SDN ENABLED NETWORK EFFICIENT DATA REGENERATION FOR DISTRIBUTED STORAGE SYSTEMS

A Thesis

by

SUJOY SAHA

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

Chair of Committee,	Alex Sprintson
Committee Members,	Paul Gratz
	Radu Stoleru
Head of Department,	Miroslav M. Begovic

May 2017

Major Subject: Computer Engineering

ABSTRACT

Distributed Storage Systems (DSSs) have seen increasing levels of deployment in data centers and in cloud storage networks. DSS provides efficient and cost-effective ways to store large amount of data. To ensure reliability and resilience to failures, DSS employ mirroring and coding schemes at the block and file level. While mirroring techniques provide an efficient way to recover lost data, they do not utilize disk space efficiently, resulting in large overheads in terms of data storage. Coding techniques on the other hand provide a better way to recover data as they reduce the amount of storage space required for data recovery purposes. However, the current recovery process for coded data is not efficient due to the need to transfer large amounts of data to regenerate the data lost as a result of a failure. This results in significant delays and excessive network traffic resulting in a major performance bottleneck.

In this thesis, we propose a new architecture for efficient data regeneration in distribution storage systems. A key idea of our architecture is to enable network switches to perform network coding operations, i.e., combine packets they receive over incoming links and forward the resulting packet towards the destination and do this in a principled manner. Another key element of our framework is a transport-layer *reverse multicast* protocol that takes advantage of network coding to minimize the rebuild time required to transmit the data by allowing more efficient utilization of network bandwidth.

The new architecture is supported using the principles of Software Defined Networking (SDN) and making extensions where required in a principled manner. To enable the switches to perform network coding operations, we propose an extension of packet processing pipeline in the dataplane of a software switch. Our testbed experiments show that the proposed architecture results in modest performance gains.

ACKNOWLEDGMENTS

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 am very grateful for the help I received from my colleagues from Flowgrammable, Atin Ruia, Muxi Yan, and Prithviraj Shome as well as Swanand Kadhe and Corey Morrison for this work. Finally, I want to thank Dr. Paul Gratz and Dr. Radu Stoleru for having served as my committee members. Apart from this I would like to thank the commentators on stack overflow who helped me out. Last but definitely not the least, I am extremely grateful to my family and friends for their constant support.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Associate Professor Alex Sprintson and Associate Professor Paul Gratz of the Department of Electrical and Computer Engineering and Associate Professor Radu Stoleru of the Department of Computer Science.

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

Funding Sources

Graduate study was supported by the ECE Graduate Student Departmental Scholarship from Texas A&M University.

NOMENCLATURE

DSS	Distributed Storage System
OF	Openflow
SAN	Storage area Network
LUN	Logical Unit Number
SDN	Software Defined Network

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION AND LITERATURE REVIEW

1.1 Introduction

Distributed Storage Systems (DSSs) have become the industry standard architecture for storage of digital data over the past few decades. They have many appealing qualities such as inherent resiliency to data-loss, self recovery, ease of access and transparency to user. They mainly comprises of digital data in form of files or blocks stored over a large number of devices inter-connected over the local or wide area network. DSSs employ various techniques to prevent data-loss using techniques ranging from simply maintaining copies of data to complex storage-efficient techniques of coding.

However, when DSSs stores digital data with various coding techniques, the recovery process for this coded data is not efficient. This is due to the need to transfer large amounts of data to regenerate the loses during a failure and may be undesirable in the future, given the ever increasing repair times for the average physical hardware disk drive responsible for the actual data storage. This results in significant delays and excessive network traffic resulting in a significant performance bottleneck. This is compounded by the fact that during repair time, additional disks may fail causing in widespread loss of data, hence there is a clear need to find ways in which we:

1. Reduce time taken for data transfer for the repair.

2. Reduce the quantity of data to be transferred.

For achieving some of the above goals, in this thesis, we propose improvements to the existing network operations of the digital data recovery/regeneration techniques within Distributed Storage (DSS). The key idea of our framework is to enable network switches to perform network coding operations, i.e., combine packets they receive over incoming

links and forward the resulting packet towards the destination. This will reduce the overall time taken to transfer the data and improve overall throughput for data transmission. We also propose a transport-layer *reverse multicast* protocol that takes advantage of network coding to minimize the rebuild time required to transmit the data by allowing more efficient utilization of network bandwidth.

The new architecture is implemented using the principles of Software Defined Networking (SDN) [1]. To enable the switches to perform network coding operations, we propose an extension of packet processing pipeline in the dataplane of a software switch. Our extension uses a principled approach and requires minimum changes to the existing SDN frameworks.

In the following sections we outline various technologies we used or studied for this thesis as well as explanations into our overall contributions to the field of DSS and how our techniques can improve the overall digital data recovery process in operational distributed storage systems.

1.2 Distributed Storage Systems (DSS)

Storage network architectures are mainly composed of inexpensive commodity devices inter-connected over a local/wide area network. These architectures come in various sizes and scales depending on operation. We commonly see various small-sized installations in a small business or research facility to major datacenters (with hundreds of millions of physical devices all linked over the network) owned by public cloud providers such as Amazon Web Services, Google Cloud Platform and Microsoft Azure which provide facilities for on-demand storage in their major data-centers across the United States. This is explained below.

A **Distributed Storage System** (DSS) is a storage architecture spread over a network. It provides accessibility, reliability, scalability and security to customers who store

data within them. While there are many such architectures with similar goals, they often have different implementations. However, they all have common features namely: Master Node, Data Node and a User Client as shown in HDFS(Hadoop Distributed File System) [2] , GFS(Google File System)[3] and BTRFS [4] etc. These features have been indicated in Figure 3.9. :

1. **Master Node** : The master node is a global observer and orchestrator for storage related operations in a DSS. It maintains contact with individual Data Nodes and transmits commands based on its interaction with a User Client. It also runs various processes and jobs which track the overall health of the DSS. In cases where there is a node failure, and data would need to be rebuilt, the Master Node remotely initiates the repair process with the individual data node or coordinates with multiple data nodes for a multi-node repair process. Both read and write operations from a client are initially received by the Master Node which assigns various jobs and subprocesses for those operations, examples include NameNode in HDFS, master and shadow master in GFS etc.

2. **Data Node** : The data node is a server attached to commodity hardware. It responds to commands from Master Node and sends information on demand. Data Nodes coordinate with Master Node during user operation such as read/write and update. They do this by transmitting and receiving control messages from the Master Node. For large files/blocks of data, Data Nodes merely store smaller blocks/file components respectively while the Master Node maintains addressing information for this data for enabling user to read/write and update data across a network of devices. For repair operations, the Data Nodes follow instructions from the Master to regenerate lost data, usually by writing data to a Data Node operated "hot spare" device (empty disk or drive).

3. **User Client** : User clients (for example HDFS client) are usually applications which allow users to read/write and update data into their respective DSS. Some DSSs can allow the client to maintain a local copy which is in "sync" with the remote copy like the Google Drive client. They also have features such as allowing file sharing among different users, allow them to maintain various levels of access privileges etc. The clients usually contact the Master Node which coordinates with the Data Nodes to perform desired operations.

DSSs can be implemented with two fundamental strategies for storing data that are either in a structured file form (NAS) or in form of blocks of data of fixed size(SAN), where NAS refers to Network Attached Storage and SAN refers to Storage Area Networks. We define and illustrate the main differences between the two and are illustrated below:

1. **NAS:** NAS provides access to storage data at the file-level through various protocols at the application layer. With the help of its "head" a dedicated hardware device (Master server) with which it accesses a local or wide area network(via TCP/IP over Ethernet), The NAS server also handles client authentication and manages file-level operations through common network protocols to transfer data across multiple disks(as per user requirement) by protocols such as FTP(File Transport Protocol), NFS(network file system) SMB/CIFS etc.

    - **Advantages**: NAS is also cost effective as it can run on a commodity device without need for expensive hardware.
    - **Disadvantages**: Higher latency and lower throughput when compared to SAN.

2. **SAN:** SAN in today's datacenters connects multiple sets of storage devices(containing multiple disks) and manages sharing of low-level data (block) with each of the devices using a networked protocol like iSCSI (Internet Small Computer System In-

