NETWORK OPTIMIZATIONS FOR DISTRIBUTED STORAGE NETWORKS

A Thesis

by

COREY CASEY MORRISON

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2016

Major Subject: Computer Engineering

ABSTRACT

Distributed file systems enable the reliable storage of exabytes of information on thousands of servers distributed throughout a network. These systems achieve reliability and performance by storing three or more copies of data in different locations across the network. The management of these copies of data is commonly handled by intermediate servers that track and coordinate the placement of data in the network. This introduces potential network bottlenecks, as multiple transfers to fast storage nodes can saturate the network links connecting intermediate servers to the storage. The advent of open Network Operating Systems presents an opportunity to alleviate this bottleneck, as it is now possible to treat network elements as intermediate nodes in this distributed file system and have them perform the task of replicating data across storage nodes.

In this thesis, we propose a new design paradigm for distributed file systems, driven by a new fundamental component of the system which runs on network elements such as switches or routers. We describe the component's architecture and how it can be integrated into existing distributed file systems to increase their performance. To measure this performance increase over current approaches, we emulate a distributed file system by creating a block-level storage array distributed across multiple iSCSI targets presented in a network. Furthermore we emulate more complicated redundancy schemes likely to be used in distributed file systems in the future to determine what effect this approach may have on those systems and what benefits it offers. We find that this new component offers a decrease in request latency proportional to the number of storage nodes involved in the request. We also find that the benefits of this approach are limited by the ability of switch hardware to process incoming data from the request, but that these limitations can be surmounted through the proposed design paradigm.

ACKNOWLEDGMENTS

I would like to thank Dave Cerrone for allowing me to pursue this thesis while working full-time for Hewlett Packard Enterprise, and for pointing me towards the Degree Assistance Program to help me get it funded.

I would also like to thank Professor Alex Sprintson for helping me get started and encouraging me to continue this research.

Finally I would like to thank my friends and family for being patient with me as I took the time to finish this work.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professors Alexander Sprintson and Narasimha Reddy of the Department of Electrical and Computer Engineering and Professor Guofei Gu of the Department of Computer Science.

All work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was supported by Hewlett Packard Enterprise. Hardware used in the experimentation section was also provided by Hewlett Packard Enterprise.

NOMENCLATURE

DFS	Distributed File System
GFS	Google File System
HDFS	Hadoop Distributed File System
FIO	Flexible Input/Output Tester
LIO	Linux iSCSI Target
MDADM	Multiple Disk Administration

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Distributed File Systems have allowed for the reliable storage of exabytes of data on hundreds or thousands of servers distributed throughout a network using inexpensive commodity hardware. These systems achieve reliability and performance by storing multiple identical copies of data in three or more locations throughout the network, with the management of these copies of data being handled by intermediate servers that track and coordinate the placement of this data. Individual files in these systems are expected to be large (in the MB to GB range) and should be available to hundreds of simultaneous users. Individual servers in the file system are divided into two classes: the "Data Nodes" which store the data, and the "Name Nodes" which track where the data is stored and service the read/write requests. A diagram of this architecture and the data flow between the Name Nodes and Data Nodes in the Hadoop distributed file system is given in Figure 1.1 [3, 1].

This design introduces bottlenecks for specific loads. Fast storage devices can easily saturate the network links connecting Name Node servers to the Data Node servers while servicing a single request [3]. Redundant write operations are often done simultaneously to all destination data nodes at once which saturates outbound links. If a Data Node fails (which happens frequently [4, 5]) it must be rebuilt using a spare node, which can consume the outbound link bandwidth of one or more other Data Nodes in the system. Most distributed file systems are not aware of the network load at any given time. As a result, large bandwidth operations such as node reconstruction and large data writes can be routed through bottlenecked Name Node uplinks and inter-switch links (ISLs) in the network. This can deny users service to data nodes that depend on those network links for connection to Name Nodes [4].

Current solutions to these problems involve increasing the network link bandwidth of the intermediate Name Nodes or using more complicated network protocols such as layer 2/3 multicast. However, increasing network bandwidth involves increasing the port count of hardware (which does not scale out well) or increasing the complexity of the network hardware (which violates the “inexpensive and commodity hardware” requirement of these systems). Multicasting is another option, however most network multicast protocols are unidirectional and not particularly reliable [6]. Multicast domains do not scale out to the thousands of simultaneous requests these systems usually require, and they offer no opportunities for more advanced redundancy schemes (such parity checking across storage nodes).

This thesis proposes a way to reduce these performance bottlenecks by offloading some actions of distributed file systems onto network devices themselves, such as routers and switches. Specifically, any action that involves the replication of data across many nodes in the network or the processing of data from many nodes into a single stream of data is offloaded to the network hardware. This better utilizes network hardware which can have large amounts of bandwidth available to it and which can leverage its position in the network topology to handle these requests.

The rest of the thesis is organized as follows. In section 2, we give additional details about the operation of existing distributed file systems and some background information needed for understanding of our experimental setup. Section 3 describes relevant previous research into optimization of distributed file systems. Section 4 describes the proposed solution in more theoretical detail. Section 5 describes an experimental system which was built to test some of the advantages of the proposed system, and section 6 describes the results from that experimental system.

