STORAGEFLOW: AN SDN APPROACH TO STORAGE NETWORKS

A Thesis

by

PRADIPTA KUMAR BOSE

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

August 2014

Major Subject: Computer Engineering

ABSTRACT

Network Attached Storage (NAS) systems have become popular due to their efficiency, ease of use, and ability to protect and restore data. Many NAS implementations provide efficient service and utilize sophisticated techniques such as coding and striping of data to better utilize the available space, provide fast recovery from disk failures and avoid loss of data. Unfortunately, the current architectures are complex and inflexible which necessitates the need to introduce greater flexibility and support for experimentation. Additionally, there is a significant potential to improve the performance of the system by leveraging regenerative coding techniques and by allowing the intermediate network nodes to perform encoding operations.

OpenFlow (OF) is a rich SDN protocol that has gained significant popularity in recent years. OpenFlow defines a standard communications interface between the control and forwarding layers of an SDN architecture, as well as the forwarding architecture of a switch. While OpenFlow currently supports only a limited number set of protocols, it has attracted significant attention from both industry as well as research community and has significant potential to be widely adopted by the industry.

The key idea of this thesis is to utilize multifunctional SDN-enabled switches that can perform both traditional forwarding operations as well as new encoding operation on the packets. For this purpose, we propose to extend the OpenFlow datapath by enabling the switch to perform encoding operations on select flows upon the request from the controller. Our approach utilizes commodity hardware, which makes it cost-efficient and attractive. In contrast to the traditional approaches which rely on dedicated servers to perform coding and striping operations, our approach has better performance and flexibility, and can be easily customized to serve the requirements of a particular storage scheme. In addition, our approach makes it easier to experiment with new applications, including the use of different encoding schemes by enabling fast prototyping and testing.

Since none of the existing SDN protocols (including OpenFlow) provide support for basic storage functions such as striping and coding, we propose several extensions of the OpenFlow protocol to support such functionality as well as encoding operations. The extensions we develop are part of a systematic approach to design an SDN-enabled NAS system. We identify some common design trade-offs and evaluate their impacts on performance and reliability. Furthermore, the thesis presents a forwarding data path extension that uses custom data structures and groups at the switch. This design also effectively reduces required bandwidth and enables traffic engineering and load balancing at network links.

DEDICATION

To my family and friends

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Alex Sprintson, for all the mentoring he has given me in this work and throughout my entire graduate school experience. During this period, I have grown a tremendous amount both in my technical aptitude and as a person. I would also like to sincerely thank Jasson Casey for his advice. I am very grateful for the help I received from my colleagues from Flowgrammable, Colton Chojnacki, Atin Ruia, Muxi Yan, and Daoqi Wang, as well as Swanand Kadhe and Corey Morrison for this work. Finally, I want to thank Dr. Narasimha Reddy and Dr. Radu Stoleru for having served as my committee members.

Last but definitely not the least, I am extremely grateful to my family and friends for their constant support.

NOMENCLATURE

TLS	Transport Layer Security
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
TFTP	Trivial File Transfer Protocol
FTP	File Transfer Protocol
SDN	Software Defined Networks
OF	Open Flow
SAN	Storage Area Networks
NAS	Network Attached Storage

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Network Attached Storage (NAS) is increasingly important, given the magnitude of data being generated in applications by users on an everyday basis. The current solutions are efficient, but lack in flexibility. Also, more efficient utilization of the network resources can be done by pushing additional functionality to the switches. We propose a new SDN-based solution, referred to as StorageFlow, to address these issues and quantify the gains achieved by such a design.

1.1 Conventional Storage Architecture

Conventional storage network architectures consist of separate devices for network and storage functionalities and span the entire range from simple RAID disks to data centers. There are two main types of storage architectures, namely, Storage Area Networks (SAN) and Network Attached Storage (NAS). SANs provide OS block-level access to data. SANs utilize protocols such as iSCSI (Internet Small Computer System Interface), Fiber Channel and Infiniband. In contrast, NAS provides OS file-level access to data through an application layer protocol. NAS is entirely software-based and include protocols such as NFS (Network File System), FTP (File Transfer Protocol), etc.

Distributed storage systems ensure reliability by introducing redundancy in the system, through techniques such as mirroring and coding. With mirroring techniques, the data is replicated across several disks. Coding techniques allow us to maintain reliability while using less storage. Such techniques avoid replication of data, instead the redundancy is maintained by storing parity (encoded) data. One of the fundamental problems in NAS systems is recovery from failures: if a node storing encoded information fails, in order to maintain the same level of reliability we need to rebuild parity at a new node. Erasure codes can be used to address specific requirements to maintain efficient resource utilization, while providing reliability [7]. Lately, to further improve resource utilization, the network coding techniques, such as regenerative coding, are being increasingly explored and used.

1.2 Coding for Storage

Architectures for storage ensure reliability by coding and distributing data across disks. Different schemes provide a different balance between reliability, availability, performance and storage capacity. For example, erasure codes transform a block of data into a longer block, such that the original data can be regenerated from a subset of the symbols in the final block. Specifically, say that each data unit such as a file is divided into k symbols. The erasure code generates $n-k$ symbols such that any k of n resulting symbols is sufficient to restore the data.

Regenerative codes are a category of codes, specially designed for storage, which address the reconstruction of lost data. This has special applications in distributed storage systems where there is a need to minimize the amount of bandwidth required to restore data redundancy after failure. An (n, k) Minimum Distance Separable (MDS) Regenerating Code can tolerate the failure of any $n-k$ storage nodes. The goal of regeneration codes is to minimize the amount of data required to be downloaded during a node repair to a theoretical minimum. Thus, these codes improve system performance by minimizing the network resources consumed during the repair process.

Despite its relatively young age, the area of regenerative coding for storage is well-studied, with a variety of coding schemes available, as surveyed by Dimakis et al. [7]. However, most studies assume that an underlying network has a mesh topology, with all coding done at end nodes. Hence, there is an opportunity to improve the performance of regenerative codes in NAS systems by taking advantages of their specific properties (such as the fact that these systems typically use a ring topology). Furthermore, the performance of regenerative codes can be further improved by enabling coding capabilities at network switches.

1.3 Motivation

Conventional architectures for Network Attached Storage systems consist of separate network switches and storage servers. Storage servers are tasked exclusively with functionalities such as coding and striping while network switches are tasked with functionalities related to packet processing. This design can be significantly modified by using Software Defined Networks, leading to separation of control and forwarding functions. Such an approach has various potential benefits, chiefly offering greater flexibility, enabling innovation, and reducing costs due to the use of generic SDN network hardware.

Design of SDN-enabled NAS networks would require modifications to existing SDN protocols. This is because there are no solutions in the SDN domain to address storage-related problems. Since there is a lack of literature exploring this approach and an exploration of this design space could be rewarding, we undertake the task of producing a feasible design.

We have a number of choices for designing with SDN. Chiefly, we have to decide on an SDN standard which will be suitable. Of the choices available, OpenFlow [8] is an option which has particularly gained traction in the SDN community. OpenFlow (OF) is a protocol designed for use in Software Defined Networks, for communication between a controller and an OpenFlow-enabled switch. OpenFlow also provides switch abstractions. This protocol allows one to easily deploy innovative routing and switching protocols in the network. It also enables the possibility of having applications suited to end use. Accordingly, in our reference architecture, we use OpenFlow as the underlying SDN protocol. However, our methods are not limited to OpenFlow and can be applied to other SDN protocols with minimum modifications.

A successful implementation of a storage network should be able to support standard functions, namely, *read*, *write* and retrieval of metadata, as well as have a provision for timeout. Trivial File Transfer Protocol [6] is a file transfer protocol, implemented over the User Datagram Protocol (UDP), notable for its simplicity. TFTP only reads and writes files from or to a remote server and lacks more sophisticated functionality. It is light and easy to implement and is suitable for experimental purposes. However, this simplicity lends itself to ease of experimentation and enables us to focus on the key features of a NAS system. Accordingly, we adopt TFTP as a file access protocol in our reference architecture. It is important to note that our approach can be used for constructing a reference architecture that uses any other file access protocol.

The goal of this thesis is to design an efficient SDN-based NAS architecture and to compare its efficiency relative to more conventional designs. There are two major objectives for this thesis. The first is to create a design using SDN which would support coding and striping functions. The second objective is to measure the performance of this design on relevant parameters. With a design fulfilling minimal levels of flexibility and efficiency, this work contributes to an understanding of the design trade-offs involved and presents a systematic and disciplined approach for design and implementation of storage networks using SDN.

1.4 OpenFlow

Software Defined Networks is an approach to networking which allows abstraction of lower level network services by decoupling the data and control plane. This leads to simplification of network management and allows a logically central controller to determine the behavior of the network elements (such as forwarding operations and packet discards). OpenFlow [3] is an SDN protocol that operates over TCP/TLS on the application layer. It defines a set of messages for controllers to configure switch states and carry out desired operations to modify traffic in the swithc, as shown in Figure 1.1. The protocol separates the control plane away from the networking devices in order to achieve a more centralized control on an otherwise distributed network. It achieves this through manipulation of flow tables on the switch through a variety of data structures and messages. There have been a total of 8 versions of the protocol defined by Open Networking Foundation, which acts as the standards development organization for OpenFlow SDN, with version 1.4 being the latest iteration.

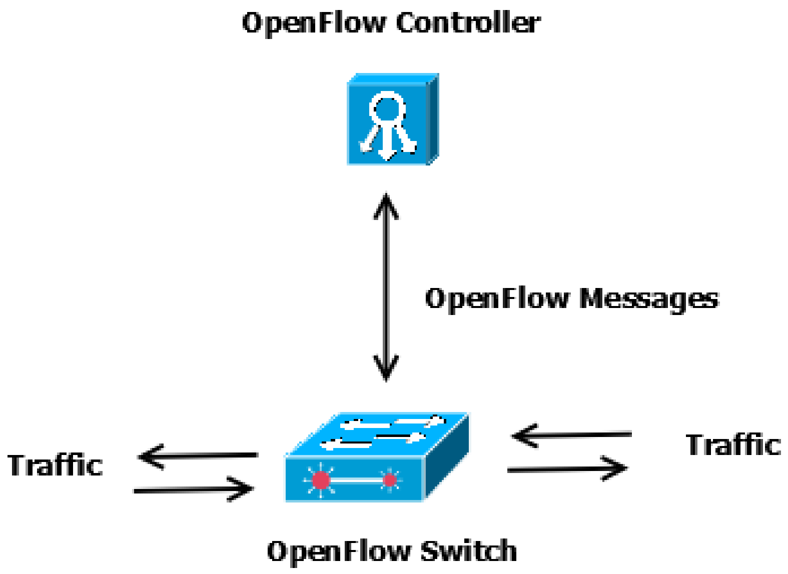


Figure 1.1: OpenFlow Controller and Switch

The switch architecture is well-defined, with a dataplane which is traversed by packets, and a switch agent to interact with the controller. An overview of this architecture is given in Figure 1.2.