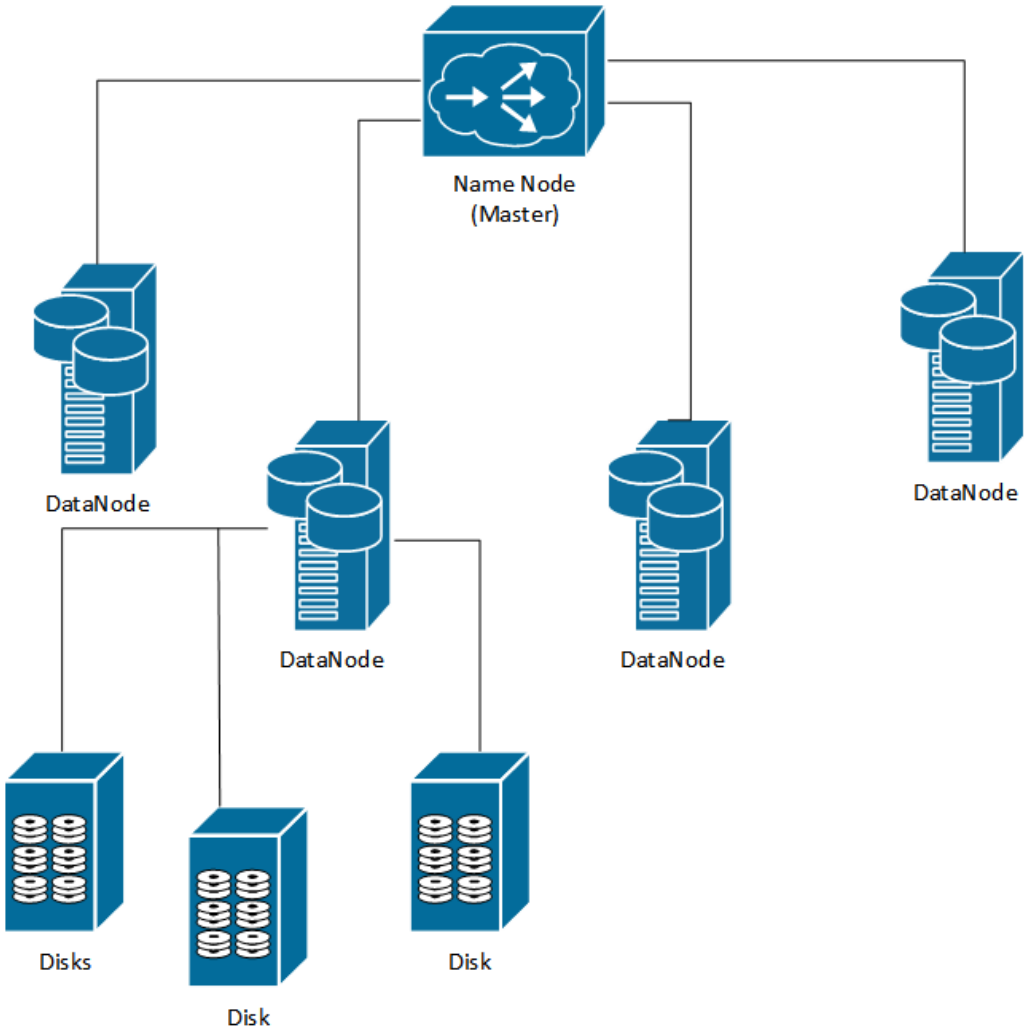

Figure 1.1: Common Distributed Storage System Hierarchy.

terface) and Infiniband or Fiber channel.

Advantages:

- SANs have low latency and higher throughputs for data transfer compared to NAS.
- SAN provides a distributed form of consolidated storage without much file system overhead.

Disadvantages:

- SAN setup is difficult and requires constant maintenance.
- SAN repair mechanisms are network-transfer inefficient and highly time consuming.

Both these approaches have repair mechanisms based upon maintaining redundant copies of data (with customization depending on file system for NAS). To ensure reliability and improving efficiency some DSSs use techniques as simple as imply mirroring data to more complex techniques such as coding.

In Mirroring, we replicate data across a multitude of disks to prevent cases of data loss. This would require more physical drives as we would always want to make sure there are enough copies to avoid loss of data. The cost of mirrored storage on these physical devices has significantly fallen over the years to a $0.2 per GB. However, the exponential increase in storage of data (upto an estimated 40,000 exabytes in 2020) has resulted in requirement of additional capacity to store this data in for of an average increase of $8 per GB, mainly because of cost of purchasing new hardware with ever increasing quantities of data. This is known as the "digital paradox". Apart from this, when we consider the fact that only 15% of the data being stored is actually important for business-critical applications, as mentioned in the Veritas Global Databerg Report [5], we see that mirroring techniques, while simple, are simply non-sustainable in the long run.

Coding techniques result in a significant reduction of storage overhead. These techniques do not need replicas of data and redundancy is maintained in form of parity or in form of some function of the storage data. In this system is that whenever a disk fails, to maintain reliability, we rebuild the parity on a new node. Here one can use erasure coding to boost resource utilization while ensuring reliability. Apart from this one can potentially implement new schemes such as regenerative codes for further optimization.

The economic need to reduce the cost of data-storage in both time and monetary ways is necessary, and thus, our work mainly is within this scope.

While our work mainly focuses on SAN based storage over network, our model can potentially be extended on to file systems which attempt to implement RAID across multiple devices over a network.

1.2.1 Distributed Storage Systems: Regeneration and Coding of Data

Some storage architectures maintain reliability by performing coding operations. A few of these additionally spread the encoded data over a network. This extra step can be considered a good strategy as this not only acts as a safeguard against internal network related issues, it also helps prevent catastrophic data loss in the wake of natural disasters such as hurricanes, tsunamis etc. For example Tokyo datacenters for major tech firms in Japan experienced a significant loss in data stored during the 2011 tsunami [6]. Apart from this, various schemes provide different balances between accessibility, reliability, storage capacity and finally overall performance and user experience. In case of erasure codes, we transform a block of data into a larger block so that we can re-generate that block from a smaller subsets of that encoded block. Now assuming that the whole data block is sub-divided into K symbols or sub-blocks, the erasure code will generate N blocks/symbols such that $N > K$. Here any K of N symbols are sufficient to regenerate the block. These types of codes are known as (n, k) Minimum Distance Separable Codes (MDS). However, currently most mainstream DSS only implement repair strategies like mirroring. A few proposed DSSs like NCCloud [7] implement techniques similar to RAID5 a *(n,n-1)* code or RAID 6 a *(n,n-2)* code. RAID [8] is the concept of redundant array of independent disks and is a data presentation technology for storing data that transparently presents to any customer or system multiple logical/physical drives as a single block or logical unit. It has both higher I/O performance for disk as well as maintains redundancy. RAID also

transparently provides fault tolerance to data stored within various RAID levels/schemes such as RAID 0, RAID 1 to RAID 5 and RAID6.

RAID levels:

- In RAID 0 we perform "striping" , where we segment out the data into multiple blocks and store them on different drives on the array.

- In RAID 1 we perform mirroring operation (copy data of one disk, write to a hot spare). This is mainly used for data duplication.

- In RAID 5 whenever there is failure of a single drive, during subsequent read operations or when we perform repair operation, we regenerate the lost data from parity stored in other drives. Here we have a single parity of the data distributed amongst the drives in form of parity blocks. From this parity we can recover from one drive failure. Here a storage manager would need to perform the work of individually downloading the data from each drive and recalculate the lost data from the parity and other drives. However, it has been observed that RAID 5 system rebuilds are susceptible to high chances of data loss especially during a rebuild as array rebuild times have grown exponentially over the years due to an explosive growth of data. During this rebuild, there is a chance of a second drive failure, thus causing the loss of data on that entire array.

- In RAID 6 we have block level striping with double parity distributed amongst multiple drives. This provides resilience against two failed drives as we can recover these. Rebuild time however is significantly more time than rebuilding a single drive as is the case in RAID 5, but with reduced chance of failure due to more fault tolerance capability. However both RAID 5 and 6 have lower reliability until the drive is replaced and repair is conducted (due to a lack of additional redundancy during the

duration when a repair operation takes place). Microsoft Azure for example uses RAID 6 [9] as it strives to ensure both reliability and storage space optimization.

Dimakis et. al. in [10] proposed simple regenerative codes which are specially designed for storage and proposed a great many number of regenerative schemes. These schemes were used to perform "exact repair" operations by performing a direct XOR of network packets and obtain high-repair rates when compared to standard erasure codes, mirroring and even Reed Solomon codes. Dimakis et.al's work specifically has significant potential in DSS where we have an urgent need to minimize the network bandwidth that goes into restoring data redundancy after failure as well as the quantity of data required to be downloaded. This improves overall system performance and it is a top priority as growing disk capacity and growing amount of user-data we have higher repair times and and growing strain on network resources. Major tech firms such as Google, Facebook etc are already looking into potential implementations to improve performance to meet an exponentially growing demand.

While in the theoretical realm, coding techniques are well studied, there are very few practically feasible implementations. In our work we look at ways and techniques at how we can improve the performance of such systems by using network coding (see section 1.3). These techniques that we propose are general enough to be implemented onto a DSS and can be further extended. As a proof-of-concept we look at ways in which we can repair RAID 5 based DSS over network.

1.2.2 Software RAID Discussion

In this thesis we use software based RAID as a setting where our proposed architecture shows promise. Currently these implementations are provided in many forms by many operating systems as:

1. A part of a file system (e.g. GPFS [11], ZFS [12], or the recent Btrfs [4]).

2. A feature in any generic logical volume manager software.

3. A layer above file system providing parity protection for user information stored in files.

4. A abstraction layer for multiple devices, consolidating into a **single virtual device** (for example, softraid from OpenBSD and **md with mdadm manager [13] on Linux Kernel**).

Currently there are some file systems available or (mostly) under development which support RAID levels to some extent, they are listed as follows:

1. Btrfs developed at the Stony Brook University is a file system for Linux with a focus on fault tolerance. It supports RAID 0, RAID 1, RAID 10. In the near future we can expect RAID 5 and 6 to be a part of this file system.

2. ZFS, a file system with logical volume manager, freely distributed by Linux, supports RAID 0, RAID 1, RAID 5 , RAID 6 and some nested RAID levels such as RAID 10 as well.

3. XFS [14] was developed as an integrated volume manager supporting operations such as striping and mirroring of various physical devices for fault tolerance.

1.3 Network Coding

In linear network coding [15], we envision nodes in the network (such as switches and routers etc), as intermediate points which receive incoming packets, perform encoding operations on these packets based on particular coding schemes and encode them together as a new unique packet and transfer them to the next intermediate point instead of simply forwarding them. Linear network coding is a method which can improve network scalability, efficiency and throughput, and potentially provide resilience against eavesdropping and packet sniffing.

In our work we we primarily focus on a simple bitwise exclusive OR operations on software switches with a modified data plane and enforce encoding strategies using Software Defined Networking(SDN [1] enabled network elements (see section 1.4).

1.4 SDN: Software Defined Networking

Software Defined Networking [1] is a recent industry led approach to general computer networking which allows abstraction of switch level routing/network services by decoupling and clearly defining the control plane and data plane. Prior to this, most network services offered by switches were *black boxes*, with proprietary hardware and software.

SDN led to a simplified model of network management with heavy focus on enterprise level operations by having a logically central controller to set and enforce network policy and determine overall network behavior such as forwarding/packet drops etc.We provide an architecture of SDN in Figure 1.2.