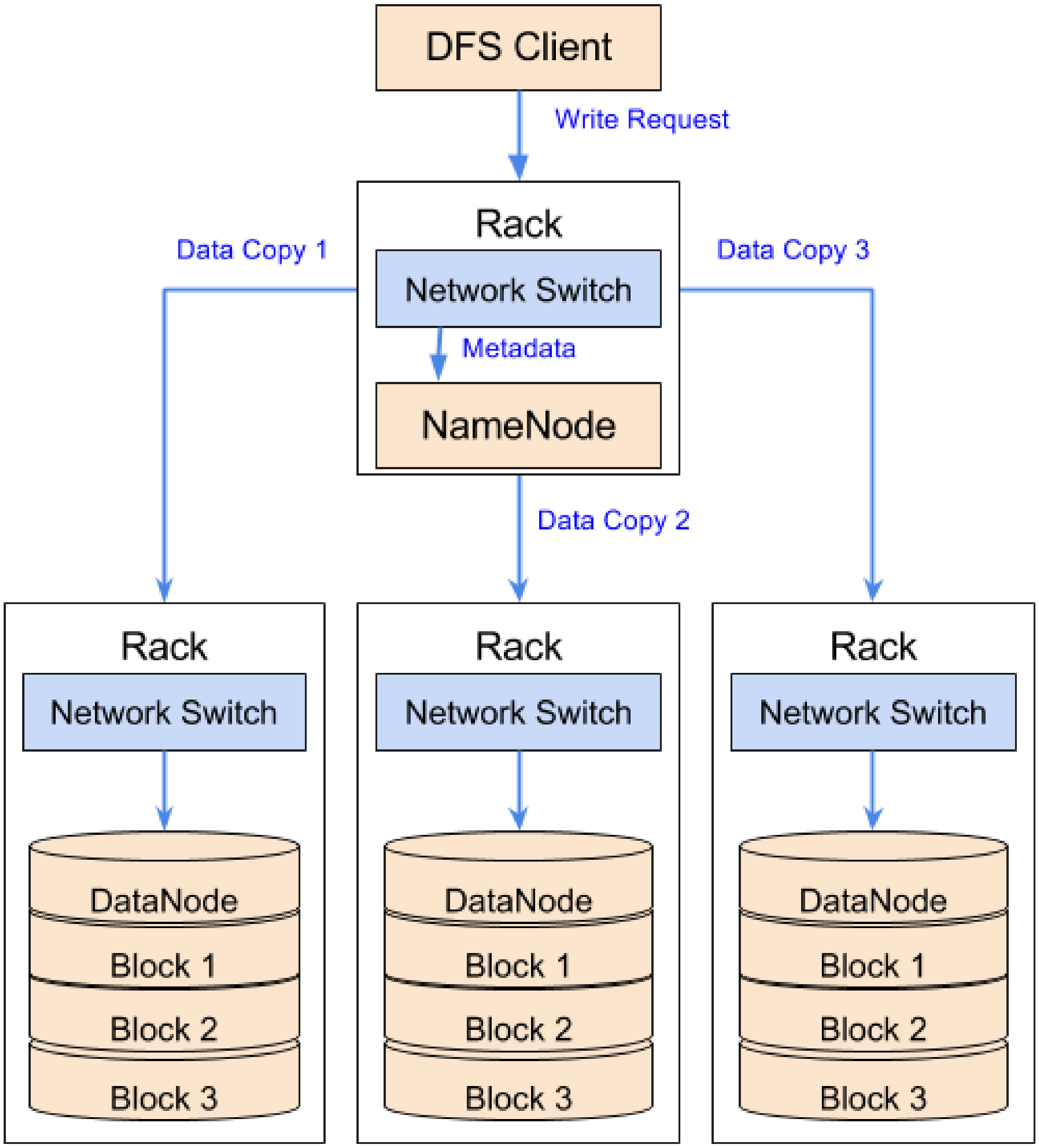

Figure 1.1: Common Distributed File System DataFlow[1]

2. BACKGROUND

2.1 Distributed Storage

Distributed File Systems were developed to meet the increasing storage needs of large companies attempting to serve petabytes of data to thousands of users simultaneously [3, 1]. There have been numerous implementations of distributed file systems in recent years to fill different storage requirements. However, all these systems attempt to provide reliable storage by distributing multiple copies of data across multiple storage locations in a network. They all attempt to make that data highly available by allowing users to access those nodes simultaneously through intermediate servers. The intermediate servers are responsible for servicing those requests while simultaneously keeping all storage nodes in sync. Well known examples include the Apache Hadoop File System (HDFS) [1], the Google File System (GFS) [3], the Gluster File System [7], and OpenStack's Ceph Storage system [8].

These systems all tend to operate by having daemons run on servers (or nodes) which fulfill specific tasks such as the tracking and managing of data or the storing of data itself. However, they differ in the granularity in which they do so. HDFS and GFS have two classes of nodes: those which store data (Data Nodes) and those which track where the data is stored and manage requests (Name Nodes). Ceph further divides the name nodes into nodes which monitor storage node state, nodes which store metadata information about the storage nodes, and nodes which act as gateways to expose the storage to clients requesting data. However, in Ceph all these nodes are fully distributed and all three name node daemons may run on the same physical server.

2.2 Network Switch Hardware

Switch hardware today typically consists of two parts: a switching application-specific integrated circuit (ASIC) and a control/management CPU or System on Chip (SoC). The switching ASIC portion is connected to the external switch interfaces and is responsible for forwarding packets from one interface to another. The control CPU is connected to the external management interfaces and has an internal connection to the switching ASIC. A diagram of this is given in figure 2.1. This control CPU is responsible for presenting a user interface to the switching ASIC in the form of a Network Operating System (NOS). This NOS usually runs on top of Linux and can be as simple as a Open VSwitch control plane communicating to a dataplane running on the switch ASIC. Recently the "Open Network Install Environment (ONIE)" initiative has produced bare-metal network hardware which can run any NOS that a user chooses to install [9].

2.3 Open vSwitch

Open vSwitch is an open-source, multilayer switch that is widely used in OpenStack deployments [10]. It is noted for being both production quality (with support for numerous standardized protocols and technologies such as OpenFlow and VLANs) and for being portable enough that it can be deployed on multiple different switch hardware platforms. This thesis makes use of the switch's robust OpenFlow support and its portability to multiple different hardware platforms. It also integrates well with the Linux kernel and allows internal taps to be made between the Linux kernel and the switch dataplane.

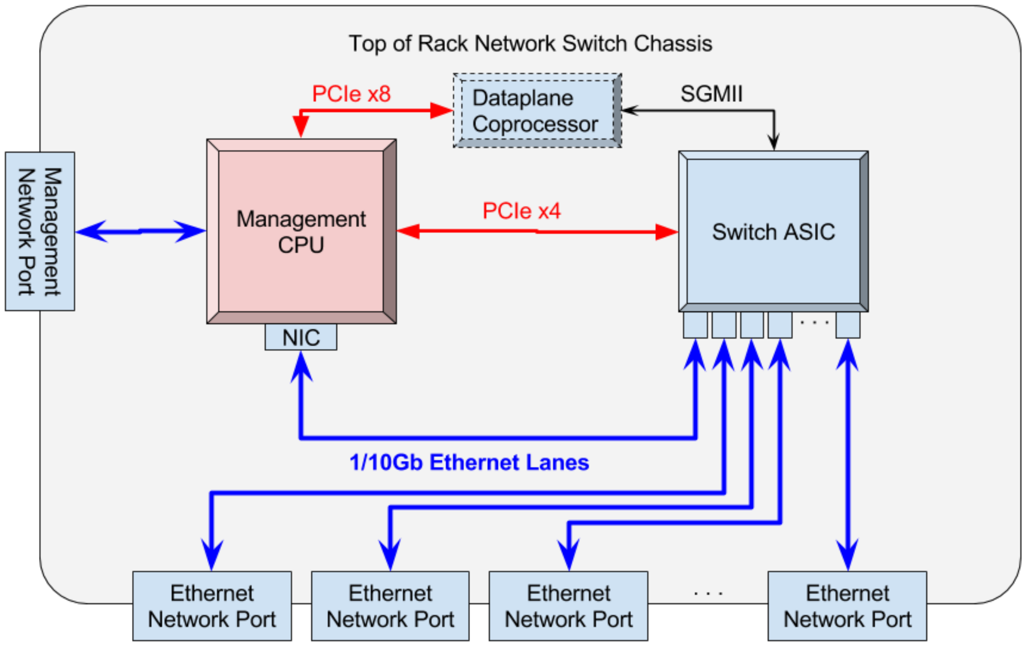

Figure 2.1: Common Network Switch Architecture [2]

3. RELATED WORK

Howard et. al. identified the problems of performance when multiple users attempted to access their early “Andrew” distributed file system simultaneously [11]. Their primary solution to this was to have local caches of any file to reduce the number of accesses back to the datastore. These conclusions were further studied by Muntz et. al [12]. However, all of these early distributed file systems were only designed to handle thousands of users and could not meet the demands of the growing internet age.