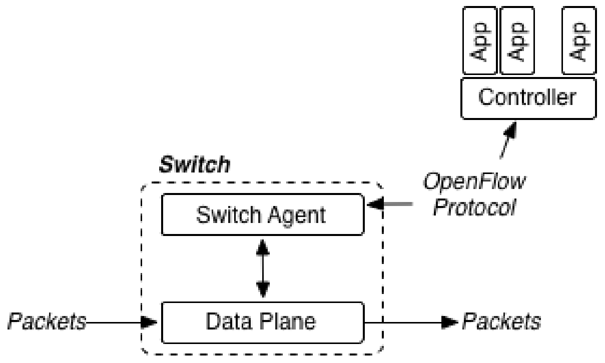


Figure 1.2: Switch Architecture and Relationship with Controller

The data plane is a prime component of the architecture of the switch. It refers to the ports, flow tables, group tables, groups, flows, flow classifiers, instructions and actions [2], as shown in Figure 1.3. Ports are the entry and exit points for packets, into and out of the switch. These packets are matched to flows, an abstraction used in OpenFlow, using classifiers. Flow tables map these flows to corresponding sets of actions. Additionally, flows may be aggregated into groups to provide similar treatment for packets belonging to different flows. Group tables keep track of the composition of groups and actions specific to each.

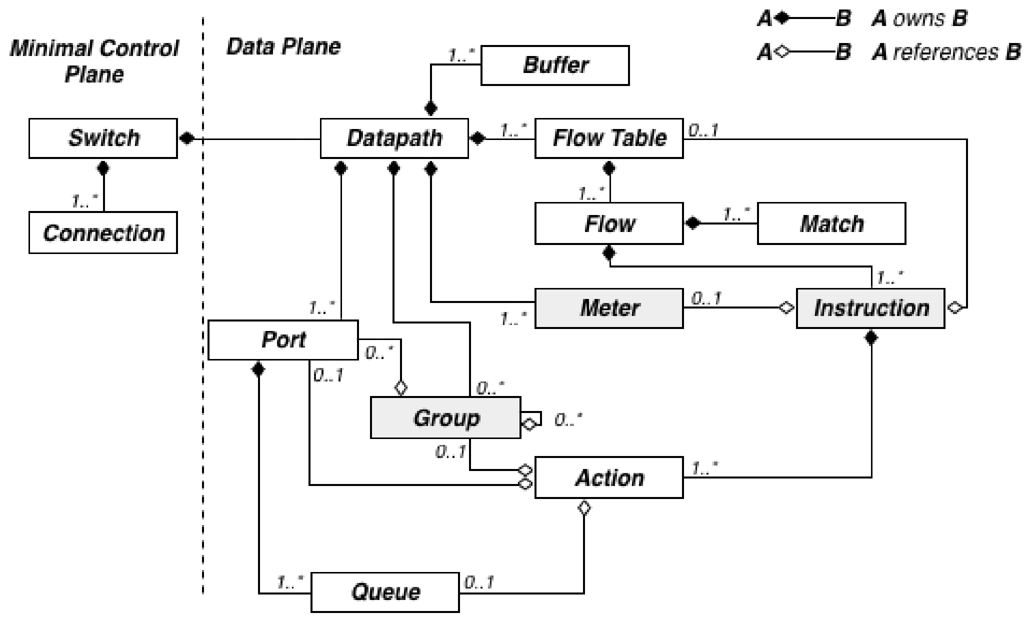


Figure 1.3: Generalized OpenFlow Dataplane

All packets undergo the same process as they traverse the switch data plane. A particular flow in the flow table is selected, using a key constructed from information extracted from the packet and its metadata. Subsequently, the matched action set can drop, mutate, queue, forward, or direct that packet to a new flow table, as referred to in Figure 1.4.

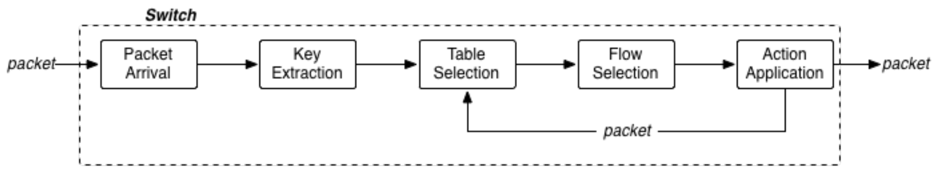


Figure 1.4: Dataplane Packet LifeCycle

OpenFlow packet signatures are contained in a message structure called Match, which is then used for classification, as shown in Figure 1.5. Packets are generally classified into flows based on their destination/source MAC, IP and port, among other packet information. For our purposes, we will need to use further information to help classify the relevant packets, which may be achieved by using match extensions.

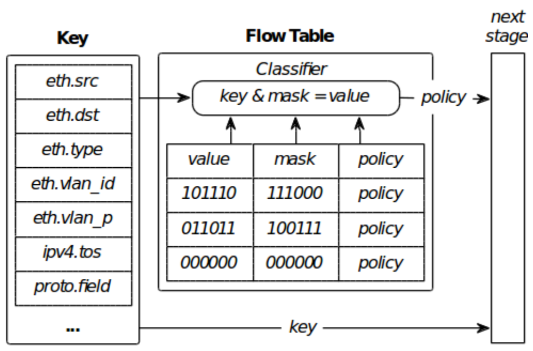
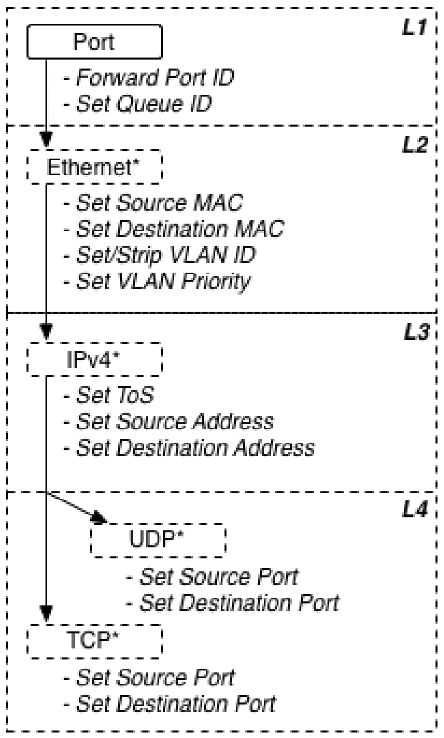


Figure 1.5: Illustration of Matching on Packet Signature in OpenFlow

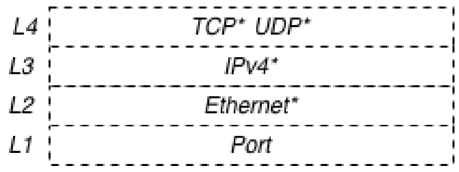
OpenFlow *Actions* specify the policies on the packets matched to corresponding flow entries. These include forwarding the packet to a specific port (type *Output*) and inserting the packet into a particular queue in a packet (type *Enqueue*). Several policies can be applied on the same flow by attaching a vector of *Actions* with various types in the end of flow modification (*FlowMod*) messages. *Actions* have dependencies and can be layered in a stack as shown in Figures 1.6a and 1.6b. For our purposes, we may need to extend the actions to better serve our application.

PacketIn is a message type issued by the switch to the controller. Its main function is to query the controller for the actions for an unknown flow that does not have an entry in the switch flow table, as shown in Figure 1.7.

FlowMod is a message type sent from an OpenFlow controller to the switch in order to modify its flow table, as shown in Figure 1.8a. It consists of a *Match* to classify the flows and a vector of *Actions* to define the policies on these flows. From



(a) Action Dependencies



(b) Action Stack

Figure 1.6: Action Dependencies and Stack

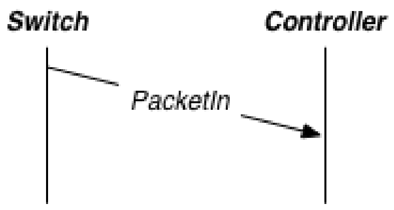
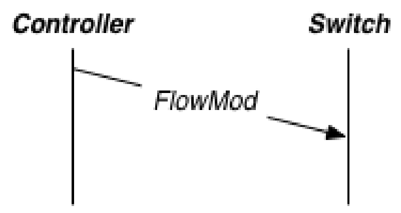
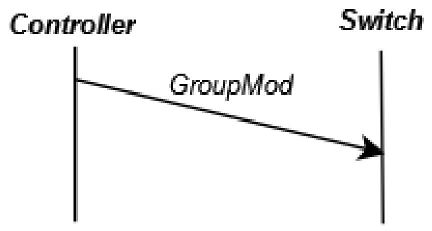


Figure 1.7: PacketIn Message

version 1.1.0 onwards, *FlowMod* carries the *Instruction* structure, which carries an *Actions* list, to modify the *Action* set, for that *Match*.



(a) FlowMod Message



(b) GroupMod Message

Figure 1.8: Switch Tables' Modification Messages

GroupMod is a message type sent from an OpenFlow controller to the switch in order to modify its group table, as can be seen in Figure 1.8b. This message was introduced in Version 1.1. It has a bucket which consists of an action set which follows the rules in *FlowMod*. The controller has applications, which can be programmed to modify network behavior according to the usage requirements of the network. Figure 1.9 shows the usual order of messages between the applications and the controller.

1.5 Trivial File Transfer Protocol

TFTP suits the purposes of our experimentation as it is a simple protocol, built on top of UDP. It is used to move files between machines on different networks. It is

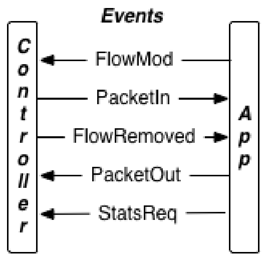


Figure 1.9: Sequence of Messages between Controller and Applications

small and easy to implement and lacks most of the features of a regular file transfer protocol. However, it fulfills certain basic requirements expected of a file transfer protocol and is thus, ideally suited for our experimentation. Standard exchanges which we are concerned with are File Read, as shown in Figure [1.10], and File Write, as shown in Figure [1.11].