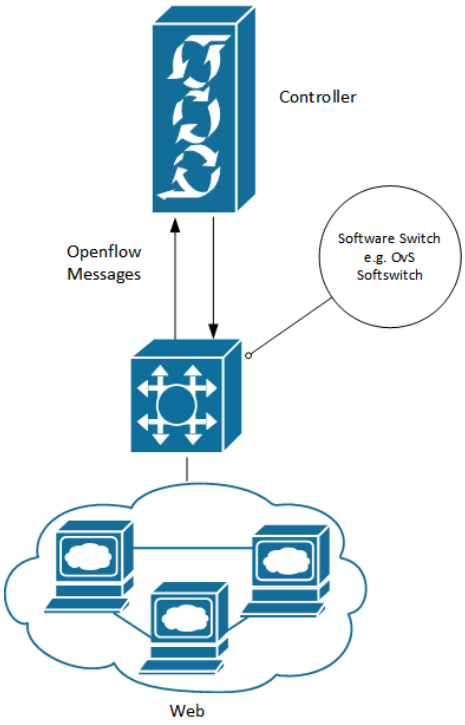

Figure 1.2: Software Defined Networks Architecture.

With the decoupling of Controller and Dataplane, SDN uses OpenFlow [1] as a protocol that operates over TCP/TLS on the application layer to communicate between controller and switch. Controllers send out messages (using the OpenFlow Protocol (OF)) to process incoming packets, configure various switches and switch states to essentially control and route traffic based on network policy of the network admin.

These controllers obtain local/global network awareness by processing incoming packets and manipulating of flow tables at the switch through many different kinds of messages and datastructures. So far there have been nine versions of OpeFlow with version 1.5 being the latest offering from the Open Networking Foundation, the regulatory body behind OpenFlow releases.

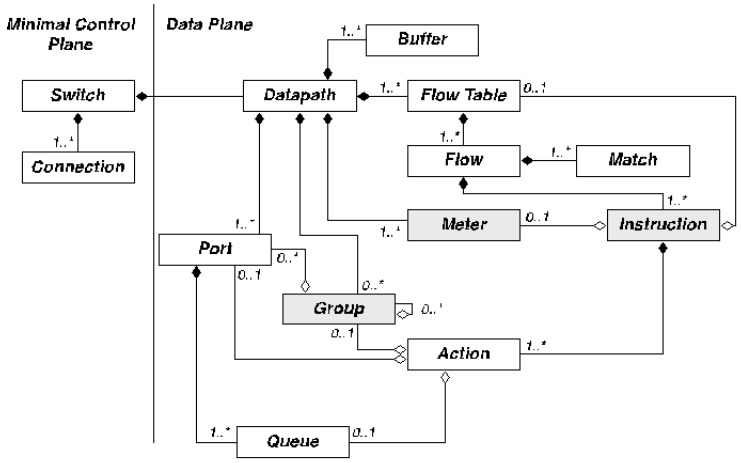

Figure 1.3: SDN Dataplane Abstraction UML Diagram.

With OpenFlow, we see industry having a largely well defined switch architecture in general with a dataplane consisting of a packet processing pipeline to process packets, and a switch target which interacts with the controller. The data plane of a switch refers to its many ports, flows, group tables, flow tables, flow classifiers, groups, instructions and actions [16]. A data model for the packet processing pipeline is presented in Figure 1.3.

Packets enter and exit from ports of a switch. Inside the switch packets undergo header extraction of various layers and then are matched to flows (an abstraction of flow-rules defined by controller) . Then we have Flow tables mapping flows to corresponding sets of controller specified actions. OpenFlow also provides extensible components starting from OpenFlow 1.1 called OXM and OXA (OpenFlow extensible match and action) which allows developers to process custom packet headers and perform user-defined routing etc.

1.4.1 SDN: Programmable Dataplane

All packets upon entering the switch undergo packet processing within a switch specific pipeline as shown in Figure 1.4. Usually, the packet will consist of various headers such as Ethernet header, IP header and TCP header and these are extracted and a key is

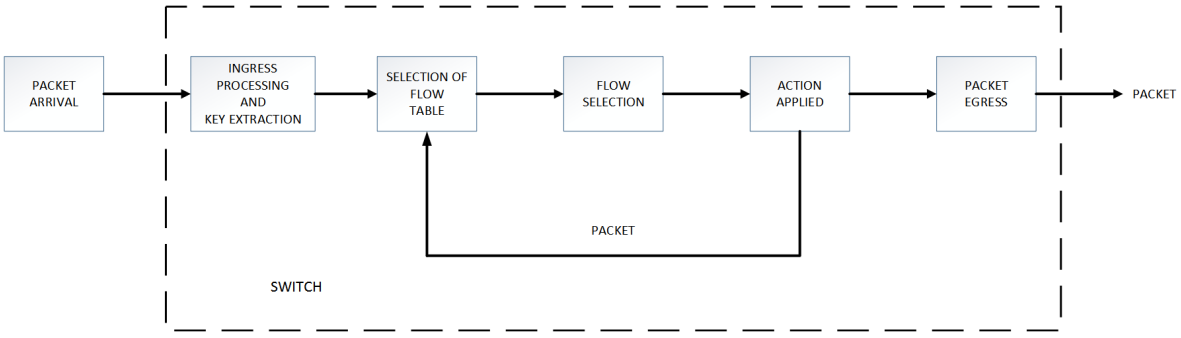

Figure 1.4: Generic Dataplane Packet Processing Pipeline.

constructed with this extracted information and packet metadata.

This key is used to select a particular flow from within the flow table. After this based on the flow-table match, we have a matched action, which can perform actions as simple as dropping the packet entirely, or forwarding to next hop with updated IP address, modify certain headers of packet, queue the packet in an output buffer, direct packet to next flow table etc.

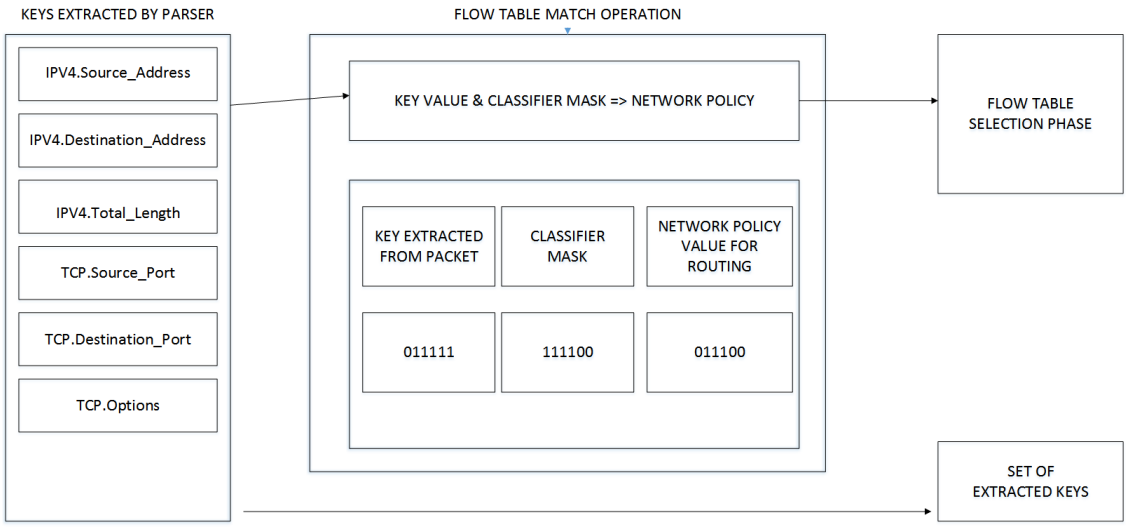

Figure 1.5: Dataplane Match Operation.

Extracted packet's openflow signatures are stored in a data structure called Match for classification purposes, as shown in Figure 1.5. Packets are generally classified into flows based on their source/destination port, MAC or source/destination IP address among other packet information. For our project, we will be extending the regular matching capabilities of the dataplane using concepts from OpenFlow Extensible Match to classify the packets of interest and apply customized actions.

OpenFlow Actions enforce controller specified policies on packets in the switch upon matching flow entries. These include inserting the packet to a particular queue, or forwarding the packet to specific output port. A multitude of policies can be applied onto a particular flow by attaching a Action vector consisting of multiple individual actions at the end of a FlowMod (flow modification) message. Actions can be layered in a stack and have dependencies on other actions as showcased in Figures 1.6 and 1.7.

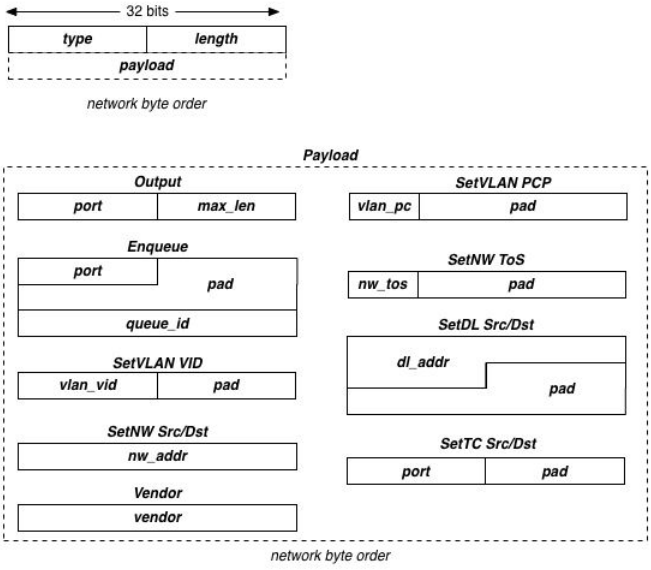

Figure 1.6: Software Defined Networks: Action Stack.