Google identified the problem with outgoing and incoming bandwidth from the “Name Node” equivalent in the Google File system and proposed a divide and conquer approach [3]. In their system the Name Node only pushes a copy of a new piece of data to a single Data Node. That Data Node then pushes the data to the next node and so on until the desired number of replicas is reached. Data Nodes report back to the Name Node once all data has been written. This strategy was later used by the Hadoop HDFS [1]. These two systems in particular address the question of physical locality of the different copies of data and they go to great lengths to provide mechanisms to ensure that in the event of rack or site failure users still have access to at least one copy of data. Determining locality is largely done through IP address assignments, but is not dynamic. The question of network bottlenecks between racks in the initial design paper is not addressed. Narayan et al. explore the use of OpenFlow to analyze and streamline some Hadoop data operations [13]. However, they do not propose reconstruction or mirroring of request at the network nodes.

Few other file systems go into much detail about their actual internal data flow as it is generally kept transparent from the client users. However, some performance measurements of different DFS systems suggest that they are either transmitting data serially from the intermediate Name Nodes or following schemes similar to GFS and HDFS [14, 15].

Some research has been done on the implementation of erasure codes in distributed storage systems to minimize network bandwidth or storage overhead in distributed file systems. Currently HDFS and Ceph both have options for Reed-Solomon based codes similar to the RAID-5 codes used in this study. Huang et al. demonstrated the benefits of parity based coding in Windows Azure Storage [16]. However, these implementations are limited in scope to specific storage classes or use cases and do not involve calculating erasure in the network devices as is proposed in this thesis.

Pamies-Juarez et al. successfully implemented a Minimum Storage Regenerating (MSR) code on top of both Ceph and HDFS [17]. This paper demonstrated a sizeable benefit in terms of storage overhead and network bandwidth over the Reed-Solomon based codes currently implemented. However, they conclude from their work that to get the theoretical expected repair performance requires careful implementation across all levels of the distributed file system. And again, no erasure coding is used in the network elements to achieve this benefit.

Dimakis et al. initially proposed using erasure coding for node regeneration and did an analysis on its expected effects [4]. This work was largely theoretical and dealt with more complicated erasure codes than the ones used in this thesis. It also did not have any implementation details on the network protocol to be used to achieve its goals. However, it is an indirect inspiration for this thesis.

4. PROPOSED DESIGN

Our proposed solution to the problems of limited Name Node bandwidth and excessive aggregate network usage is to create a new type of node, referred to as a *Switch Node*, that runs on switch hardware itself. This node would take over the duties of replication during write operations and of checking concurrency between different copies of data for reads. Name Nodes would be responsible for controlling the new Switch Nodes and configuring them to replicate data to and aggregate data from specific Data Nodes. Initially, this control would involve communicating to an agent running in user space on the switch OS which has direct input to the switch data plane. A diagram of this proposed architecture in a single-switch configuration is given in figure 4.1.

4.1 Switch Nodes

The large bandwidth available to a network switch platform and its connection to multiple nodes in a rack make it an ideal place to do replication tasks. Data coming in from a Name Node can be replicated by the switch hardware before being pushed out on switch interfaces. As redundancy schemes get more complicated, any operation which requires transmitting data from multiple nodes to decode the data needed by one node can be done by the switch hardware. In this case, the Switch Nodes are responsible for collecting information from the multiple Data Nodes, doing whatever erasure coding operation is possible or necessary on the data it has collected, and forwarding the result on to the next Node in the system.

Since this is critical data that is being replicated or modified, care must be taken to ensure that the data is still transferred reliably. Also, the switch's relatively weak CPU and memory resources mean that such tasks should be kept as efficient as possible if they run in the user space of the NOS.

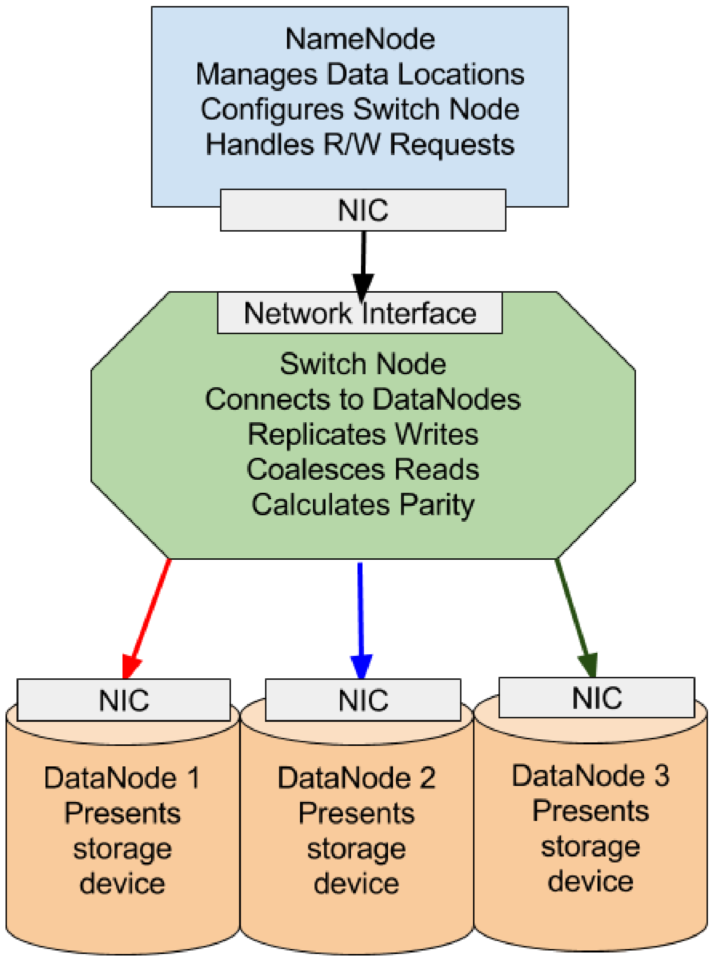

Figure 4.1: Proposed DFS Architecture

Many switch ASICs already do some replication tasks and have specialized hardware paths to make this efficient. They also already have internal hardware paths to do quick XOR operations as used in CRC checksums. These paths could be extended to support the rebuild of failed nodes when said nodes are using erasure codes to maintain redundancy.

4.2 Advantages

Our approach has a few advantages over the current approaches. The first is that it reduces the amount of aggregate bandwidth required to fulfill any individual request, similar to multicast. By passing information to the Switch Node and letting the Switch Node replicate the requests the distributed file system can take better advantage of the internal communication links between the network OS and the switch dataplane to replicate data. These operations are also simple enough that they could be done by the switch dataplane itself in the future using new transport layer protocols.

The second advantage is that this reduction in aggregate bandwidth also comes with a theoretical reduction in overall request latency. In the current systems if a single outbound write request saturates the network link it runs on then multiple simultaneous identical requests will divide this bandwidth and cause all requests to take longer to complete. By contrast, if write operations can be replicated simultaneously on all intended outbound interfaces of a switch, the latency of the request is reduced to the time it takes to write a request to the slowest storage node in the request instead of the combined latency of all storage nodes in the request.

Finally, DFS systems are beginning to use parity checking to achieve better storage density among their redundant copies (similar to the evolution from RAID-0 and 1 to RAID-5) [16]. In our system, the Switch Node would also take over the calculation of parity data. This would greatly increase the bandwidth efficiency of these parity calculations and data reconstruction in the event of data node failure and recovery.

4.3 Switch Node Orchestration

Determining which switch node is the ideal place to do any particular operation is largely dependent on the operation in question and the architecture of the network that it is running upon. The number of source and destination nodes and their connections to switches in the system should be taken into account for any individual request.