TFTP has certain basic features, specifically, the protocol supports five kinds of packets, each of which has a separate opcode, listed in the table [1.1]. The first packet of the transfer is sent to port 69 of the server from an ephemeral port of the client. The corresponding response to first packet is sent from an ephemeral port, which handles all subsequent packets for that particular transaction. Hence, on completion of the first packet pair, both parties must make note of the corresponding ports and direct future packets to them for the duration of that transfer.

Table 1.1: Opcode vs. Packet Types for TFTP.

Opcode	Packet/Operation Types
1	Read Request (RRQ)
2	Write Request (WRQ)
3	Data (DATA)
4	Acknowledgment (ACK)
5	Error (ERROR)

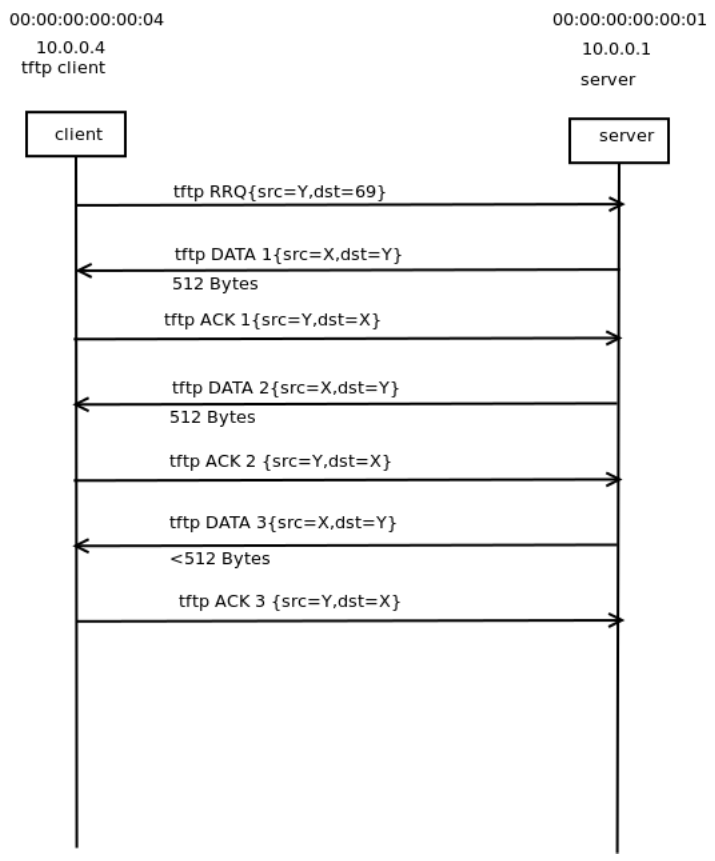


Figure 1.10: TFTP Read Sequence

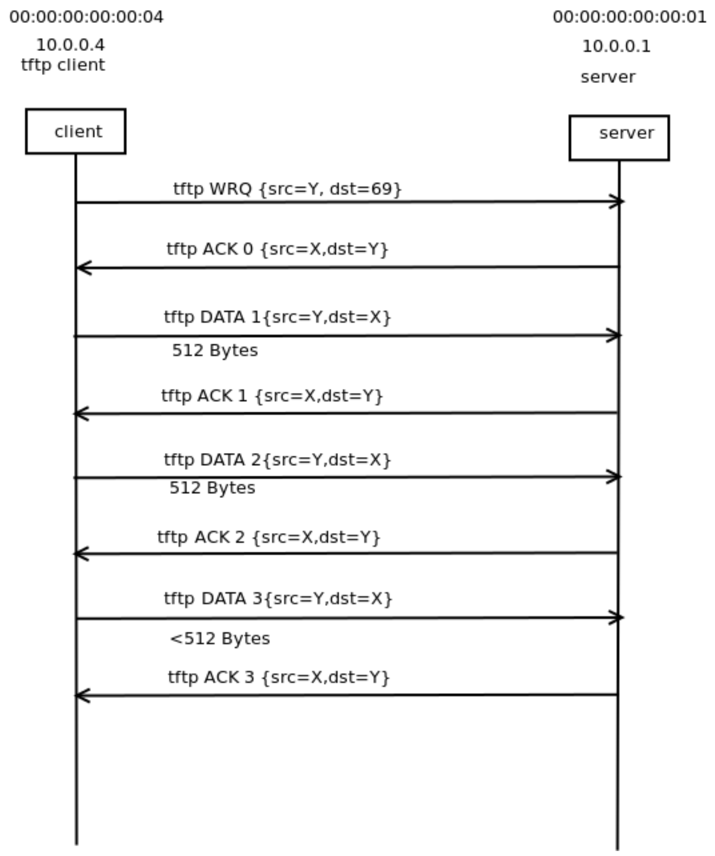


Figure 1.11: TFTP Write Sequence

Another feature of the protocol is that acknowledgement packets from the client notify the server of correct receipt of previously sent packet and prevent retransmissions. Failure to receive the ACK within a certain timeout value causes retransmission. The end of a transfer is marked by a DATA packet that contains between 0 and 511 bytes of data. This packet is acknowledged by an ACK packet like all other DATA packets and the host acknowledging the final DATA packet may terminate its side of the connection on sending the final ACK. However, it is encouraged that the host sending the final ACK wait for a while before terminating, in order to retransmit the final ACK if it has been lost.

1.6 Related Works

SDN has generated a lot of interest from the networking community, and consequently, there has been a plethora of research regarding its capabilities and applications. Nunes et al. [10] provide an overview of this technology and discuss the newer horizons opened up by this, as well as compare it to older technologies. It is notable that this may be used to solve practical problems which may not have been possible with older technologies, as well as have applications which would be more useful than those on conventional networks.

There has not been much exploration of NAS design with SDN space, to the best of our knowledge. Most published literature in storage domain tends to assume non-SDN networks and thereby focus more on maximizing the gains within those constraints. However, Nemeth et al. [9], discuss potentials with OpenFlow architectures and acknowledges coding performed in network switches as being one promising direction of research. This work discusses case studies where this architecture can be beneficial and is a pioneering work in this domain. The authors discuss using Bloom Filters, and Network Coding, and addition of new actions to OpenFlow to enable these novel actions. However, the work only tangentially broaches the topic and a detailed approach to practical implementation focussed on this problem is lacking.

There has been a lot of interest in the field of regenerative coding for storage networks. Dimakis et al. [7] provide a comprehensive overview of this domain. However, considerations for coding on NAS are notably absent, and this presents an opportunity for us to expand the applications of this into NAS. We undertake a study of suitability and advantages of such schemes on NAS, with coding functionality at switches.

2. DESIGN

Selection of design choices afffects the overall performance and efficiency of the system. Understanding the consequences of these choices can provide us a more systematic approach when it comes to implementing a storage network with SDN.

A simple SDN network topology, as shown in Figure 2.1, can be designed with an OpenFlow Controller and OpenFlow-enabled switches. The central controller would have an application customizing the network behavior for our purposes, and would support various functionalities related to storage. The switches may need to have additional functionality, like coding, in order to provide desired results.

Such a design would potentially allow us to obtain three advantages, namely, to leverage SDN to minimize network traffic, optimally route the traffic to avoid network congestion and maintain metadata about the network and file locations at a central location, preferably at the controller application itself. This design would also allow for flexibility in design, by using supported commodity network hardware, and a reduction in overall network complexity, by pushing certain functions to switches.

With the design mentioned here, a number of schemes could be possible. The schemes which are possible include data duplication without coding, design with controller application acting as full TFTP proxy, design with TFTP Proxy server inside the network and a design incorporating coding. Our work aims to systematically study these possibilites in greater detail and gradually try to evolve towards the most efficient design.

2.1 Conventional Architecture

To gain perspective, we discuss to discuss conventional architectures of data centers. Conventional architectures statically map web services to smaller networks.

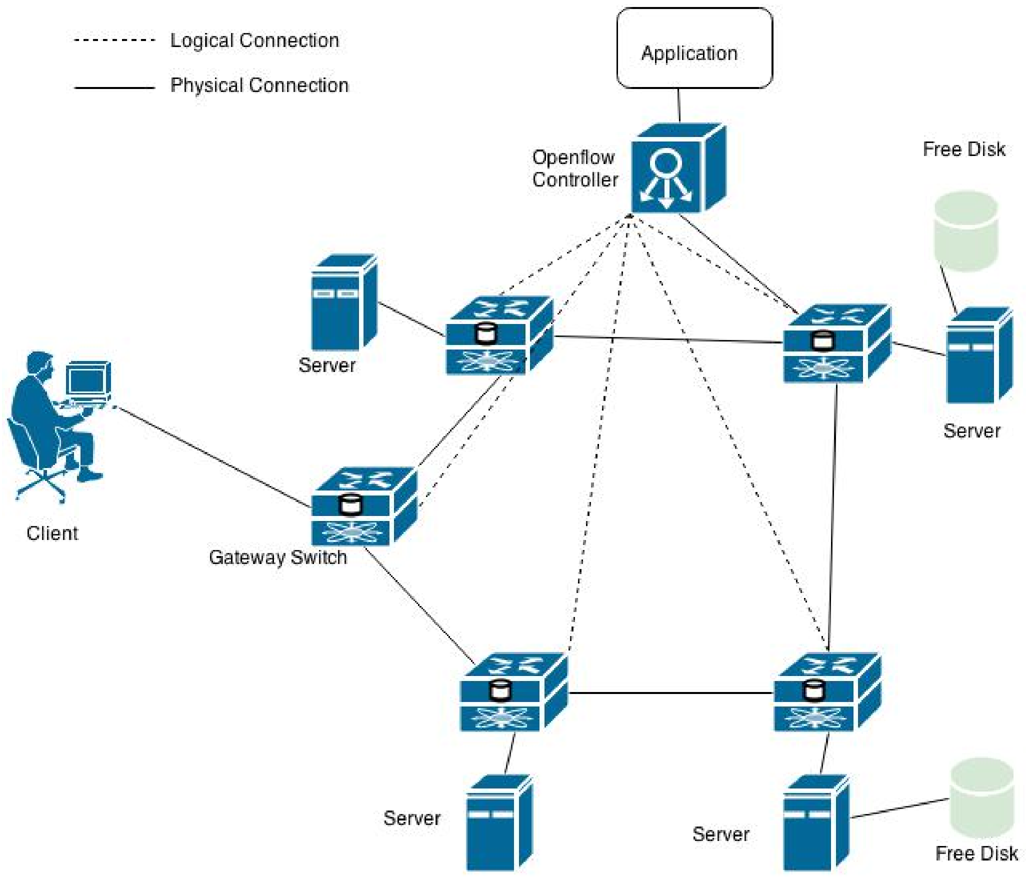


Figure 2.1: System Topology

The network generally consists of specialized devices to handle and route traffic and servers with attached disks for storage. Additionally, specialized devices to carry out striping and coding related functions are present. All traffic is routed internally, dependent on a combination of various factors, including availability of storage space, traffic congestion, and location of coding servers. This architecture inherently promotes the concentration of traffic at a few points in the network and on certain links. This in turn leads to an inefficient utilization of the available network resources. Additionally, this architecture is relatively rigid and does not support innovation in coding schemes or portocols.