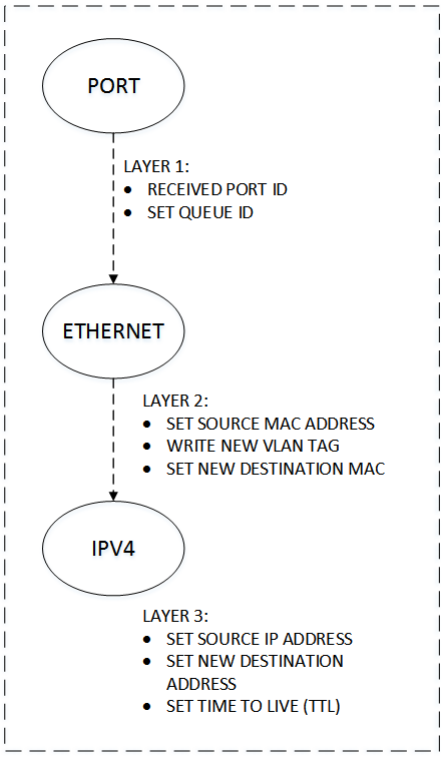

Figure 1.7: Software Defined Networks: Action Dependency and Action Types

We can see in as shown in Figure 1.8 that we use FlowMod message from OF (OpenFlow) controller to switch to modify its flow table. As explained earlier, this FlowMod consists of a Match (in our case an extended Match) for classification of flows (based on certain packet headers) and a vector of Actions (in our case a set of extended actions) that define network policies for these flows.

1.5 Related Protocols

1.5.1 ISCSI Protocol and Terminology

Internet Small Computer Systems Interface or ISCSI [17] is a storage networking protocol for linking data storage facilities where it provides users and systems access at the block level to storage devices which a reconnected over a TCP/IP network. Using iSCSI

Figure 1.8: Software Defined Networks: FlowMod.

we can send SCSI commands over the network and facilitate data transfers and manage data blocks located over long distances. It can be used for both local and wide area applications involving storage read/writes/updates. This protocol has a client and server.

1. iSCSI Client or **Initiator** : These send out SCSI commands to SCSI storage devices on remote servers (iSCSI targets). It allows organizations to consolidate storage into enterprise datacenters while providing the customers and hosts such as web or database servers the "illusion" of locally attached disks (actually spread out over the network).

2. iSCSI Server or **Target** : Any storage resource or any instance of iSCSI storage node running on a server is a potential target. Nearly all modern mainstream server OS (Linux, Windows server etc) tend to have iSCSI target functionality.

In our work, we compare our finding to that of iSCSI performance for a repairing a faulty drive with a particular logical unit number (LUN).

Note: Any LUN represents a uniquely addressed SCSI device that could even be part of the physical SCSI device according to general SCSI terminology. However in iSCSI , LUNs are used to identify disk drives and an initiator will negotiate with the iSCSI target to establish a "physical" connection to the device over the network.

1.5.2 Pragmatic Group Multicast

Multicast is a method of network addressing to deliver information to a set of destinations at the same time. Pragmatic General Multicast [18] or PGM is a well known experimental IETF multicast protocol and used for reliable data transport from one server to several clients. It uses an efficient strategy to deliver messages over each link of the network only once, while creating copies at Network routers/switches . However, like UDP multicast does not guarantee delivery of a message and may be dropped, be received out of order etc. With PGM, receivers have ability to detect out of order or lost messages and do re-ordering so that users receive an inorder data stream.

TCP uses ACKs to acknowledge receipt of packets sent and its retransmitting mechanism is also dependent on ACKs. This would be not efficient for a multicast stream, hence PGM uses Negative Acknowledgments or NACKs which is sent back to the server as a unicast message whenever there is data loss. The data lost is sent back as a Repair Data (RDATA) packet from a Designated Local Repairer (DLR) near the "loss" location or from the original source .

For our work, we use ideas proposed in this experimental protocol, and instead of using

it for multicast, we use it to perform a reverse multicast operation, as we aim to perform parity calculations over the network. Also we want to minimize packet loss hence we use NACKs for retransmitting packets. This is further explained in chapter 2.

1.6 Related Work

While new applications using SDN framework to solve age-old problems in networking has generated a lot of interest from the academia, networking community, and consequently, industry, there has been a vast multitude of research regarding its usage and further in applications. We obtain a overview of SDN by Nunes et al. [19] which discusses the newer avenues opened up by this, as well as how it compares with older technologies related to various aspects of networking. One area where there has been very little research is in the area of Storage. Mostly we found that most publications in storage domain tends to be developed for non-SDN network with a primary focus on maximizing gains within those constraints of those networks and its associated protocols.

We find pioneering work from Nemeth et al. [20] where we find potential with OpenFlow architecture and modifying it to suit our purposes for performing a principled approach to network coding on switches. However, the major limitation of that work was that it broaches the topic of coding packets together (which was not a new idea and has been referenced in 2008 [10]) with hardly any use of it in a highly specialized area of storage nor did it have any consideration for a layer four protocol based implementation which would preserve multi-hop routing information. Our work also differs from them as we modify the packet processing pipeline in SDN compatible switch. Also there has been rising interest in the field of regenerative coding for storage networks. Dimakis et al. [10] provide a comprehensive overview of this domain. To the best of our knowledge, there were no works on facilitating data regeneration through in-network coding, and this presents an opportunity for us to expand the applications of this into DSS. In our work

we attempt to design a principled approach to coding in switches by designing a special protocol for it as well as extending core components of data plane to implement coding operation switches with a focus on optimizing repair operations and possibly in the future read/write operations for DSS where data is stored in form of parity and regular blocks.

2. STORAGEFLOW: ARCHITECTURAL DESIGN

2.1 Architectural Design and Requirements

The proposed architecture of StorageFlow has the following key elements:

1. SDN-enabled switches with packet encoding capabilities for coding packets originating from different flows based on "encoding rules" established in the switch;
2. A reliable and robust transport-layer reverse-multicast protocol for data transfer;
3. A SDN controller which inserts flow-rules to switches in its underlying topology based on instructions from the *storage manager application*;
4. A *storage manager application* for setting up a reverse multicast tree.
5. Data storage nodes that store individual copies of data, keep track of the data stored on the node, and respond to read/write requests from the storage manager application. These nodes are essentially servers with local/commodity disk access capabilities.

Figure 2.1 depicts a high-level view of the proposed architecture.

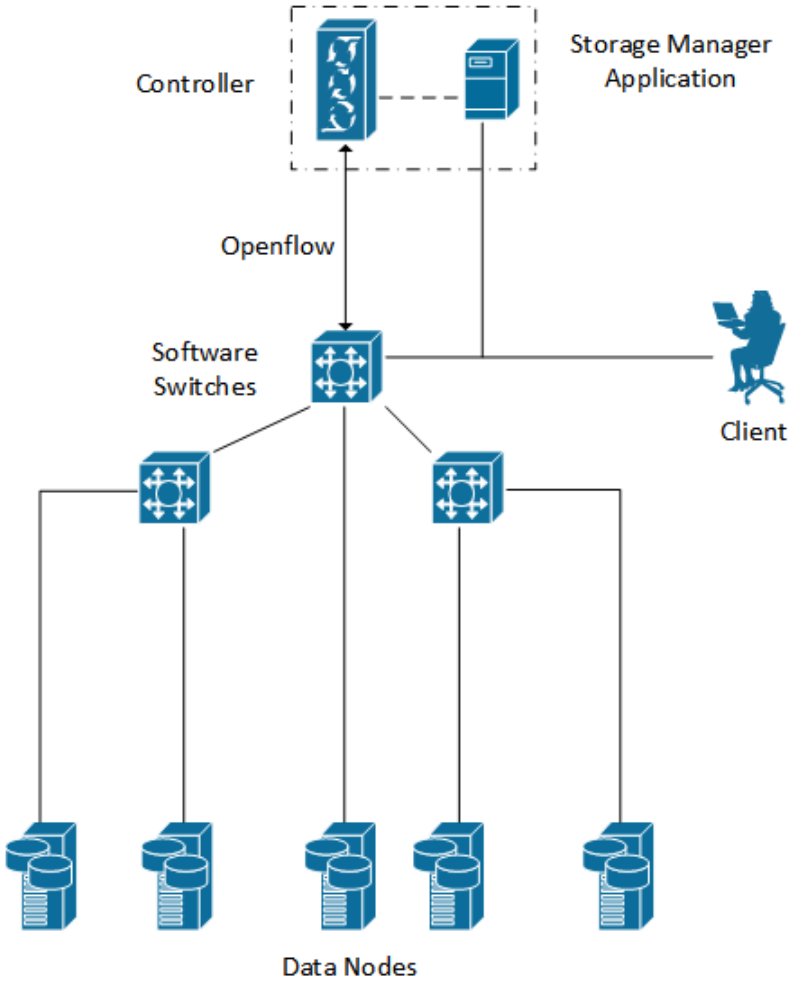

Figure 2.1: System Architecture.

2.1.1 Description Of Key Elements

1. **SDN-enabled switches with network coding capabilities.** In our work we propose a principled support for the networking coding operation through new SDN data-plane primitives. This extended dataplane will enable the switches to combine the packets received over incoming links and forward the results towards the required destination. This is made possible with the addition of a new buffering component to the standard switch data plane. This modified data plane incorporates a multi-

tiered *Parking Lot* component that will temporary store the packets of one or many flows while awaiting the arrival of a matching packet/packets required to perform the network coding operation to its full capability. More details about extending switch functionality are presented in Section 2.3.

2. **Reverse multicast protocol (RMP).** The transport-layer multicast protocol enables end-to-end reliability and congestion control across a reverse multicast tree. The RMP protocol is discussed in details in Section 2.2.

3. **SDN controller.** SDN controller maintains information about network topology. This information is passed along to the storage manager to identify the optimum reverse multicast tree. The SDN controller also detects link failures and programs intermediate switches by inserting the appropriate flow rules. The SDN controller can run additional applications such as traffic engineering and network management.