In this system, replication should be done as close to the leaf nodes as possible to reduce aggregate bandwidth. For a write operation replicated to multiple nodes, a minimum spanning tree should be created with the source of the data as the head node and every destination for the data is a leaf node. The tree's head and leaves may only be attached to switch nodes, and there must be one or more switch nodes in the tree. Any switch node that has more than one connection to a lower level of the tree will be configured to do replication (assuming that is possible). Read cases in existing replication systems usually only involve a single storage node and the requester and thus there is no need to perform optimization.

Data reconstruction or parity calculation, on the other hand, should be done as soon as a node has access to all required data. The target node can be either the requesting client or a storage node that is rebuilding or storing parity data. To simplify the operation, the calculation should be done in as few locations in the network as possible. Ideally, parity calculation would be done by a single network node. For a parity read case a minimum spanning tree should be created with the requester at the tree's head and all involved storage nodes as leaves. The scheme used for the parity calculation will determine whether recalculation can be done at any switch node (XOR parity) or just at a switch node that has access to all data (more complicated parity schemes).

Control of the replication and parity calculations should be done by Name Node or another agent that has an overview of the switch topology of the system. That way it can choose the ideal places to do replication and parity calculation operations. Thus the Name Node (or other entity controlling the Switch Nodes) must have knowledge of both the network architecture and the storage node locations in the network in order to be effective.

4.3.1 Storage Tree Algorithm

To generate the various replication paths through the network systematically, an algorithm is necessary to generate the trees by which data will be distributed to all the storage nodes. These "replication trees" must have one Name Node N to act as the root node. The replication tree will also have a number of Data Nodes, $D_1, D_2, \ldots, D_k$, at the leaves of the tree. The number of Data Nodes in the tree, k, will depend on the copy requirement of the distributed file system implementing it, but there must be at least two. Finally, the replication tree will have a number of Switch Nodes $S_1, S_2, \ldots, S_n$ which may or may not be in the resulting tree. The number of Switch Nodes can be as few as one or as many as are in the entire Distributed File System's network, n. There will also be a number of links between nodes, $C_1, C_2, \ldots, C_l$, which may or may not be under load at the time of the write request. Data and Name Nodes may ONLY connect to Switch Nodes, but Switch Nodes may connect to any node. In addition to these constraints on the number and connections of Nodes, there may be upper limits on the amount of connections that a switch can handle due to bandwidth constraints or processing power.

Given the original Name Node N, necessary Data Nodes $D_1, \ldots, D_k$, all Switch Nodes in the network $S_1, \ldots, S_n$, all links in the network $C_1, \ldots, C_l$, we propose a general algorithm to generate a desired replication tree T which can be used to replicate data from a Name Node to two or more Data Nodes. This general algorithm examines the state of the network when the request is made and determines which switch nodes in $S_1, \ldots, S_n$ are connected to the Name Node and the necessary Data Nodes. Let these nodes be considered the set $\mathcal{S}_r$. All S in $\mathcal{S}_r$ are required to be in the resulting replication tree T.

The algorithm then uses the Prim's algorithm to build a minimum spanning tree that connects nodes in $\mathcal{S}_r$. Specifically, the algorithm initializes a tree T that initially includes a node in $\mathcal{S}_r$ connected to N. The algorithm then adds a new Switch Node S_j that has the least-utilized link between itself and a node in the current tree T. The algorithm proceeds in a similar manner until all nodes in $\mathcal{S}_r$ are connected by T. Additionally, the algorithm keeps track of the number of inbound or outbound links any given switch node can process. If the new Node S_j is to be added via a link to an existing S in T which is over-utilized, that link is moved to a secondary list and a new S_j is found. If T cannot be completed without over-utilizing an S, links are used from the secondary list until a complete tree T is found. Once the tree is completed, all leaf nodes which are not in the set $\mathcal{S}_r$ are removed from T. The pseudo-code of this algorithm, referred to as Algorithm 1 is provided below.

Once the tree T has been generated, one can simply traverse the tree from the head Name Node and set up replication for any switch node which has more than one child. Flow table rules for intermediate Name Nodes which are not doing replication can also be added if the network switch is controllable via OpenFlow. Figure 4.2 illustrates how one of the replication trees may look in a simple 4-switch mesh network. All green links are used by the tree to replicate the write request from the client to Block A to two data nodes which are storing Block A. The switch in the upper-left quadrant of the mesh network is responsible for replicating the copy and sending it to both the locally attached first data node and the outbound link so it can be routed to the second data node.

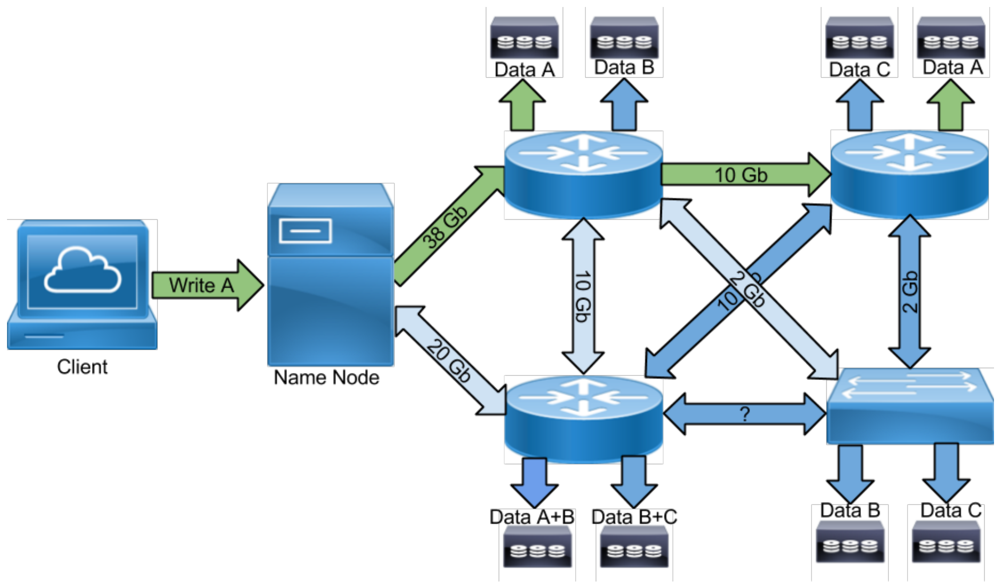

Figure 4.2: A Replication Tree Configured to Replicate Data Block A

Note that there is a third data node in figure 4.2 which stores a copy of data Block A. This third node is storing the result of and bitwise XOR operation on Data Block A and Data Block B. This data node allows the system to regenerate either Data Block A or Data Block B as necessary with additional bitwise XOR operation. Since there are three copies of the Data Blocks A and B across all Data Nodes in the network, it also allows the Name Node to report that it has three copies of Data Blocks A and B in the system. In existing systems data from both Block A and Block B must be transferred to the Data Node through its single switch link which will delay node updates.

However, the storage tree algorithm can also be used to generate trees for XOR parity node regeneration. Instead of assigning the head of the tree to the Name Node, assign it to the node which will hold the regenerated data. For the data nodes, choose one source (preferably the least utilized) for each piece of data used in the parity calculation. After the tree is generated, traverse the tree from the bottom up to determine what operations need to be done on any given data node (or to combine different data nodes) before it gets to the node under reconstruction. Care should be taken to ensure that no data corruption occurs as a result of these operations and that the resultant data from any operation creates a useable symbol for an erasure code. And example of the XOR parity recalculation tree for Block A and Block B is given in figure 4.3.