2.2 Architecture with SDN

With SDN, we can think of overcoming some of the problems associated with conventional architectures. Hence, we discuss some possibilities before arriving at an optimum design. We keep in mind that the main advantage of SDN is to have a more centralized view of the network and possibly combining functionalities into network devices.

Initially, we attempt to see if storage functionalities can be provided by the SDN network without using customizations at the device level at all. We also check if simple packet loss cases would be taken care of by the design without modification of the TFTP protocol. This design consists of a controller application which keeps an overview of the entire network. All data is assumed to be duplicated by the network to at least two data servers in the network. The application is aware of the availability of all data and its locality. Thus whenever a request is sent to the storage network, a OF PacketIn message would be generated by the gateway switch and the controller would decide on how to process the request, based on the header of the packet. For read requests, the controller would redirect the transaction to one of the servers which contains requested data and is relatively less burdened at the moment. Similarly, for write requests, it would select two servers where the data could be written and ask the switch to duplicate the transaction to the selected servers. The sequence diagrams, depicted in Figure [2.2] and Figure [2.3] given here describe the network behavior and actions with greater clarity.

We ensure that the design is transparent and covers adverse cases without changes to the TFTP protocol and see that the network is able to respond in standard TFTP fashion.The sequence diagrams shown in Figure [2.4] and Figure [2.3] describe the network behavior and actions, in the face of packet loss.

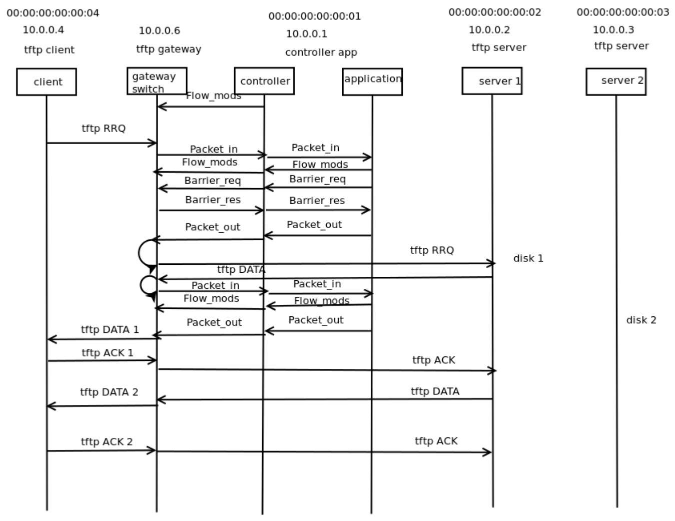


Figure 2.2: Sequential Diagram of Network Actions during Read Request for Case without Coding.

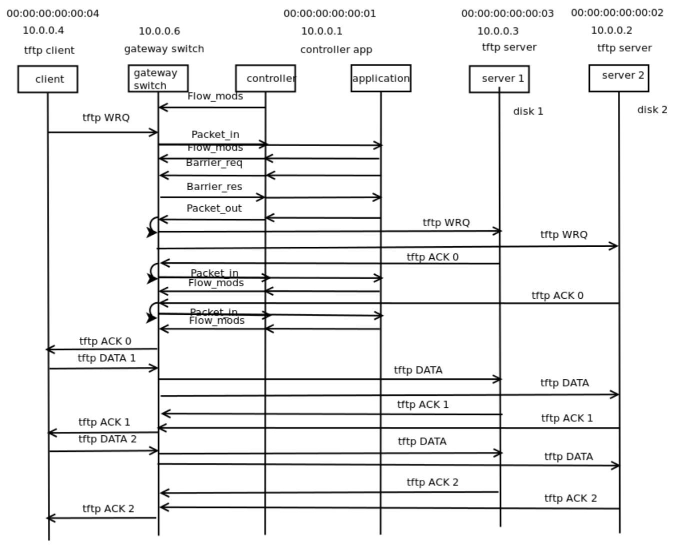


Figure 2.3: Sequential Diagram of Network Actions during Write Request for Case without Coding.

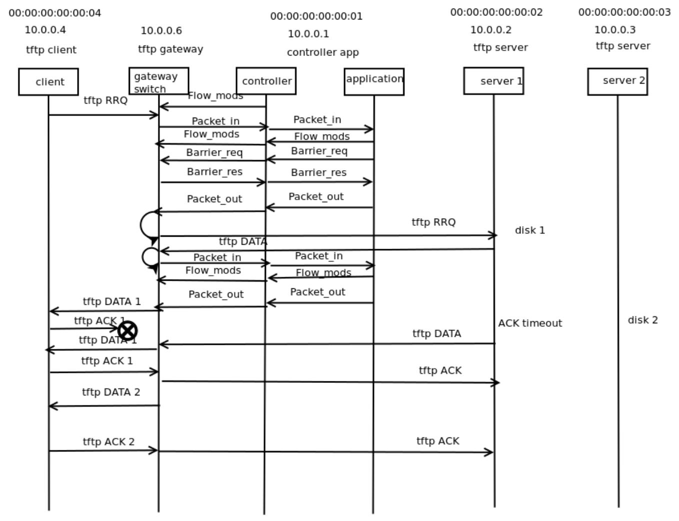


Figure 2.4: Sequential Diagram of Network Actions during Read Request for Case with Packet Loss, without Coding.

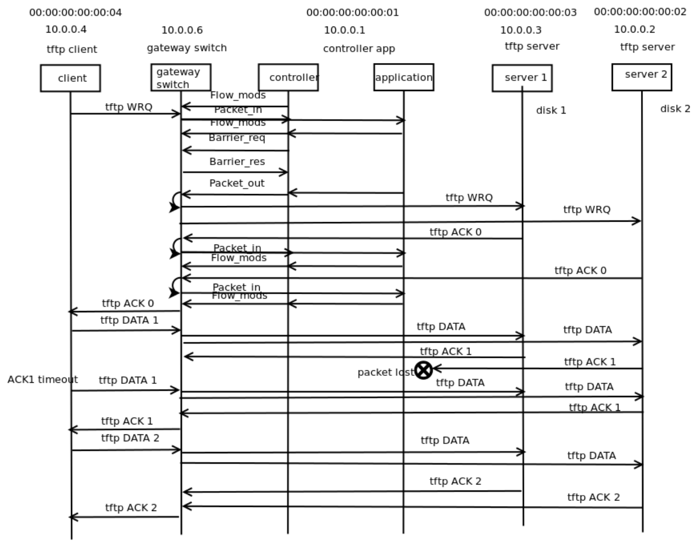


Figure 2.5: Sequential Diagram of Network Actions during Write Request for Case with Packet Loss, without Coding.

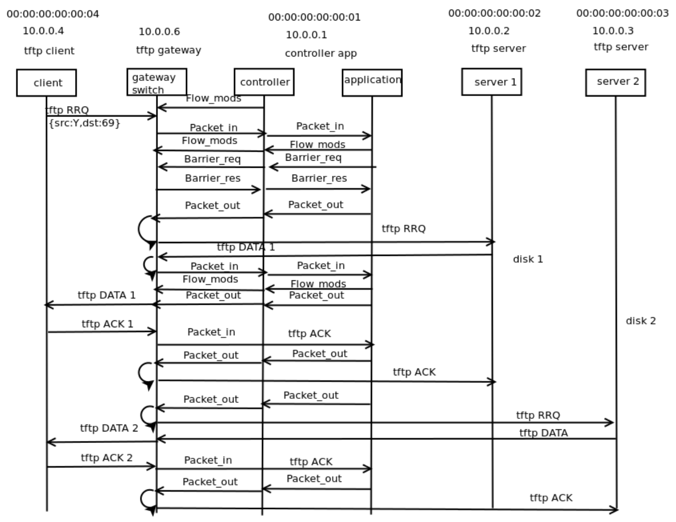


Figure 2.6: Sequential Diagram of Network Actions during Read Request for Case with Application acting as Full Proxy Server.

In order to improve upon the previous design, we try to incorporate coding into our architecture. As a first pass, we can assume that the controller application be responsible for coding and thence simply redirect packets to the internal TFTP servers. This is possibly the simplest architecture for accomplishing coding in an SDN network. Controller application acts as full proxy and is involved in all packet exchanges with external client and is aware of the contents of all disks in the storage network. All incoming packets are *PacketIn* to the application which creates corresponding OF *PacketOut* to respective file servers. The sequence diagrams, depicted in Figure 2.6 and Figure 2.7, bear out the viability of this design, for standard *Read* and *Write* transactions.

This design allows us to exploit the advantages of coding in reducing the storage space required while combining it with a SDN design. However, this design suffers from a major drawback, in the form of OF *PacketIn* messages to the controller. As all packets into the network are sent to the controller application for further

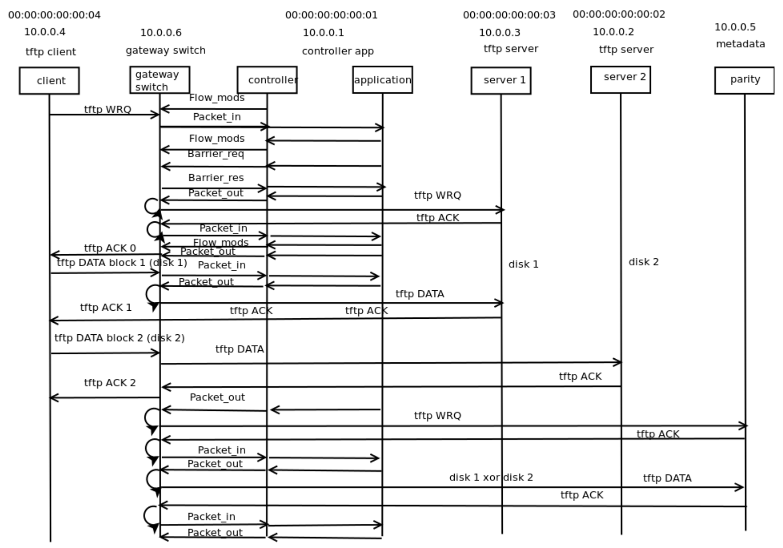


Figure 2.7: Sequential Diagram of Network Actions during Write Request for Case with Application acting as Full Proxy Server.

processing, the load on the controller would increase rapidly in a standard storage network. This would overload the network capacity beyond usability and essentially negates any advantages provided by use of coding.

