

[3] A. S. Bonifacio, A. Menolli, and F. Silva. Hadoop mapreduce configuration parameters and system performance: a systematic review. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2014.

[4] S. Byan, J. Lentini, A. Madan, and L. Pabon. Mercury: Host-side flash caching for the data center. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[5] Google. Protocol Buffers — Google Developers. https://developers.google.com/protocol-buffers/, 2016.

[6] Intel. HiBench: HiBench is a big data benchmark suite. https://github.com/intel-hadoop/HiBench, 2016.

[7] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[9] B. J. Ko, V. Pappas, R. Raghavendra, Y. Song, R. B. Dilmaghani, K.-w. Lee, and D. Verma. An information-centric architecture for data center networks. In *Proceedings of the second edition of the ICN workshop on Information-centric*

networking, pages 79–84. ACM, 2012.
[10] D. Kreutz, F. M. V. Ramos, P. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):63, 2015.
[11] P. Mehrotra, J. Djomehri, S. Heistand, R. Hood, H. Jin, A. Lazanoff, S. Saini, and R. Biswas. Performance evaluation of Amazon EC2 for NASA HPC applications. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, pages 41–50. ACM, 2012.
[12] Palo Alto Research Center. CCNx — PARC's implementation of content-centric networking. https://blogs.parc.com/ccnx/, 2016.
[13] K. Schneider, C. Yi, B. Zhang, and L. Zhang. A practical congestion control scheme for named data networking. In *3rd ACM Conference on Information-Centric Networking (ICN 2016)*. ACM, 2016.
[14] J. Shafer, S. Rixner, and A. L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133. IEEE, 2010.
[15] J. Sherwood. An implementation of content-centric networking sockets for use with hadoop. Stetson University, 12 2011. Senior research project.
[16] J. Sherwood. Proposal: An implementation of hadoop utilizing content-centric networking. Stetson University, 5 2011.
[17] J. Shi, T. Liang, H. Wu, B. Liu, and B. Zhang. Ndn-nic: Name-based filtering on network interface card. In *Proceedings of the 3nd International Conference on Information-Centric Networking*. ACM, 2016.
[18] The Apache Software Foundataion. Apache Hadoop. https://hadoop.apache.org/, 2016.
[19] The Apache Software Foundataion. Apache Hadoop - HDFS Architecture. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2016.
[20] The NDN Project. Named Data Networking (NDN) - A Future Internet Architecture. https://named-data.net/, 2016.
[21] The NDN Project. Tools and Applications - Named Data Networking (NDN). https://named-data.net/codebase/applications/, 2016.
[22] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang. Named data networking. *SIGCOMM Comput. Commun. Rev.*, 44(3):66–73, July 2014.
[23] M. Zhu, D. Li, F. Wang, A. Li, K. Ramakrishnan, Y. Liu, J. Wu, N. Zhu, and X. Liu. CCDN: Content-Centric Data Center Networks. 2016.

Received October 2016; revised March 2017; accepted June 2017