4. **Storage manager application.** This application would be an extension of the existing DSS constructs that process read/write requests from users of the DSS, keep track of where data is stored across the many data storage nodes, maintains the health of the DSS as a whole by ensuring there are sufficient copies of data, and creating and distributing data nodes as needed. An example of this existing construct in a DSS would be a NameNode in a Hadoop distributed file system. This storage manager application will be further augmented for performing repair operations at DataNodes by remotely by setting up reverse multicast tree. Additionally it would have the capability of communicating to the SDN controller for setting up parameters to set network policies enabling for network coding operations at switches.

   In our system, this application is also responsible the regenerating process, including detecting storage node failures (upon receiving a "failure" notification from the

Master), setting-up data transfer, and adding the rebuilt node to the system. When storage manager detects a data node failure it establishes the reverse multicast tree and sets up the rebuild process. To do this, the storage manager issues a unique Tree ID for each reverse multicast connection and communicates it to the source and destination nodes.

In Figure 2.1, the storage manager is implemented as an SDN application. However, the storage manager can be implemented as a separate entity that interacts with the controller through API with a communication socket.

5. **Data Storage Nodes.** This is also an extension of an existing DSS construction. In the current system data storage nodes are responsible for storing data on any local data storage devices available to the node, keeping track of what data is stored, and responding to read/write requests in a timely manner from the Storage Manager Application. In addition to this, in our system data storage nodes must also listen for Reverse Multicast requests and communicate to the storage manager when RMP requests are started and completed.

6. **Regeneration with Storage Manager**

   Whenever a data node failure is detected by the storage manager, the storage manager sets up a rebuild process. To that end, the storage manager identifies the reverse multicast tree and issues appropriate commands to network switches (through the SDN controller) to set up specific routing and coding rules. The storage manager then notifies the receiving data storage node to begin the data rebuilding process by initiating a Reverse Multicast protocol handshake between multiple servers and a single target client and the protocol then handles the data transfer. This notification is done via extensions to communication channels already inherent to the DSS, such as Remote Procedure calls for Hadoop or the Ceph API. After the rebuilding

process completes, the receiving node notifies the storage manager that all data is present and the operation is complete. The storage manager then instructs the controller to update the flow rules and thus releases the Tree ID back to the pool. It also updates its internal data storage structures with the locations and contents of the new data storage node the same way it would do so for data reconstruction in the current systems.

2.2 Reverse Multicast Protocol

To ensure reliable recovery of the data by the receiver node through in-network coding, we introduce a new end-to-end transport layer protocol, referred to as *The Reverse Multicast Protocol (RMP)* . The RM protocol is responsible for indicating to switch nodes which packets should be coded together and how the coding should take place. It is also responsible for controlling the data rate at the source nodes and retransmission of missed/dropped packets. The protocol uses the reverse multicast tree between the sources and the destination that has been set up by the storage manager, with intermediate nodes configured to perform coding operations on incoming links. In order to implement this in practice, we use an unused IP protocol number 143 to mark our Reverse Multicast packets (142 protocols numbers have been defined so far).

In unicast protocols (such as TCP) each server only services one client. Extending this principle to support a protocol with multiple sources with in-flight packet encoding is a significant challenge as it would require switches to have dedicated buffers to "park" packets from multiple sources, "combine" packets together, and then forward the resultant packets to the receiver or the next stage. The receiver would then need to send ACKs to all sources such that the next window of data is sent. Now notice the inherent challenges of the approach are as follows:

- Initial handshake between the sources and the receiver node to initiate the data trans-

fer. These handshakes are established by the Storage Manager Application or Master.

- Tagging packets with sufficient information that they can be accurately coded by intermediate switches while in flight.
- Detecting packet loss and directing packet sources to re-transmit the data if needed.
- Ensuring that all sources are transmitting at an appropriate rate to minimize the chance of packet loss at the intermediate switches.
- Terminating the connection when the information transfer is completed.

Packets of the RMP belong to one of the following seven types. Packets of type DATA are used for carrying data from the leaf nodes of the tree to the receiver. The packets exchanged by the intermediate nodes carry coded data. Packets of types ACK and NAK are used for flow control and recovery from packet loss. Packets of type SYN, ACP, ACP-ACK are used for the initial handshake. Finally, packets of type FIN are used for terminating the connection.

2.2.1 RMP Message Layer

The RMP header includes information about the source and destination ID of the packets. Each repair operation consists of a unique Tree ID. Again, these IDs are allocated to each data node by our Storage Manager Application. This information is used by the intermediate switches to identify packets that need be combined and destination IP address is used while determining the output interface for the resulting packet. Figure 2.2 depicts the structure of RMP header.

The RMP header includes the following fields:

- *RMP Tree ID* - the ID of the reverse multicast group. The ID is assigned by the

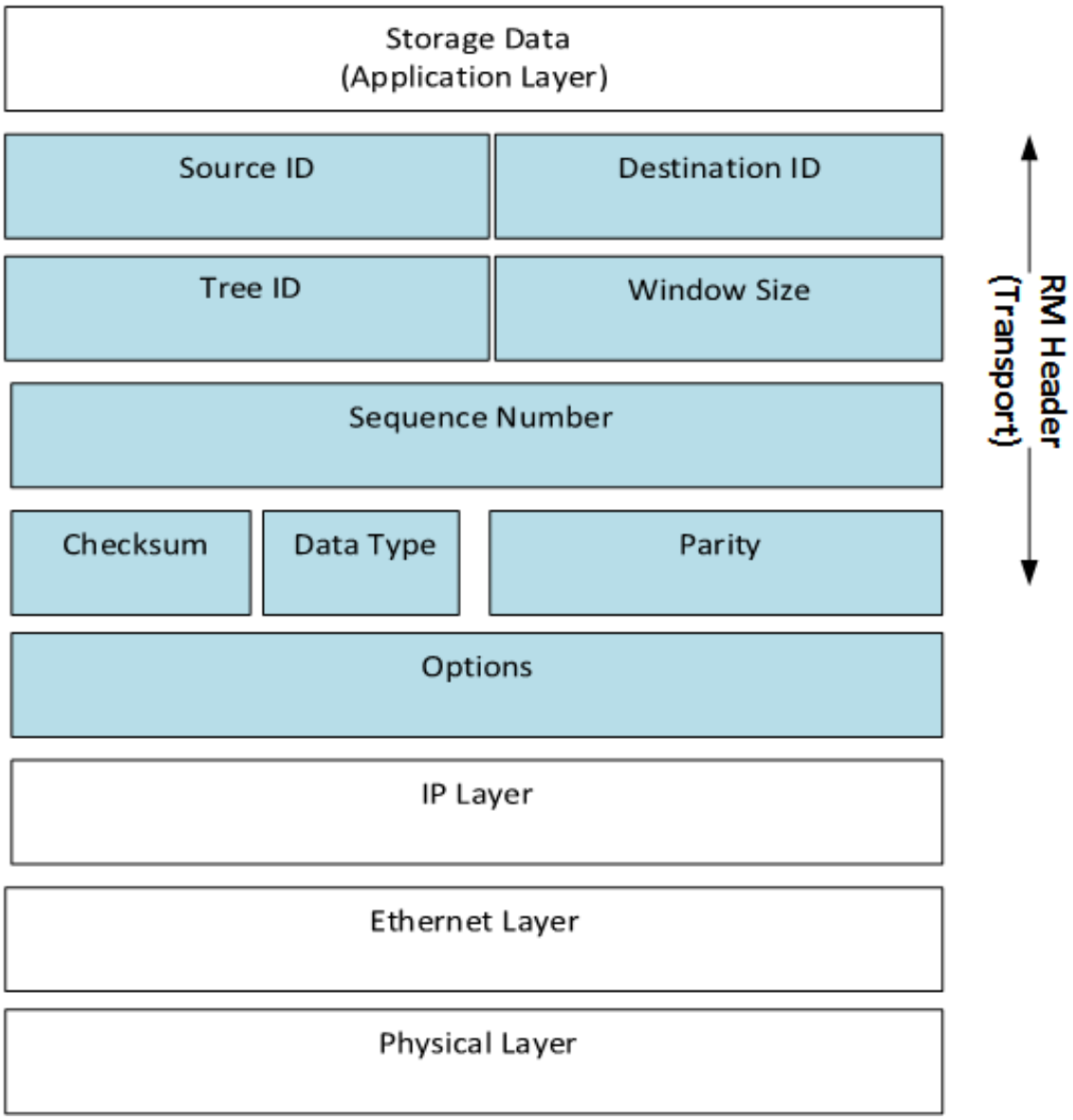

Figure 2.2: Reverse Multicast Protocol Header and Field Description.

storage manager. Since several reverse multicast connections can be active at the same time, tree ID is used for identifying the specific connection the packet belongs to. RMP Trees are assigned to specific groups of storage servers and is precomputed by the Storage Manager application.

- *RMP Storage Source ID* - the ID of the source node or intermediate network node that generated this particular packet. This unique ID is assigned to every node in the network ahead of time by the application using the RMP.

- *Data-type field* - the type of packet being transported (SYN, ACP, DATA,ACK,NAK,FIN).

- *RMP Storage Destination ID* - the ID of the receiver node (the root of the reverse multicast tree).

- *RMP Sequence Number* - the sequence number for DATA and ACK packets. The sequence numbers are used by the switches to determine which packets should be coded together. The packets combined together must have identical sequence numbers.

- *RMP Options* - an optional field for uses such as scaling(similar to TCP scaling operation), storing coding coefficients etc.

- *Window Size* - the value of the current advertised window. the value of the current advertised window. This field is used for flow control. Using the scaling factor in Options, we can have a larger effective window.

- *RMP Checksum.* - checksum of the payload. This field is used for verifying the integrity of the packet.