The insights from these designs allow us to make an optimized design which would serve our purpose. We conceive a design where coding can be carried out on the switch and the controller application would serve only to make decisions about initial flows and keep track of availability of the data and its localities in the network. This design potentially allows us to gain advantages from coding as well as leverage the inherent advantages of SDN while avoiding the controller overload associated with the designs discussed previously. In this design, for the read sequence, coding of packets takes place in the switch with controller knowledge. The switch keeps packets from one server in a buffer until packet from the other server involved in this exchange reaches the switch, after which they are combined and forwarded. This high level description is discussed in Figure 2.8 and Figure 2.9, for *Read* and *Write*

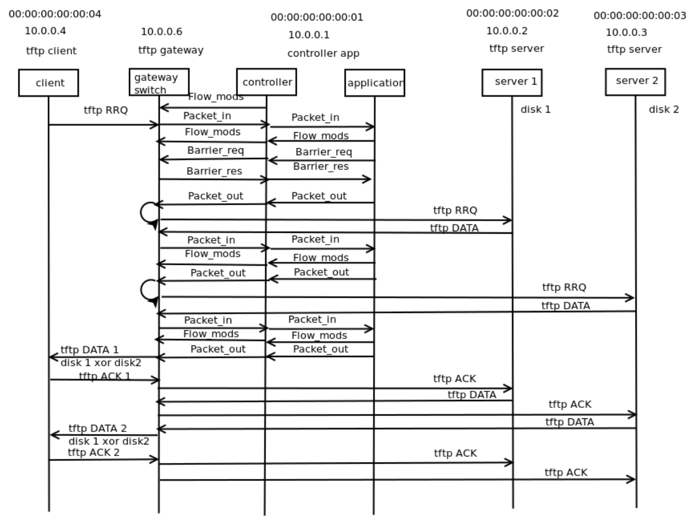


Figure 2.8: Sequential Diagram of Network Actions during Read Request for Case with Coding at Switch.

transactions.

The switch enqueues packets to explicit coding queue based on their source or their destination, depending on whether it is a read transaction or write transaction. Some data plane modification is required to carry out the standard transactions successfully. This design has advantages of SDN architecture, combined with the advantages of coding. There is a minimum overhead introduced and this design serves the purposes of our exploration well. However, it needs to fulfill basic transparency requirements to be able to work without modifications to the standard protocol. On closer scrutiny, it is able to handle packet loss cases without changes to the protocol , by adding simple timeout functionality to the switches and controller. This is described in the sequence diagrams shown in Figure [2.10] and Figure [2.11].

Having decided that this overall architecture would be the best to pursue further, we are faced with several basic questions to resolve, in order to make sure that the design works perfectly. We have to design a mechanism to detect disk failures. The

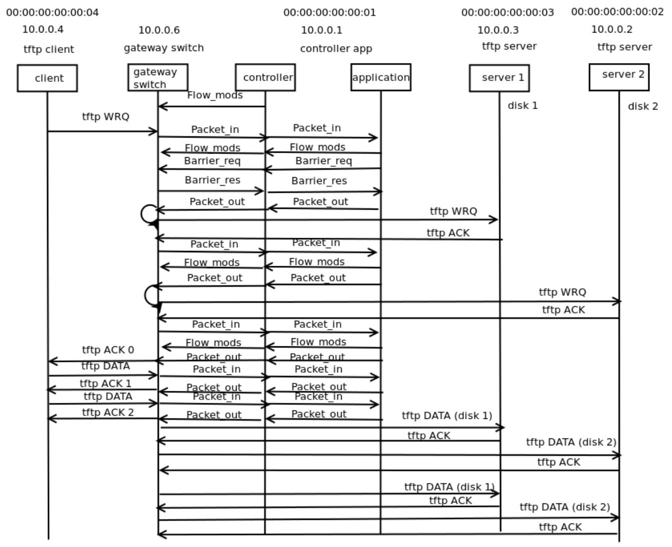


Figure 2.9: Sequential Diagram of Network Actions during Write Request for Case with Coding at Switch.

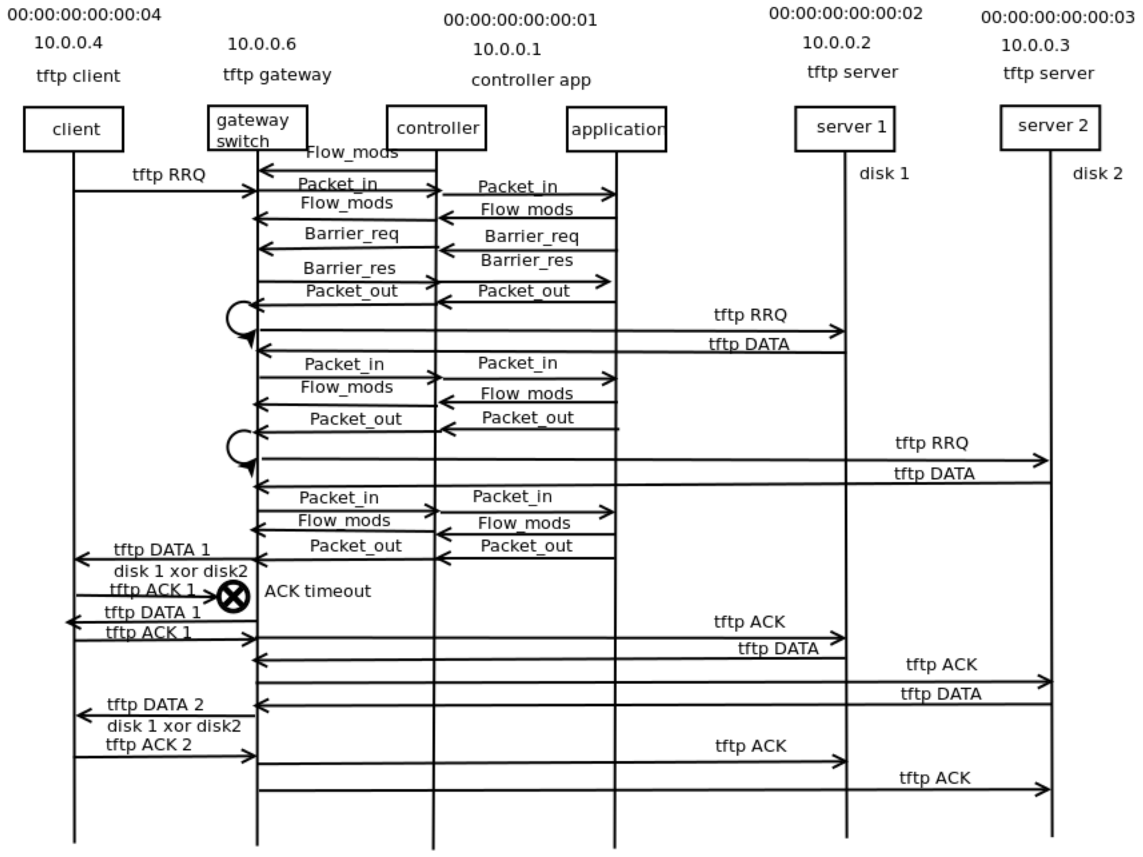


Figure 2.10: Sequential Diagram of Network Actions during Read Request for Case with Coding at Switch with Packet Loss.

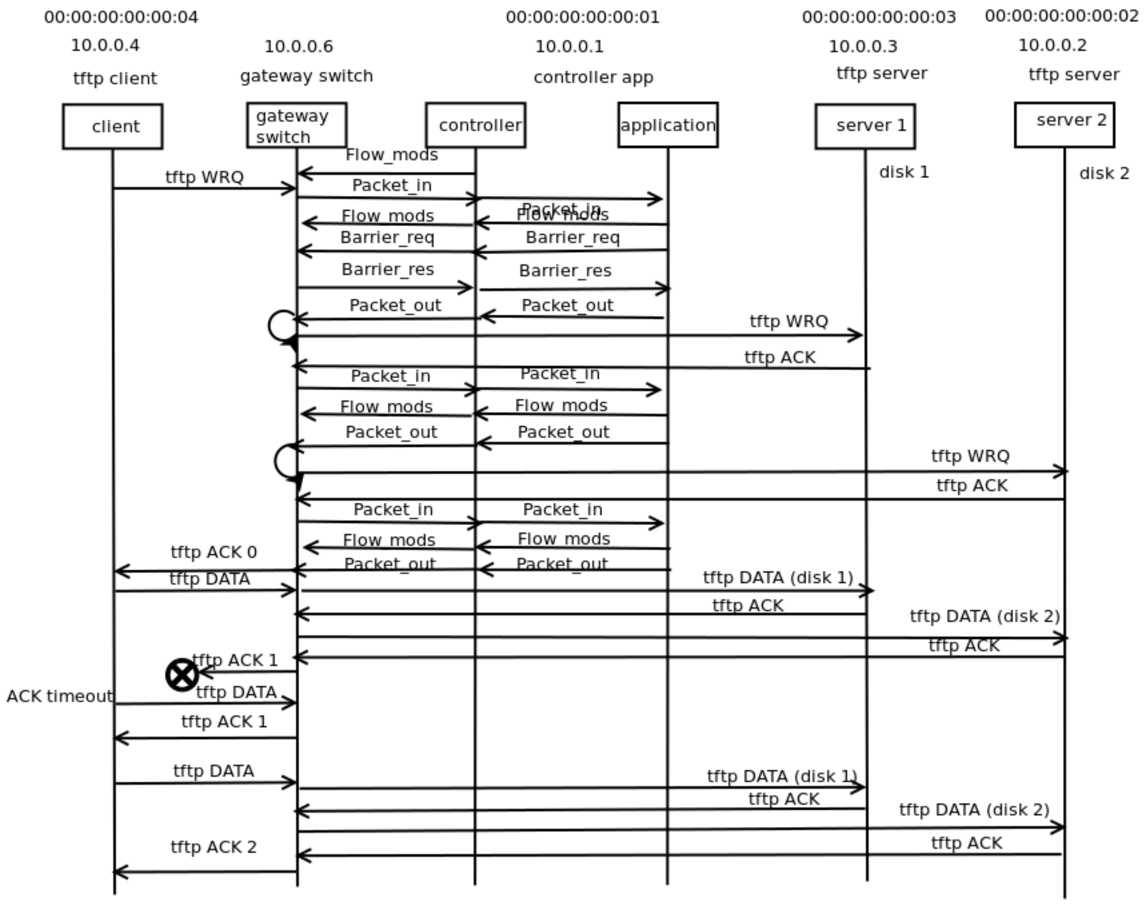


Figure 2.11: Sequential Diagram of Network Actions during Write Request for Case with Coding at Switch with Packet Loss.