Algorithm 1 Storage Tree Algorithm

```
for all $S$ in $S_1 \dots S_n$ do
    Set $S.numconnections = 0$
    if $S$ is connected to $N$ then
        Set $S.priority = 0$ AND $S.required = true$ AND $S.parent = null$
    else if $S$ is connected to any $D$ in $D_1 \dots D_k$ then
        Set $S.priority = \infty$ AND $S.required = true$
    else
        Set $S.priority = \infty$ AND $S.required = false$
    end if
end for
Add $S_1 \dots S_n$ to a Queue $Q1$ sorted by $S.priority$
Create an empty Queue $Q2$
while $Q1$ is not empty OR All $S.required$ are not in $T$ do
    Pop minimum $S$ from $Q1$
    Dequeue minimum $parent$ from $S.parentpriorityqueue$
    if $parent.numconnections < parent.maxconnections$ then
        Add $S$ to $RT$
        Add $C$ connecting $S$ to $S.parent$
        for all $C$ connected to $S$ do
            Given $S2$ which is the other Switch Node connected to $C$...
            $S2.parentpriorityqueue.push(S)$
            $S2.priority =$ current link utilization of $C$
        end for
    else
        If $S$ doesn't exist in $Q2$ then add it.
        Find $S$ in $Q2$ and add $parent$ to $S.parentqueue$.
        For $S$ in $Q1$, set $S.priorty =$ current link utilization of $C$ connected to next
        $S.parentpriorityqueue$
        Enqueue $S$ in $Q1$
    end if
end while
while All $S.required$ are not in $T$ do
    Pop minimum $S$ from $Q2$
    Dequeue minimum $parent$ from $S.parentqueue$
    Add $S$ to $RT$
    Add $C$ connecting $S$ to $S.parent$
    for all $C$ connected to $S$ do
        Given $S2$ which is the other Switch Node connected to $C$...
        $S2.parentpriorityqueue.push(S)$
        $S2.priority =$ current link utilization of $C$
    end for
end while
```

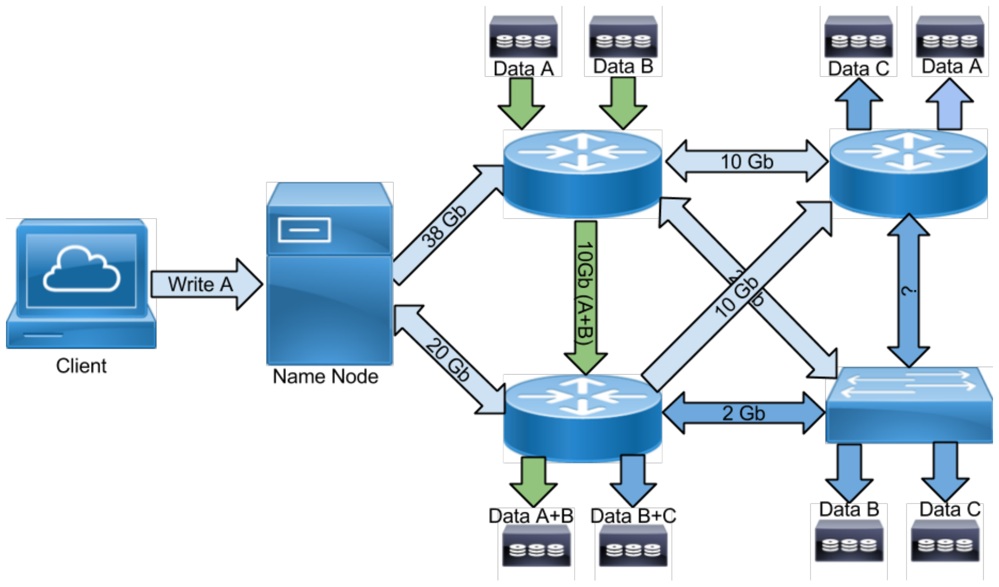

Figure 4.3: A Parity Calculation Tree Configured to Reconstruct Data Block A+B

5. EMULATION

In order to demonstrate the advantages of this system we built a small scale version of a distributed file system that allowed for more granular control of the flow of data through the network. This was done by using common open source utilities to emulate the behavior of different pieces of the distributed file system. This allowed us to better measure and track the flow of data throughout the DFS for the purposes of this paper. However, the techniques described in the "Proposed Design" section should be applicable to any DFS which uses intermediate nodes to track storage metadata.

5.1 Experimental System Design

In our proposed system there are three types of nodes: Data Nodes, Name Nodes, and Switch Nodes. To emulate them, these can be run either as VMs in a hypervisor cluster or as actual systems running on a network. Each node emulates the role and behavior of a server on a network in a common Distributed File System deployment. They also provide enough control that the behavior of the distributed file system can be changed easily.

5.1.1 Data Nodes

Each Data Node is a single server with a local storage device that it is presenting on the network. It has a single network adapter attached to it and runs an open-source iSCSI target implementation called LIO [18]. This target can use either the system's RAM to emulate a flash SSD storage device (otherwise known as a RAMDisk) or a locally attached hard drive to represent the slower devices common in large DFS setups today. The system has enough RAM to present a meaningful target (when emulating a flash SSD) and enough CPU resources to ensure that the only bottleneck that can be introduced by the system is due to the drive itself. This implementation was chosen because existing DFS systems are known to connect to iSCSI target devices [14].

5.1.2 Name Nodes

This node handles the end-user or mid-level storage system actions. It has one or more network adapters and runs the open source open-iSCSI initiator software [19]. It also runs the open source FIO [20] tester application which is responsible for all tests and simulated workloads in the environment. It is allocated enough CPU and RAM in all tests to ensure that there is no possibility of the system resources being a bottleneck in the test. However, the network link is allowed to be a bottleneck in order to demonstrate the advantages of the proposed solution.

In control tests this node will also run the software RAID manager known as “Multiple Disk Administrator” or “MDADM” [21]. This software handles data replication, data striping, parity calculation, and rebuilding in the event of node failure. It was chosen first and foremost because it is a known enterprise-level that supports RAID 5 and 6.

5.1.3 Switch Nodes

This node is either a software-based network switch or an ONIE compatible switch. The switch has a limited amount of CPU and Memory resources to emulate the actual limitations of switch hardware available today and to better characterize the additional hardware requirements that will be necessary to support this technology. There are also one or more internal network taps that act as TCP endpoints within the switch. An LIO target for upstream devices (either the client or upstream switches) will be presented on one of these tap interfaces, and the switch will connect to downstream storage nodes via these internal tap interfaces.

When running as a VM this node will have numerous NIC adapters attached to it which act as the external facing network ports of a switch. These NIC adapters are attached to the NIC adapters of storage nodes, the client VM, and other switch VMs through dedicated bridges on the hypervisor. Each hypervisor bridge has only two connections, one to a single NIC on the switch VM and the other to another node in the system and thus acts as a simple wire in the system. In this way, the client VM can only get data from the storage VMs via their connections to and the operations done by the switch VMs. This also gives us the ability to construct multiple network topologies by changing the way the Client, Storage, and Switch VMs are all connected together.