- *RMP Parity* - parity of the payload. This field is used for enforcing proper encoding by the switch .

The data portion of packets are encoded as well as the RMP Parity field. This is particularly useful for encoding switches with more than two ingress points as a unique parity(generated as a result of the encoding) can indicate that all the ingress sources were

fully encoded. This parity field is further used as component of flow rules to forward fully encoded packets to the next hop. The checksum feature, as can be inferred is updated at each encoding operation, at each encoding switch. The client will drop the partially coded packet as it will verify the received packet's checksum and drop it if it is not expected.

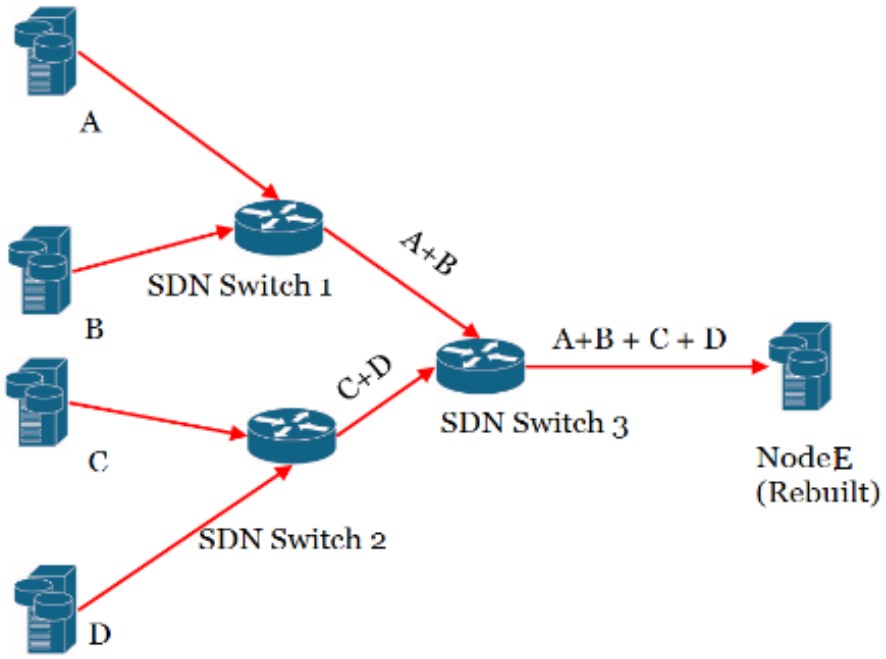

Figure 2.3: Operation of the Reverse Multicast Protocol.

2.2.2 RMP Message Sequences

Initial 3-way handshake. The receiver initiates the 3-way handshake with the source nodes at the beginning of the connection. First, the receiver sends SYN messages to target source device. Then, the sources accept the SYN message and return a ACP (accept) message. On receiving the ACP (Figure 2.7) message from the required number of sources, the client sends a ACK (accept-acknowledgment) (Figure 2.5) to all sources.

Data delivery. Following the initial handshake, the source nodes start to transmit DATA (Figure 2.4) packets. Figures 2.4 to 2.8 depict the operation of the RMP protocol while transferring packets. We describe protocol features as follows.

1. Each source sends a window $cwnd$ of packets to the receiver. This stage is very similar to TCP as the client sends out broadcast ACK to each server to send the next window of packets. By maintaining a common window for all servers/changing the window to maintain same sending rate for each server to client, we can control the overall rate of transmission. The receiver (in a "reversal" of methods employed in TCP) also serves as the single synchronization point for all data transfers and conducts flow control operations by adjusting the congestion window.

2. If a full window of DATA packets is received, with correct parity information the client sends the ACK messages back to the servers. In these ACKs, client specifies a common window size and an updated sequence-number for the next set of packets that it requires. This window size, similar to TCP is dependent on packet loss and unforeseen circumstances such as timeouts.

    - **Slow Start phase:** In RMP the Slowstart phase initially starts with a congestion window size $cwnd$ of 1 or any other base value as deemed necessary by the developer. This Congestion Window is increased by 1 for each DATA packet received by the receiver and like TCP we double the window size each round-trip time. This transmission rate is increased up-to the receiver's RM common advertised window $rwnd$, or if we detect packet loss or if we reach a predefined $ssthresh$ or slowstart threshold.

    - **Congestion Control phase**, At this stage RMP stops utilizing slowstart approach and instead uses the standard congestion avoidance in the form of a linear growth of $cwnd$. In this case our $cwnd$ is incremented by 1 per $cwnd$ of

packets received at the receiver. It follows the additive increase multiplicative decrease model, where when the receiver detects loss, it sets half of current $cwnd$ as $ssthresh$ and new $cwnd$ (same as FAST recovery in TCP), with the difference from TCP being that the receiver will use the $cwnd$ to reduce the overall window size of packets to be sent by all servers participating in the RM transfer, thus maintaining a common rate for all incoming data transfers as mentioned earlier.

- **Fast Retransmit phase(part of congestion control).** During the fast retransmit phase the RM sender receives a specified number (3 in this case, like in TCP) of no-acknowledgments (NACKs) with the same sequence number, the sender will infer that segment was dropped. The sender will then retransmit the packet with updated $cwnd$.

3. The receiver and intermediate nodes verify the integrity of each packet by comparing the expected parity (encoding parity) and checksum value (verify content integrity). When the number of received packets equals the window size, the receiver sends a broadcast ACK message to all sources. This process is repeated until the receiver obtains the last encoded packet and transmits a FIN (finish) (Figure 2.8) packet to all the servers.

4. The senders send a FIN message for the final packet and wait for a FIN message from the receiver acknowledging its receipt. Finally, the receiver notifies the Storage Manager that the local node has been regenerated.

2.2.3 Recovery From Packet Losses

Modern datacenters are capable of running lossless Ethernet networks using priority flow control to minimize packet loss. In addition, flow rules (discussed in Section 2.3) will

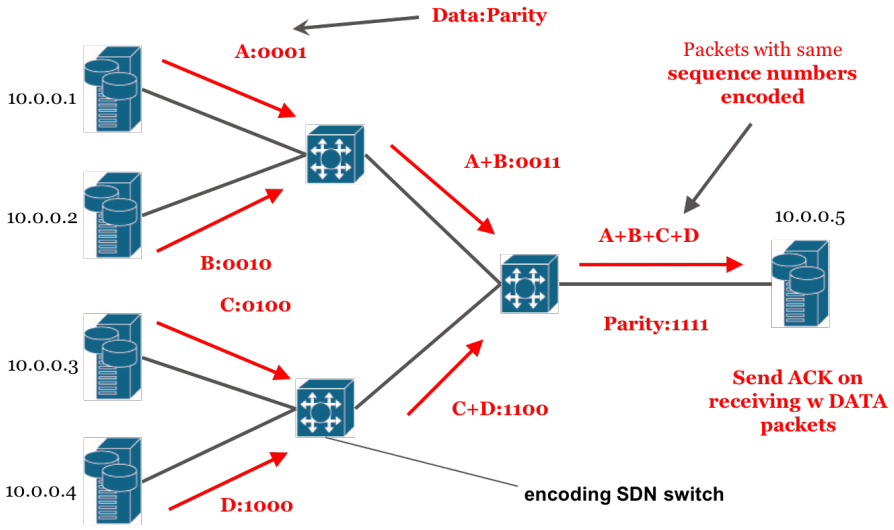

Figure 2.4: Reverse Multicast Protocol: DATA (Application Data) Message.

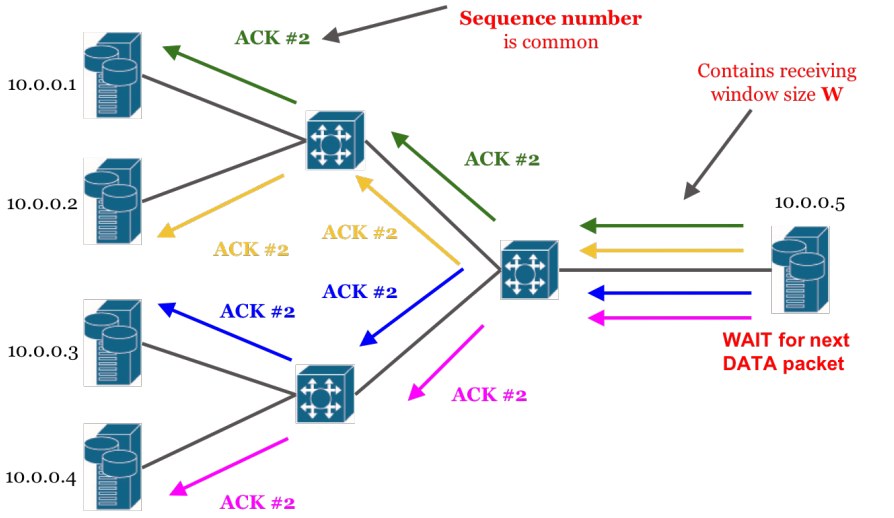

Figure 2.5: Reverse Multicast Protocol: Broadcast ACK (Acknowledgement) Message.

be written to ensure that intermediate switch nodes will not forward packets to the next destination in the network until they are fully encoded.