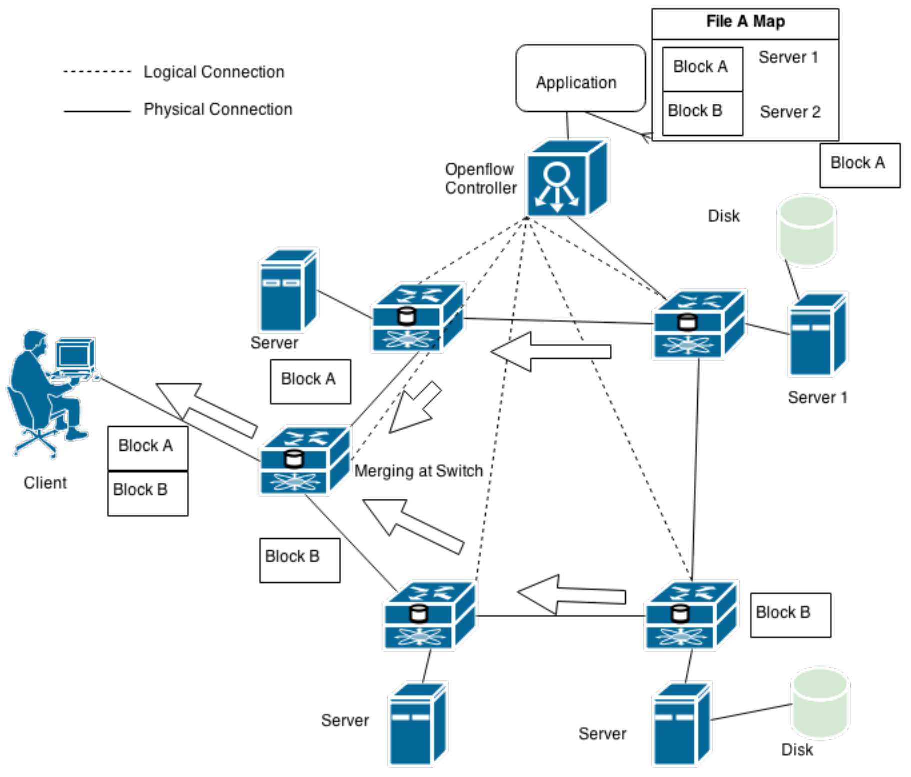


Figure 2.12: High Level View of Network for Striping with Coding at Switch.

easiest proactive way would be to periodically read a 1 block file (¡ 512 bytes) from all the disks and detect failures from ones which do not respond. Once failure is detected, the data lost in that disk could be reconstructed at a spare disk, using the data stored in the application and through coding at the switch.

A simple transaction, involving multiple internal servers to service one external client, with all the assumptions of additional switch functionality and controller application functions is shown in high level in Figure [2.12]. This design seems to meet all criteria required for a successful implementation of a TFTP based storage network.

3. IMPLEMENTATION

3.1 Challenges

There are a number of challenges involved in practical implementation of coding and striping described in this work. We need to make sure that the switch can classify on the TFTP protocol and certain fields present in the protocol. Additionally, we require some way of storing packets at the switch and releasing them when certain conditions are met. With the current techniques available in OpenFlow, the implementation of these steps is not possible without some extension or modification.

On closer examination , we find that new match and action messages are required. We also need to define a custom datastructure, located at the switch and accessible from either the *Group* or *Flow* tables, in order to be of practical use. Also, we need a custom controller application to use these changes successfully. Thus, the requirements for a practical working model can be divided into the broad categories of switch dataplane modification, OpenFlow extensions and custom controller application.

3.2 Dataplane Modification

The Openflow dataplane comsists of *Flowtables*, *Flows*, *Matches*, *Actions*, *Ports* and *Datapath*. Generally, these structures suffice for implementations. However, to implement the design we arrived upon, we will need special functionality. Specifically, we require the ability to store packets at the switch, and release them once their counterpart packet from the corresponding server in the group arrives. Thus, we introduce a custom datastructure, the *Parking Lot*, which we use to manage the packets. This datastructure fits into the dataplane, as shown in Figure 3.1.

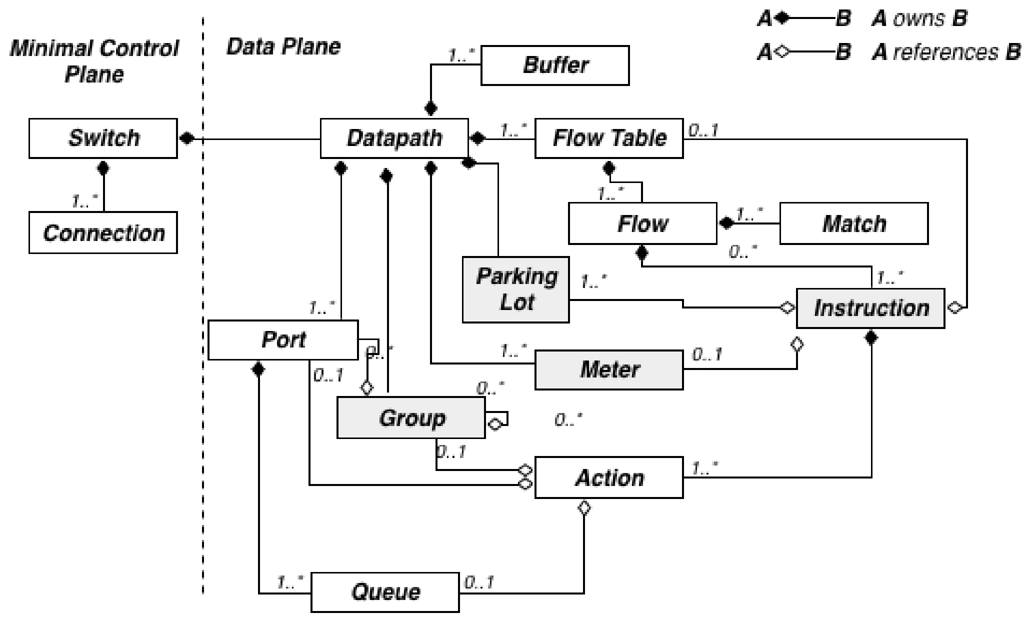


Figure 3.1: UML Diagram of OpenFlow Data Plane, with Additions for Coding at Switch.

3.3 Match Extensions

The match structure is used by OpenFlow to refer to a entry in a flow table. Match supports various extensions and is known as OpenFlow Extensible Match (OXM). OXMs allow us to extend the protocol to support further matches as required by the implementer. We exploit the fact that TFTP requests are always initiated on port 69 to classify the flow as being TFTP in the flowtable. The reply to this request is *PacketIn* to the controller, as it is not present in the flow table. After analysis of the contents of this packet, the controller can then mark this as a response to the previous initiator request. For subsequent exchanges in the transaction, the controller and data plane would require further details specific to the protocol. Specifically, we need to extract *Opcode*, *BlockId* and *Filename* present in the packets. As a result, we need to create match extensions for each of these fields.

Match extensions are not supported by all versions of OpenFlow. A comparison is given in Table 3.1.

3.4 Action Extensions

In addition to matches, we need additional extensions for action. This is because TFTP tracks block ids for each transaction. The next exchange in the transaction is thus dependent on the block id. Hence, to streamline the transaction, it is required that blockids be set explicitly by the switch where the coding/striping takes place. So, capability to set this property using action extensions is required. Action extensions are supported by versions of OpenFlow given in the Table 3.1.

Table 3.1: Support for Match Extensions in OpenFlow Versions.

Experimenter	v1.0	v1.1	v1.2	v1.3.0
Match Experimenter	-	-	Yes	Yes
Action Experimenter	-	Yes	Yes	Yes

3.5 Algorithms

Having decided on the switch modifications necessary, it is possible to come up with an algorithm which makes use of them to give us the desired network behavior. The algorthm is divided, to make it convenient to follow the different behaviors at the switch, for different cases.

Procedure ”packetDecision” in Algorithm 1 is relevant for the initial decision at the switch. This algorithm forwards the packet to the relevant flow table for further processing. The following procedures provide further actions for the packet.

Algorithm 1 Switch packet decisions.

```
procedure packetDecision(packet)

    if in_port is outsidenetworkinterface then
      outsidepacketprocedure(packet);
    else      insidepacketprocedure(packet);
    end if
end procedure
```

Algorithm 2 refers to the case when the packet originates from outside the storage network and details the actions required to handle this.

Algorithm 2 Outside packet decisions at switch.

```
procedure OUTSIDEPACKETPROCEDURE(packet)
    if packet is RRQ then
      packet_in;
    else if packet is ACK then


        extract ACK blockid;
        next_blockid=curr_blockid+2;
        send ACK to corresponding internal server;


    else if packet is WRQ then
      packet_in;
    else if packet is DATA then
        extract DATA curr_blockid;
        next_blockid=curr_blockid+2;
        send DATA to corresponding internal server;


    end if
end procedure
```

Algorithm 3 refers to the case when the packet originates from inside the storage network and details the actions required to handle this.

Algorithm 3 Inside packet decisions at switch.

```
procedure INSIDEPACKETPROCEDURE(packet)
    if packet is ACK then
        if curr_blockid==0 then
            packet_in;
        else if curr_blockid!=0 then
            ackprocedure(groupid,curr_blockid);
        end if
    else if packet is ERR then
        check flow id: empty ack queue of server; send error to external server;
    else if packet is DATA then
        if curr_blockid==1 then
            packet_in;
        else if curr_blockid!=1 then
            dataprocedure(groupid,curr_blockid);
        end if
    end if
end procedure
```

Algorithm 4 refers to the case when the packet is an ACK originating from inside the storage network and details the actions required to handle this.