5.2 Node Connections

As mentioned, all Data Nodes are configured to present a fixed size RAMDisk as an iSCSI target. The Storage Orchestrator will send commands to the switch VMs which will in turn use their iSCSI initiator instances to these presented disks for all locally attached Storage VMs. The Storage Orchestrator will then direct the mdadm instance on the switch VMs to combine the iSCSI drives into new disks (using pre-determined RAID schemes). Lastly the orchestrator will direct switch VMs to present their new disks as iSCSI targets on their internal interfaces. Upstream devices will then connect to these new disks and the process will start again at the next highest level. This process repeats until you get to a single presented disk that Name Nodes or clients can connect to in order to write to a set of disks on the network.

5.3 Benchmarks

In order to determine the performance of the proposed system a number of benchmarks were measured between a control configuration, where all replication and rebuilding is done by the Name Node, and the proposed system's configuration. This benchmarks will include the amount of time taken to write increasing amounts data to the system, the time taken to read data from the system, and the time it takes to rebuild a failed node in the system. The read-write benchmarks are largely based on the work by Depardon et al. [14] while the rebuild benchmark is a simple measure of the time it takes the system to rebuild a failed node.

6. RESULTS

6.1 Experiment 1

The first experiment is done in a highly virtualized environment in order to prove the feasibility of the system and determine its benefits to replication duties under ideal conditions. A single Switch Node running Open VSwitch inside a VM is connected to between 3 and 5 Data Nodes which are also running as VMs. The available bandwidth between the Switch Node and the Data Nodes is kept quite high, but all traffic from the Data Nodes is forced to run though the Switch Node. The NameNode runs on a separate server connected to the switch VM via a single 1Gb link in order to present a possibly bottlenecked Name Node link. The DFS configuration is shown in Figure 6.1.

For this experiment, only basic replication between the Data Nodes is done. For this experiment, varying sizes of write operations are given to the system and the time it takes them to complete is measured. These write operations were performed with a single-thread, sequentially in 512k chunks with 16 IO operations allowed to be in-flight The number of Data Nodes is then increased to measure the effect that additional nodes has on the two systems. The results of these tests are in tables 6.1, 6.2, and 6.3. Each table lists the time it takes to service the given size request when replication is being done at the Switch as opposed to when it is being done by the NameNode.

The results show that when replication is done at the switch the amount of time it takes to service any size request is severely reduced. This is because each Data Node individually was capable of pushing line-rate traffic to the Name Node. So in this system when replication is done by the Name Node the bandwidth available for replication is divided equally to the number of Name Nodes and the request time increases accordingly. The proposed system has no such bottleneck.

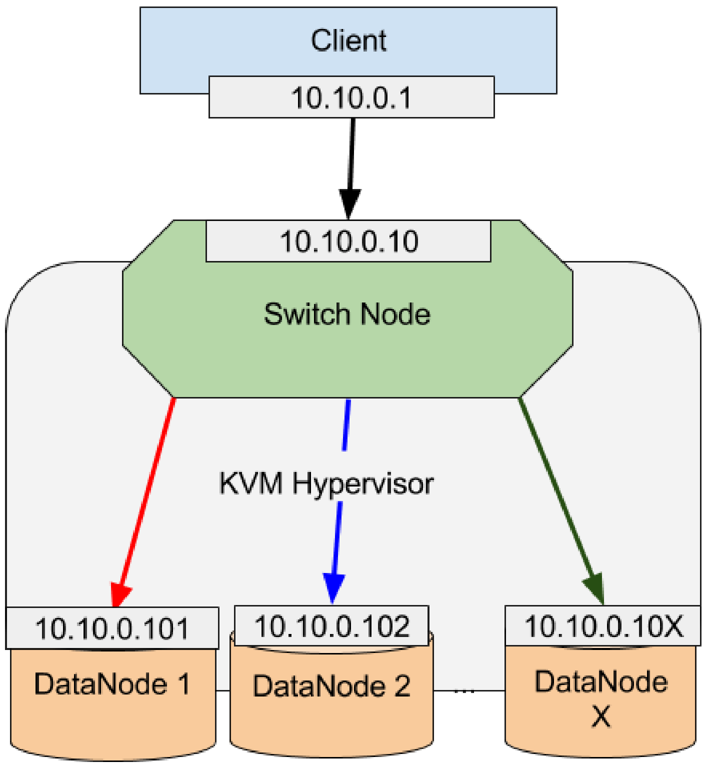

Figure 6.1: Data Flow for Experiment 1

In addition to writes, read performance was measured on the three-node system. It should be noted that since the system only needs to read from a single copy that no benefit to request times was expected from the proposed architecture. However, the delay introduced by moving data to and from the userspace of the switch in the proposed system could have had a measurable effect on reads. The results of this test are in table 6.4.

In fact, the opposite was observed. In almost all cases when reads were coalesced by the Switch Node they arrived slightly faster. However this gain was almost negligible and is likely due to the lack of link competition between Data Nodes when their reads are

Size	256MB	512MB	1GB	2GB	4GB	8GB	16GB
Switch	2.334	4.785	9.536	19.937	37.66	81.27	175.55
NameNode	6.842	13.684	27.367	54.733	109.46	218.93	437.46

Table 6.1: Write completion time for increasing transfer sizes in seconds (3-copies)

Size	256MB	512MB	1GB	2GB	4GB	8GB	16GB
Switch	2.392	4.731	9.548	19.736	46.018	89.299	153.384
NameNode	9.124	18.256	36.505	73.011	146.358	291.988	583.484

Table 6.2: Write completion time for increasing transfer sizes in seconds (4-copies)

Size	256MB	512MB	1GB	2GB	4GB	8GB	16GB
Switch	2.308	4.571	9.155	18.717	37.208	73.637	146.779
NameNode	11.418	22.835	45.661	91.303	182.663	365.319	729.884

Table 6.3: Write completion time for increasing transfer sizes in seconds (5-copies)

Size	256MB	512MB	1GB	2GB	4GB	8GB	16GB
Switch	2.294	4.584	9.182	18.381	36.804	73.532	148.246
NameNode	2.380	4.774	9.428	18.949	36.926	73.789	147.363

Table 6.4: Read completion time for increasing transfer sizes in seconds (3-copies)

coalesced by the Switch Node as opposed to being transmitted to the Name Node.

While single-threaded performance is a good starting point, modern DFS typically service hundreds to thousands of users at once [3, 1] so Data Nodes should be prepared to handle accesses from multiple users simultaneously. The added load of many different users should not have an effect on the performance of such a system. To this end, two multithreaded tests were performed. The first generated 8 threads which each wrote then read 1GB of data apiece, again in 512k chunks. The results of this test with different nodes is in table 6.5.

Copies	3	4	5
Switch	91.941	93.421	93.738
NameNode	194.565	255.464	317.214

Table 6.5: Total request completion time for 1GB RW from 8 simultaneous users (seconds)

To further stress the system, 256 threads were generated to read and write 16MB of data apiece. This simulates the typical cycles of a DFS devoted to small file sharing (such as Facebook's image engine).[14] The results of this are in tables 6.6 and 6.7.