Since the RMP depends on multiple sources supplying packets for encoding, loss of

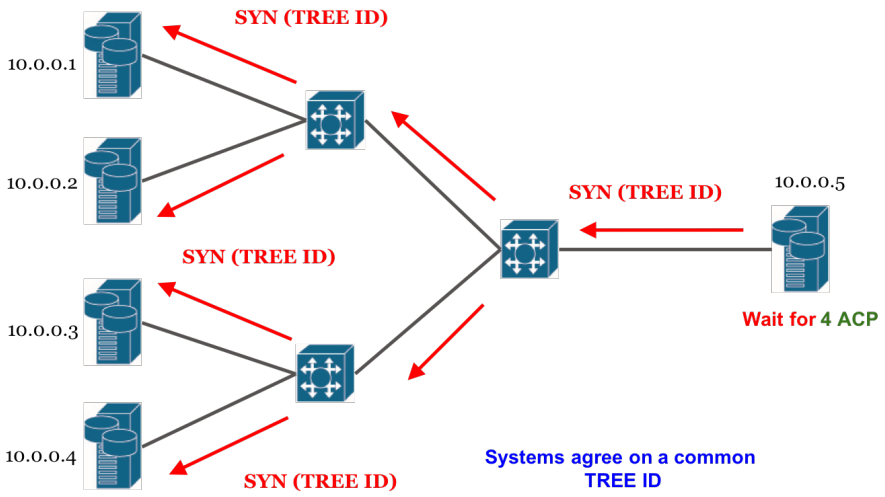

Figure 2.6: Reverse Multicast Protocol: Broadcast SYN (Synchronous) Message.

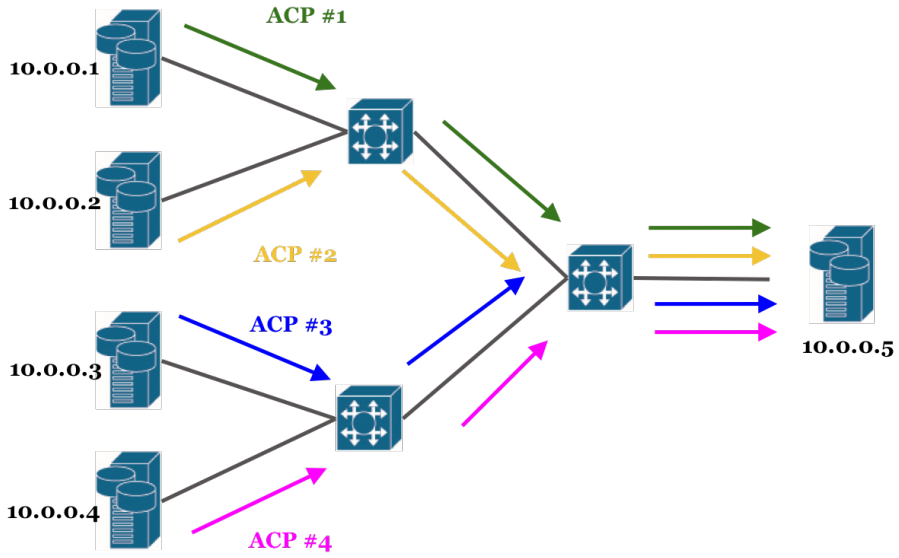

Figure 2.7: Reverse Multicast Protocol: ACP (Accept) Message.

any packet is possible at any link. When the receiver discovers a lost packet (by its sequence number in a set of packets) it sends a NAK message to all sources nodes. This message contains lost packet's destination ID, sequence number, and Tree ID. This NAK

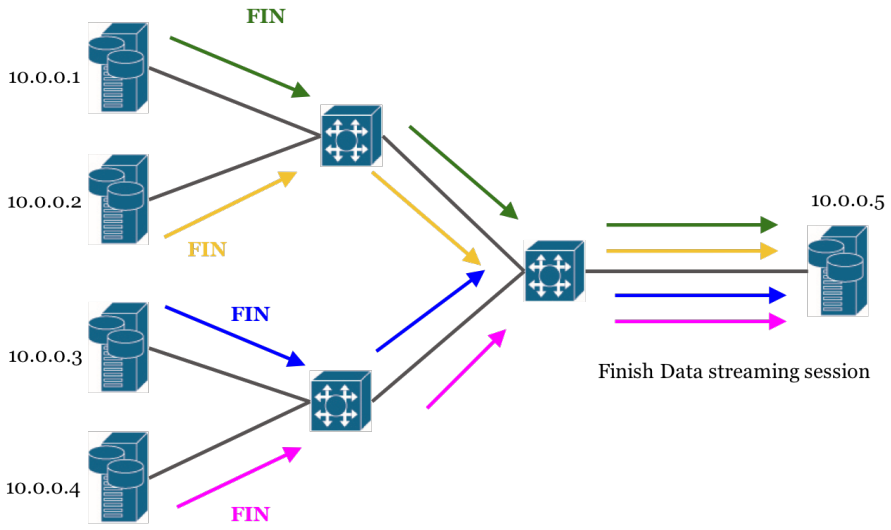

Figure 2.8: Reverse Multicast Protocol: FIN (Final) Message.

will prompt source nodes to resend the packet corresponding to the lost sequence number. This process is illustrated in Figure 2.9. To control congestion, the receiver also reduces the advertised window size after sending multiple NAKs. This will result in reducing the rate of overall data transfer and minimizing the probability of further packet loss at intermediate switches. NAK is identical to ACK except for sequence number identifying the exact packet to be retransmitted.

2.3 Dataplane Support For Coding Operations

To support the encoding operations at the intermediate switches, there is a need to extend the packet processing pipeline. The typical pipeline of an SDN-enabled switch includes several stages (see Figure 2.13). The first stage handles packet arrivals and initial buffering. This is followed by the *Field Extraction* stage, in which several key fields such as IP address and port numbers are extracted from the packet header and stored as packet metadata. These extracted fields are used for the *table selection* and *flow selection* stages. SDN-enabled switches contain multiple flow tables. Each entry in table defines packet

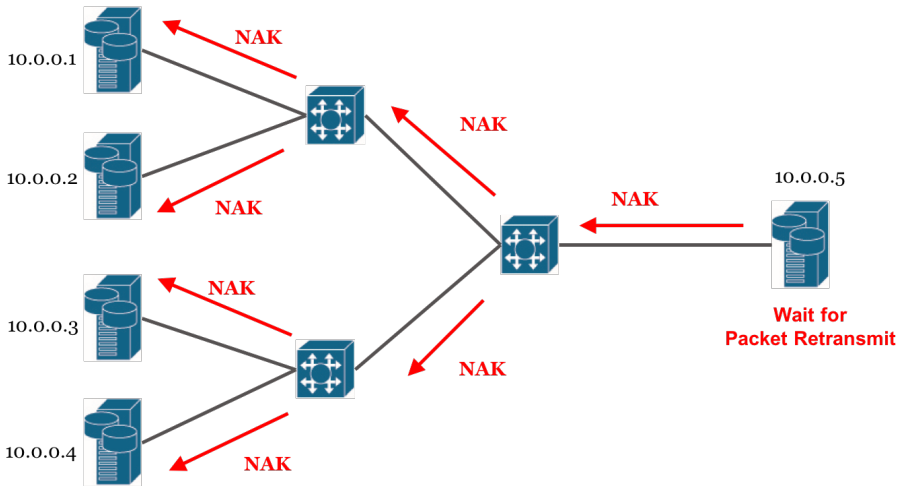

Figure 2.9: Reverse Multicast Protocol: NAK (No Acknowledgement) Message.

processing rules which are expressed by a sequence of *Actions*. The forwarding rules determine whether the packets are forwarded to destination or sent back for table selection at the next layer.

To extend the data path we follow the framework proposed by Casey et. al. [16]. The framework focuses on the OpenFlow data model, but is general enough to capture a general SDN data planes. In this framework, the dataplane consists of a set of basic primitives, such as buffers, ports, flow tables, meters, and queues. Each abstraction has a set of interfaces that allow the controller to query the abstraction, configure its properties, obtain statistics, and subscribe to events. More the specifically, there are four types of general interfaces in any North Bound Interface:

- *Capabilities.* This interface is used by the controller for discovering the functionality supported by this abstraction.

- *Configuration.* The interface is responsible for changing the behavior of the abstraction.

- *Statistics.* This interface is used for delivering the statistics about the abstraction operation to the controller.

- *Events.* This allows the controller to receive notifications about specific events associated with this abstraction.

The modified packet processing pipeline includes a new primitive, referred to as a *Parking Lot*, as well as extensions of *matching*, *action*, and *instruction* primitives. Figure 2.13 depicts the UML diagram of the the extended data plane that includes the new primitive. In addition, we extend the *extraction* stage of the pipeline to support the header of the RMP protocol. This and the North Bound Interface for Parking Lot is explained in the following sections.

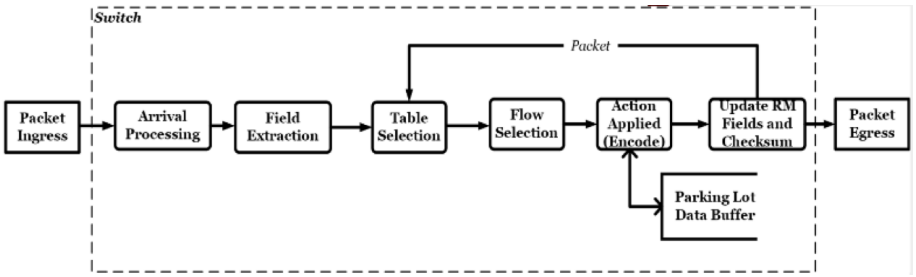

Figure 2.10: Packet Processing Pipeline Extended for Encoding Operations

2.3.1 Parking Lot Primitive

To support encoding at switches, we needed to to buffer the in-flight RMP DATA packets that arrive before their counterparts. The Parking Lot is implemented as a buffer in dataplane that supports fast insertion/deletion and fast lookup. For our implementation we used an unordered-map (actually multi-map to be precise) data structure [21]. Our experimental studies indicate that the Parking Lot does not introduce a significant overhead

in terms of performance (see Section 3.1). Figure 2.11 depicts the structure of the Parking Lot. The packets that belong to a different Tree ID are stored in a different branch. Each branch contains packets awaiting their counterparts belonging to the same Tree ID.

For each new RMP packet, we first check whether the Parking Lot already contains the packet originated from a different node that belongs to the same Tree ID and has the same sequence number. If not, the packet is added to the Parking Lot. If not, the packet is coded together with its counterpart in the Parking Lot. The resulting part is either remains in the Parking Lot (in case another packet needs to be added) or forwarded via an outgoing interface.