Algorithm 4 Switch packet decisions for TFTP ACK packets

```
procedure ACKPROCEDURE(curr_blockid,groupid)
    if curr_blockid!=last_blockid+1 then
      queue.enqueue(groupid,packet);

    else if curr_blockid==last_blockid+1 then
      queue.dequeue(groupid);
      send all dequeued packets;
      last_blockid=last_blockid+2;

    end if
end procedure
```

Algorithm 5 refers to the case when the packet is a DATA originating from inside the storage network and details the actions required to handle this.

Algorithm 5 Switch packet decisions for TFTP DATA packets

```
procedure DATAPROCEDURE(curr_blockid,groupid)
    if curr_blockid!=last_blockid+1 then
      queue.enqueue(groupid,packet);

    else if curr_blockid==last_blockid+1 then
      queue.dequeue(groupid);
      send all dequeued packets;
      last_blockid=last_blockid+2;

    end if
end procedure
```

The network behavior, cannot be made possible without custom behavior at the controller. This is achieved by an application program at the controller, which works in tandem with the rest of the sytem to ensure successful implementation. The algorithm for that is also described in Algorithm 6.

Algorithm 6 Controller Application

```
procedure CONTROLLERAPP(packet)
Map{FileName,Servers} fileMap;
Map{ExternalServer, LinkedList{InternalServer}} connMap;
Map{InternalServers,ExternalServer} ConnCheckMap;

    if packet_in packet==RRQ then
      servers=fileMap.get(filename);
      ConnCheckMap.add(server,externalserver);
      packet_out to servers

    else if packet_in==ACK then
      externalserver=ConnCheckMap.get(server);
      connMap.get(externalserver).add(server);
      flow_mod to add internal servers
      group_mod to create group of flows
      packet_out

    else if packet_in packet==WRQ then
      servers=fileMap.get(filename);
      ConnCheckMap.add(server,externalserver);
      packet_out to servers

    end if
end procedure
```

4. EXPERIMENTAL SETUP

In order to evaluate the efficacy of our approach to the design, we try to quantify the advantages offered. The chief advantage of our design lies in the fact that the switch is capable of coding, resulting in greater efficiency in usage of network infrastructure.

Figure 4.1 shows the experimental setup, which is a more realistic representation of a storage network, as compared to the topology described previously. The topology consists of data servers, in a ring formation with attached storage disks. The network employs (n, k)-Regenerative Coding, causing the data to be distributed among n servers in the network, with any k servers' data being sufficient to reconstruct a failed disk. In the figure, server 0 has to reconstruct the data onto the free disk using data from k disks.

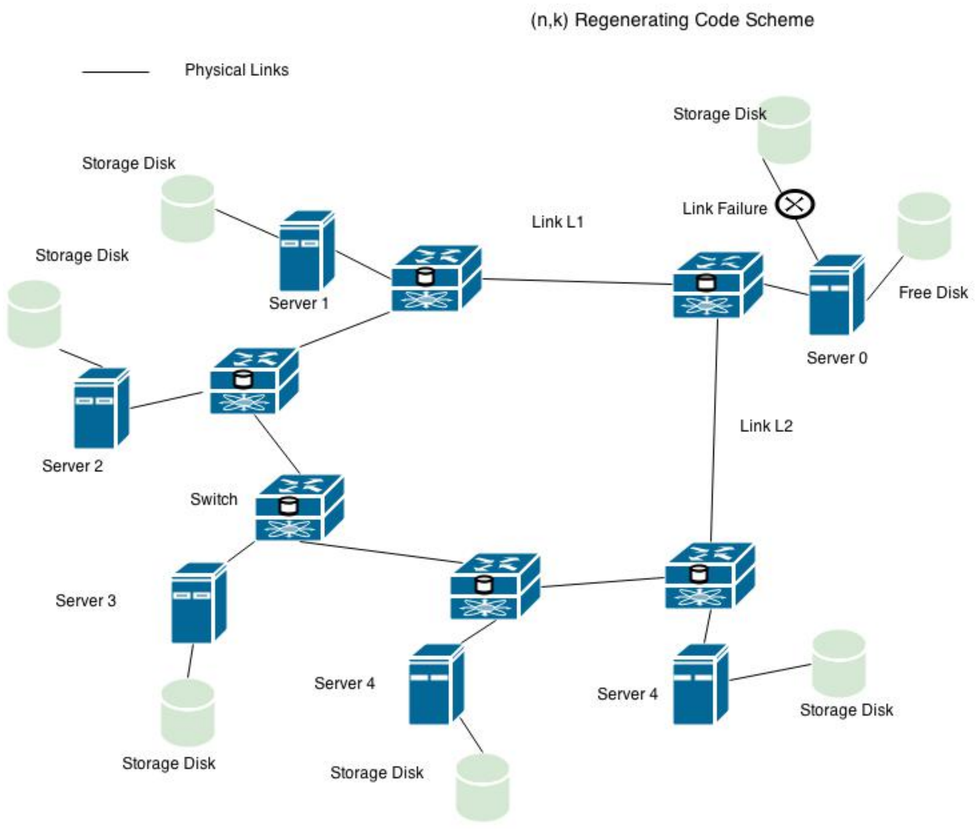


Figure 4.1: Experimental Setup showing Example Topology for (n,k) Regenerating Code.

5. THEORETICAL CALCULATIONS

Our design gives us advantages with regard to total messages required to be exchanged to set up connections and consequently, total round trip time calculations. We compare it to a standard TFTP proxy network, shown in Figure 5.1.

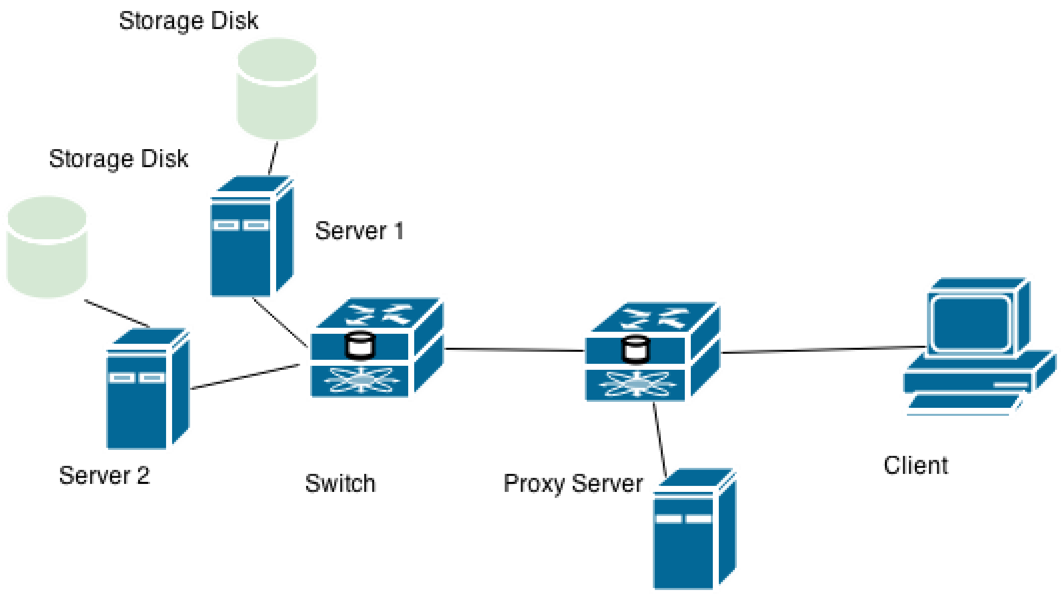


Figure 5.1: TFTP Proxy.

On comparison, with this, we find our design has advantages in total round trip times, as shown in Table 5.1.

Table 5.1: Comparison of Time Taken for Operations

Operation	Baseline(TFTP Proxy Network)	Coding on Switch
Read	3*RTT	2*RTT
Write	6*RTT	4*RTT

Thus, our design has definite advantages in reducing total RTTs required for a transaction.

6. SIMULATION

We try to quantify the advantages caused by switch coding, over standard rebuilding at th end nodes. We set up a simulation in Network Simulator NS2. The topology is as shown in Figure 4.1. We consider the disk to be rebuilt using data from other nodes, which have a constant bitrate UDP traffic of 1 Gigabyte per second. We simulate background noise in the link, which follows random Poisson distribution having a mean of 100 MBps and burst time 500 milliseconds, followed by an idle time of 100 milliseconds. The link capacity is varied and measurements are taken.

Our design also offers us advantages in network traffic usage, by coding at switches. Considering a network where (5,3) MDS coding is used, we can compare time taken to transfer for data needed for rebuilding, as shown in Figure 6.1.

For larger values of k, the performance of our design improves. For a case, where rebuilding requires data from 4 disks, the comparison of transfer times is as shown in Figure 6.2.

For a case, where rebuilding requires data from 5 disks, the comparison of transfer times is as shown in Figure 6.3.