Copies	3	4	5
Switch	36.704	36.730	36.709
NameNode	36.540	36.554	36.528

Table 6.6: Total request completion time for 16MB Read from 256 simultaneous users (seconds)

Note that 16MB requests from 256 simultaneous users generates 4GB of total requested data. While the added load slowed down the overall system slightly, the system

Copies	3	4	5
Switch	39.908	41.887	42.521
NameNode	109.781	146.509	182.978

Table 6.7: Total request completion time for 16MB Write from 256 simultaneous users (seconds)

proved robust enough to handle the additional requests with minial slowdown, even in the relatively restricted environment of the switch VM.

6.2 Experiment 2

For the next experiment, the environment from experiment 1 was reconfigured with 3, 4, and 5 Data Nodes. A RAID-5 array was then laid on top of the Data Nodes, so that data was striped between them. interspersed with the data was a simple bitwise XOR of the data from that same stripe from the other nodes, so that in the event of a single node's failure lost data could be regenerated with as simple an operation as possible. The DFS configuration is shown in Figure 6.2.

A single disk was then removed from the array and a new disk added. This prompted a rebuild operation by the NameNode, and the time it took to rebuild the 16GB node was then measured. The results of this operation are in table 6.8. The results show that the rebuild time is relatively constant for our proposed system while the time increases linearly with the number of Data Nodes in use for the control system.

DataNodes	3	4	5
Switch	221	219	243
NameNode	295	440	586

Table 6.8: Time to rebuild a failed 16GB Data Node (seconds)

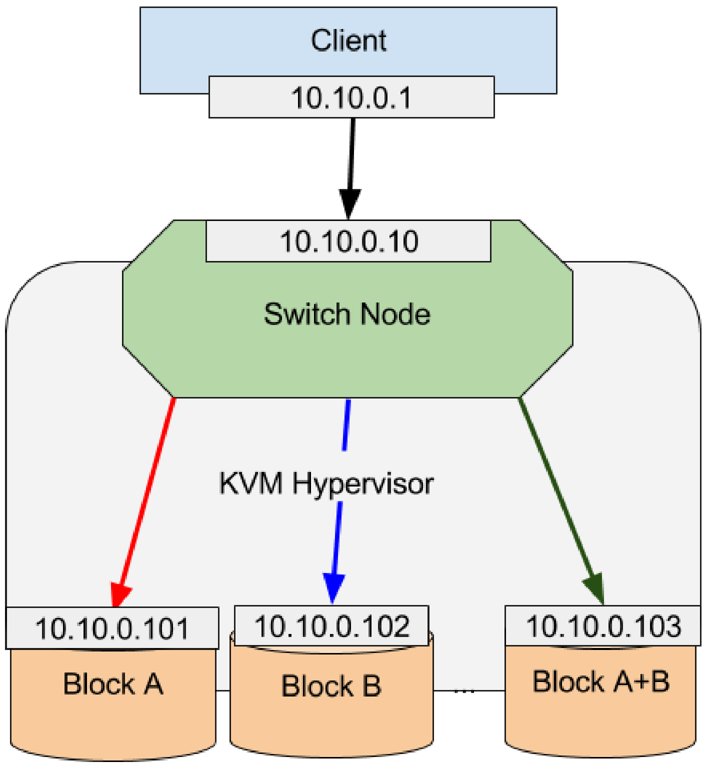

Figure 6.2: Data Flow for Experiment 2

6.3 Experiment 3

For the third experiment, all virtualized components of the system are moved to physical machines. Data Nodes are low power servers with RAM-based disks. The single Switch Node implementation is moved to an Altoline 6700 ONIE switch running the Pica8 Network OS in Open VSwitch mode. MDADM and target software was rewritten and recompiled to run on the Altoline switch since it ran in a different architecture than the switch VM and had more limited resources. Only replication was implemented in the rewritten

MDADM software. Links between all nodes were increased to 10Gb speed to hopefully remove link speed as a bottleneck. The DFS configuration is shown in 6.3.

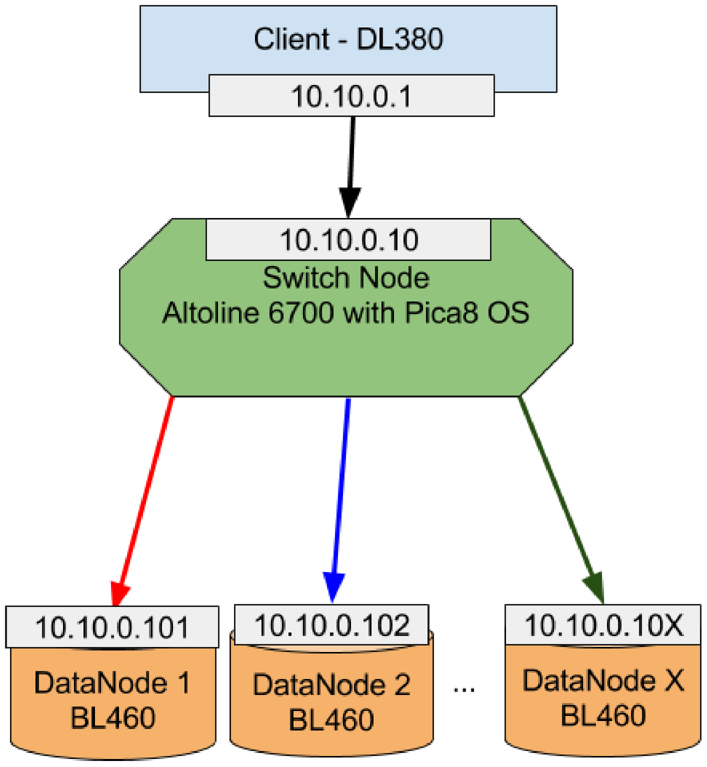

Figure 6.3: Data Flow for Experiment 3

The result of writes to a 4 data-node array set up only to do replication in experiment 3 is given in table 6.9

The first thing to note is that the throughput of the baseline data indicates this environment is still somewhat limited by the available network bandwidth. Testing of the

Size	256MB	512MB	1GB	2GB	4GB	8GB	16GB
Switch	0.513	1.078	2.198	4.314	8.437	17.619	34.498
NameNode	1.304	2.798	5.945	15.042	41.271	63.939	110.956

Table 6.9: Write completion time in a physical environment (4-copies)

individual disks show that each one is capable of about 7Gbps of bandwidth while the baseline reaches its maximum at 2.2 Gbps. While the data does show a decrease in completion time when replication is done at the switch (and a decrease in time overall due to the higher link speed), this decrease is not as drastic as seen in experiment one. Bandwidth for replication at the switch was never measured to be higher than about 5Gbps from the Name Node. After obtaining technical drawings of the design of the Altoline 6700 switch, it was found that the switch could only support about 20Gbps of traffic between the switch ASIC and the user-space CPU where my replication software was running. While other switch designs (including the Intel design mentioned earlier) could handle more traffic, this did present an opportunity to test the storage orchestrator algorithm of the proposed design.

6.4 Experiment 4

For this experiment, a basic tree hierarchy was set up as shown in figure 6.4 with the Name Node and Client at the head of the tree and Data Nodes at the leaves of the tree. The storage orchestrator software was then configured to examine the given network and attempt to build a storage replicating tree given knowledge of the topology. The orchestrator algorithm connected to agents running on the switch which manually examined switch counters to determine the current bandwidth through the individual links of the switch. After it determined the desired topology, it contacted those same agents to further build the replication tree though multiple nodes. Traffic was then run from the Name Node through the replication tree and the results in table 6.10 were obtained.

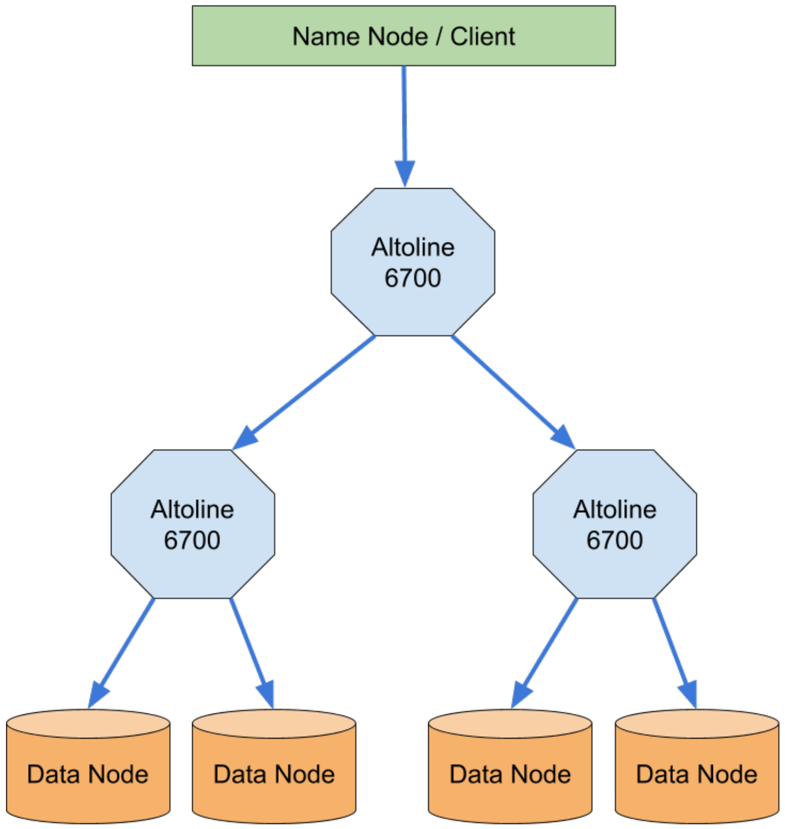

Figure 6.4: Network Topology for Experiment 4

Size	256MB	512MB	1GB	2GB	4GB	8GB	16GB
Switch	0.403	0.736	1.453	3.033	5.862	12.355	25.598
NameNode	1.282	2.764	5.747	11.885	25.344	52.618	108.716

Table 6.10: Write completion time in a tree topology (4-copies)

These results show that the 4-node tree was able to write at close to the maximum bandwidth of the 4 Data Nodes simultaneously by distributing the replication duties across the network. The network was able to do this reliably through protocols and implementations which are already used in DFS systems in the industry.

7. CONCLUSION AND FUTURE WORK

In this these, we proposed a new node and system architecture to solve the problems of limited name node bandwidth and high node reconstruction bandwidth in distributed filesystems. We described the new “Switch” Node in detail and detailed how it could be integrated with existing distributed file systems. We discussed the benefits of having this Switch Node do data replication and reconstruction operations. We described how Name Nodes could use knowledge of the network topology to configure them efficiently using a ”storage tree algorithm".

We then emulated a distributed file system and demonstrated the improved request latency and bandwidth utilization that could be achieved with our proposed system in a congested distributed file system doing only basic replication. We then implemented a basic XOR parity scheme in the system and demonstrated that rebuild times were significantly reduced when using our proposed system. Next, we implemented our proposed Switch Node on actual switch hardware and measured the effect that limited switch resources had on the proposed system. Finally, we demonstrated how our proposed storage tree algorithm could surmount the problems of limited switch bandwidth by increasing the number of network switches in the DFS and dividing replication duties among them.

For future experiments, the proposed system applied to an existing distributed file system such as Ceph or Hadoop to show its benefits in those systems. There is also a lot of opportunity in moving the actual replication and recombination duties from the switch CPU into the switch dataplane ASIC itself in order to reduce switch CPU overhead, though care must be taken to ensure that this is done in a reliable fashion. Optimizing switch ASICs to handle these kinds of loads without impacting normal traffic would also be an interesting area of research.

REFERENCES

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp. 1–10, IEEE, 2010.

[2] Intel, "Intel onp switch reference design," Tech. Rep. 326647-003US, Intel Corporation, February 2013.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, pp. 29–43, ACM, 2003.

[4] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE transactions on information theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[5] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," in *Proceedings of the VLDB endowment*, vol. 6, pp. 325–336, VLDB Endowment, 2013.

[6] A. Mankin, V. Paxson, A. Romanow, and S. Bradner, "Ietf criteria for evaluating reliable multicast transport and application protocols," 1998.

[7] E. B. Boyer, M. C. Broomfield, and T. A. Perrotti, "Glusterfs one storage server to rule them all," tech. rep., Los Alamos National Laboratory (LANL), 2012.

[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on operating systems design and implementation*, pp. 307–320, USENIX Association, 2006.

[9] Unomena, “ONIE - open network install environment.” Web. http://onie.opencompute.org/, 2015. Accessed: 2016.07.21.

[10] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, “Extending networking into the virtualization layer..” Web. http://openvswitch.github.io/papers/hotnets2009.pdf/, 2009.

[11] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, “Scale and performance in a distributed file system,” *ACM transactions on computer systems (TOCS)*, vol. 6, no. 1, pp. 51–81, 1988.

[12] D. Muntz and P. Honeyman, “Multi-level caching in distributed file systems,” in *Proceedings of the Usenix Winter 1992 technical conference*, pp. 305–313, Citeseer, 1991.

[13] S. Narayan, S. Bailey, and A. Daga, “Hadoop acceleration in an openflow-based cluster,” in *High performance computing, networking, storage and analysis (SCC), 2012 SC Companion:*, pp. 535–538, IEEE, 2012.

[14] B. Depardon, G. Le Mahec, and C. Séguin, “Analysis of six distributed file systems.” Web. https://hal.inria.fr/hal-00789086/document, 2013.

[15] G. Donvito, G. Marzulli, and D. Diacono, “Testing of several distributed file-systems (hdfs, ceph and glusterfs) for supporting the hep experiments analysis,” in *Journal of physics: conference series*, vol. 513, p. 042014, IOP Publishing, 2014.

[16] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, “Erasure coding in windows azure storage,” in *Presented as part of the 2012 USENIX annual technical conference (USENIX ATC 12)*, pp. 15–26, 2012.

[17] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandic, “Opening the chrysalis: on the real repair performance of msr codes,” in *14th*

USENIX conference on file and storage technologies (FAST 16), pp. 81–94, 2016.

[18] N. Bellinger, “Linux scsi target-lio.” Web. http://linux-iscsi.org/wiki/LIO, January 2016. Accessed: 2016.01.29.

[19] M. Christie, “Open-iscsi project on github.” Web. https://github.com/mikechristie/open-iscsi, January 2016. Accessed: 2016.01.29.

[20] J. Axboe, “Fio readme.” Web. https://github.com/axboe/fio/blob/master/README, January 2016. Accessed: 2016.01.29.

[21] W. Reese, “Increase performance, reliability and capacity with software raid,” *Linux Journal*, vol. 2008, no. 175, p. 3, 2008.