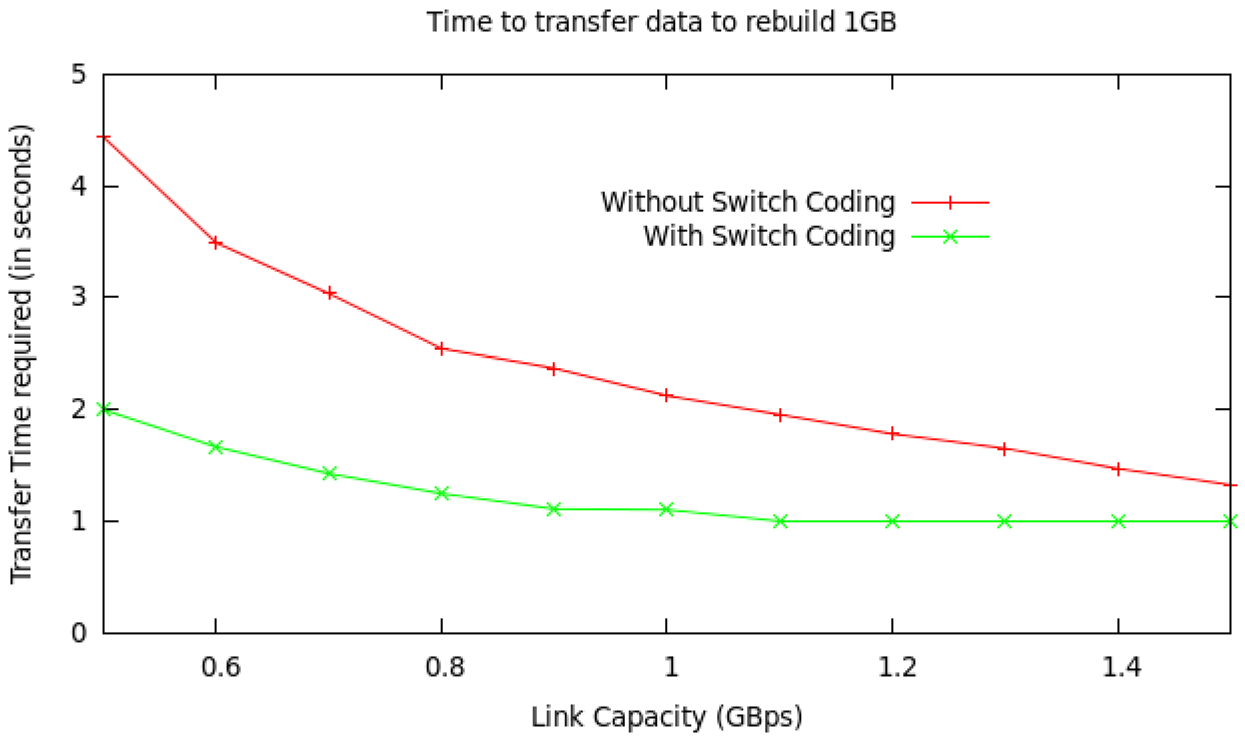


Figure 6.1: Transfer Time for Rebuilding 1GB vs Link Capacity for Rebuilding from 3 Disks.

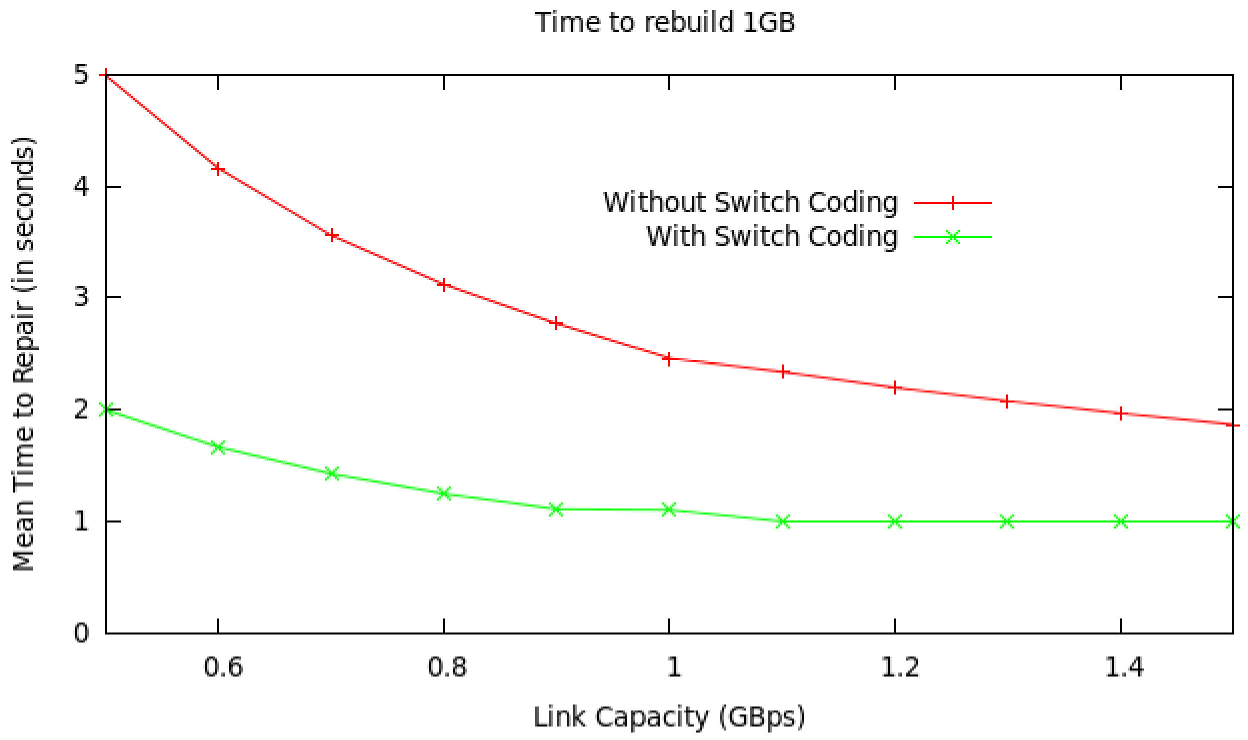


Figure 6.2: Transfer Time for Rebuilding 1GB vs Link Capacity for Rebuilding from 4 Disks.

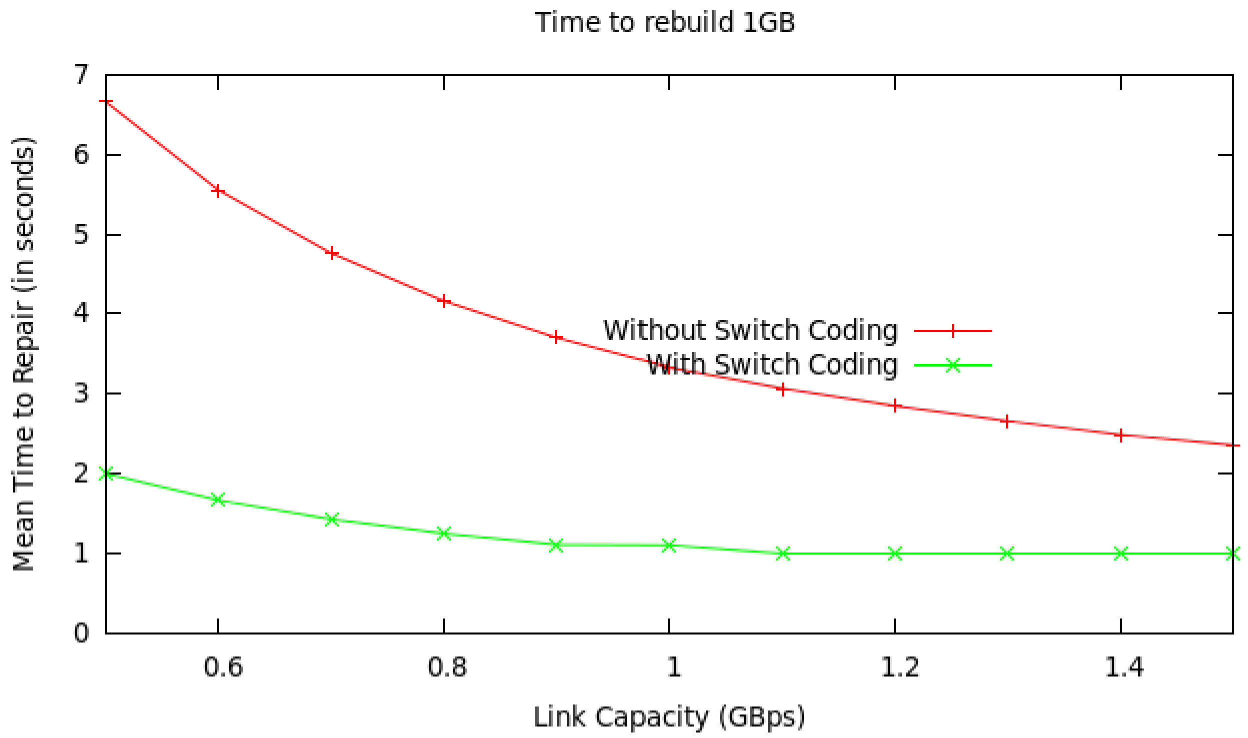


Figure 6.3: Transfer Time for Rebuilding 1GB vs Link Capacity for Rebuilding from 5 Disks.

7. CONCLUSION AND FUTURE WORK

StorageFlow can serve as a gateway for future innovations in storage networks. The work can be extended to use more specific SDN implementations, support more widely used protocols for file transfer and to implement more complex coding schemes by extending the techniques developed here. This also serves as a blueprint for developing future applications in OpenFlow. The advantages gained from this design includes support for additional functions on networking devices, traffic reduction and easy programmability of system to support experimentation, with clear demonstrated benefits.

The work can be extended in many directions. It would be possible to have a similar analysis of performance, after implementation on a commercial OpenFlow system, using CPqD SoftSwitch for OpenFlow 1.3 [5] and Ryu controller [4]. A large-scale distributed network such as GENI [1] may also be used to implement and study the design. The same procedures can also be repeated on other network protocols. Traffic in more realistic scenarios can help study the impact of design choices on performance in various storage applications.

Another possible direction of study is the theoretical analysis of benefits and drawbacks of various coding schemes for the network with switch coding capability, similar to the study undertaken in [11], but with greater focus on practical topologies. Given that most studies do not consider cases with intermediate coding at networks, this is a wide area of study and should yield promising results.

REFERENCES

[1] Geni - Global Environment for Network Innovations. Retrieved June 11, 2014 from `https://www.geni.net`.

[2] Open Flow messages description. `http://flowgrammable.org/sdn/openflow/`. Accessed: 2014-05-07.

[3] Openflow Switch Specification - Version 1.3.0 (Wire Protocol 0x04). Retrieved April 28, 2014 from `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf`.

[4] Ryu - SDN Framework. Retrieved June 11, 2014 from `http://osrg.github.io/ryu`.

[5] Soft Switch. Retrieved June 11, 2014 from `https://github.com/CPqD/ofsoftswitch13`.

[6] The TFTP Protocol (revision 2) - Request for Comments. Retrieved April 28, 2014 from `http://www.ietf.org/rfc/rfc1350.txt`.

[7] Alexandros G Dimakis, Kannan Ramchandran, Yunnan Wu, and Changho Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489, 2011.

[8] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[9] Felicián Németh, Ádám Stipkovits, Balázs Sonkoly, and András Gulyás. Towards smartflow: Case studies on enhanced programmable forwarding in openflow switches. *ACM SIGCOMM Computer Communication Review*, 42(4):85–86, 2012.

[10] B Nunes, Marc Mendonca, X Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. 2014.

[11] Dimitris S Papailiopoulos and Alexandros G Dimakis. Locally repairable codes. In *information theory proceedings (ISIT), 2012 IEEE international symposium on*, pages 2771–2775. IEEE, 2012.