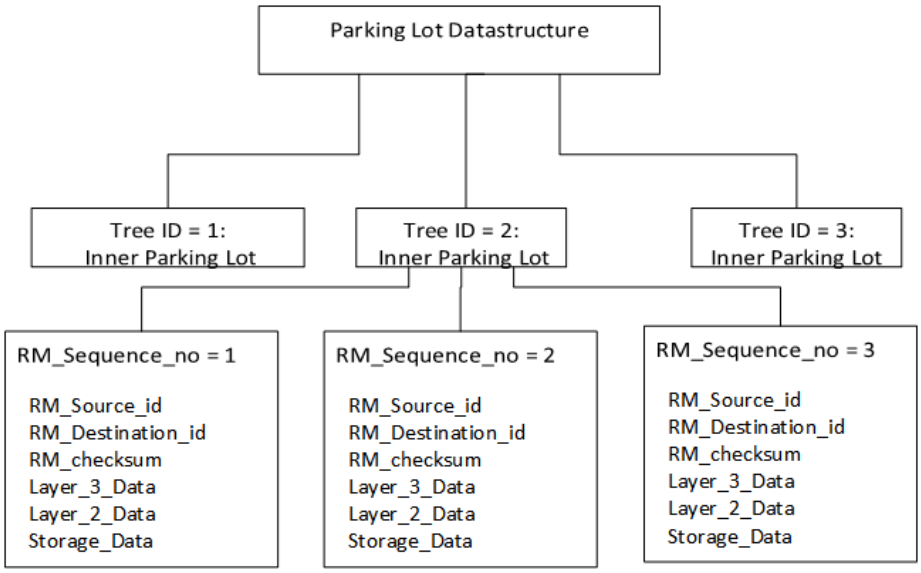

Figure 2.11: Parking Lot Data Structure

2.3.2 Parking Lot Design

For this work we needed a data structure with fast insertion/deletion and fast lookup, to take advantage of the fast switch dataplane as well as reduce as much processing latency as

possible. Thus, we used an unordered-map data structure in form of $unordered\_multimap$ from C++ STL as it has all our desired characteristics [21]. Here since packets can very well arrive out of order at any switch, we do not really have any need for having any form of ordering in the Parking Lot. By not having a ordered data structure we can perform faster lookups and insertions, and since we want to have the lowest latency at switch for these operations, we use unordered_map(). They largely have a $O(1)$ constant lookup and insertion time.

2.3.3 Parking Lot Implementation

Packets are pushed to the Parking Lot if there is no other packet with similar Tree ID, Destination ID and sequence number. If there is a packet already in the inner-Parking Lot, we perform encoding operation of the new and the stored packet and update the entry(the checksum and DATA portion is encoded).

For each inner Parking Lot, there exists a unique "full" parity (set by the Storage Manager during system setup and written to switch as flow-rules by controller). - This parity indicates when the particular packet has been "fully-encoded". Till this condition is met, packets can be encoded within the same entry. This can allow us to support multiple inflows of packets belonging to the same RM-Tree, with each packet encoding with an older encoded packet and parity field being updated. Once it becomes "fully-encoded" (obtains the required parity) the packet is sent to the Field Extraction/matching phase where it will be routed to the next hop using a custom action. This entire process is illustrated in Figure 2.12

Important header information: As mentioned above, the extended field extraction stage should be able recognize and copy data from the following fields of the RMP header:

- RM *Tree ID* - This fields allows us to establish forwarding and coding rules for packets with various tree IDs.
- RM *Message Type* - This field enables different behaviors for different RMP packet types, such as DATA, SYN, ACKs, NACKs, and FIN. In particular, the coding operations only apply to packets of the DATA type.
- RM *Source ID* - allows us to distinguish between packets that arrive from different child nodes of the tree.
- RM *Destination ID* - provides information about destination node of each RMP packet.
- RM *Parity* - provides the current parity of the packet. This is to be updated to parity specified in flow-rule when full encoding is performed.

We proceed to discuss the key custom Openflow action, referred to as *BIT_CODE*.

BIT_CODE action. This action is applied for each packet of type DATA. For each incoming packet, a look-up is conducted on the Parking Lot. In particular, we use the packet's RM sequence number and RM *Destination ID* fields to check for a corresponding packet with different source ID but same Tree ID, Destination ID and sequence number. These data portion of these packets then undergo a bitwise encoding operation (this paper is limited to XOR operation, but it can generalized to a linear coding scheme over the finite field). The Parking Lot entry is then updated with details of new packet. Next, the new RM *parity* is calculated for the new packet. The new packet either remains in the Parking Lot or forwarded to parent node of the tree.

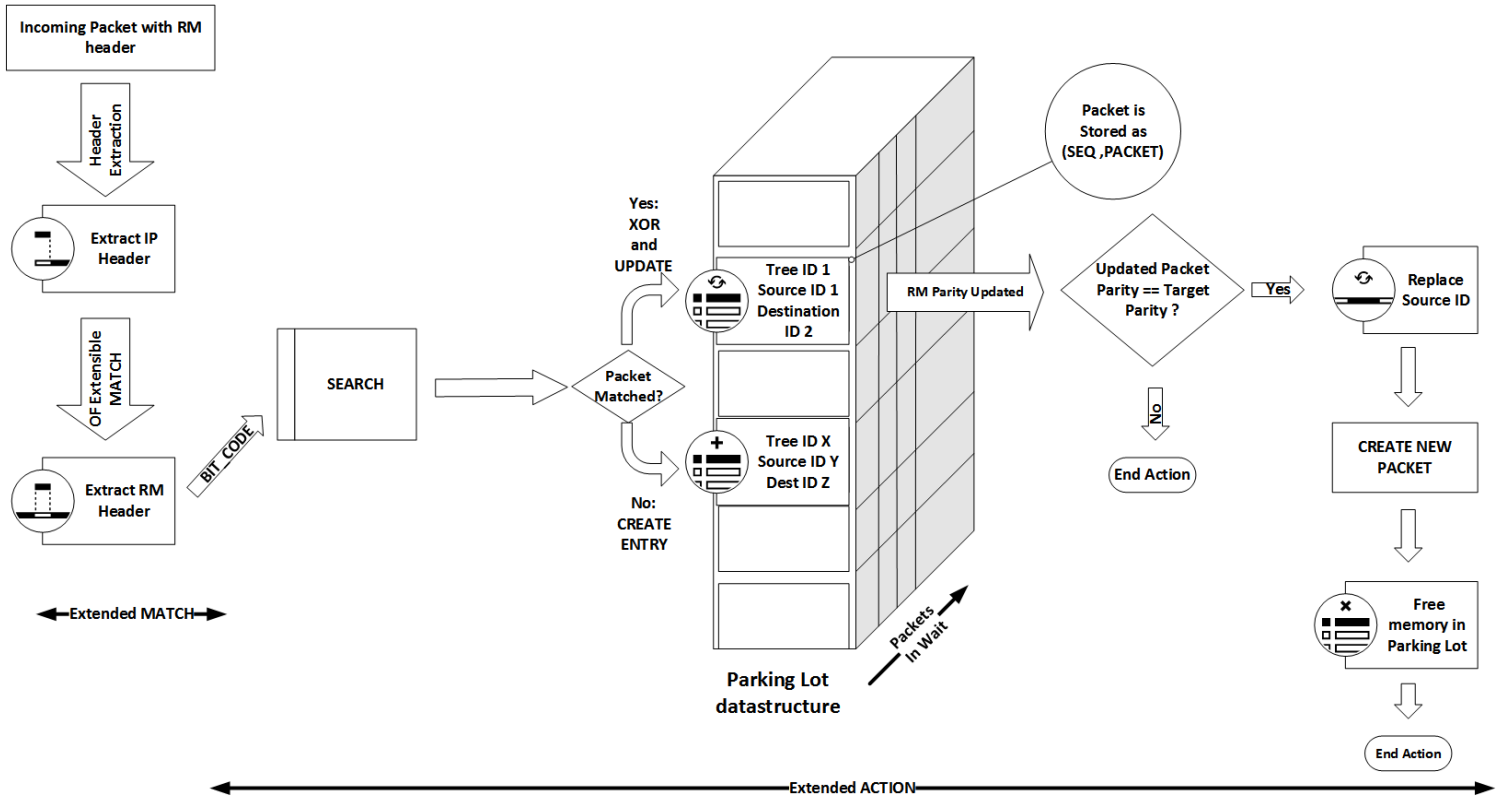

Figure 2.12: Operation Flowchart of Parking Lot within SDN Compatible Switch

We proceed to discuss the final custom Openflow action, referred to as **FWD_ENCODED.** This action is used to forward the processed packet to the next stage. Packets with target RMP tree ID and target client IP and target parity address are forwarded to the next hop. Both the above mentioned actions can be implemented by using Openflow Extended Actions, and we set the flow rules according to our needs using the Openflow Experimenter messages between the controller and dataplane.

2.3.4 SDN Dataplane Abstractions and Extensions

In order to enable controller's access to new primitives, the SDN protocol needed to be extended as well. In our implementation, we use the *Experimental* messages of the OpenFlow protocol. These messages allow the controller to collect statistics such as size variations in the Parking Lot, query the number of packets stored in Parking Lot of a certain reverse multicast tree. The controller can also configure the number of distinct Tree IDs supported by the Parking Lot, as well as create and delete new branches that store different Tree IDs.

In Figure 2.13 we indicate the change in the standard SDN dataplane abstraction. Further the Parking Lot structure has the following interfaces:

- *Capabilities.* This interface allows the controller to determine the Parking Lot capabilities, such as its maximum capacity (i.e., the total number of packets that can be stored), the maximum number different branches that store Tree IDs, the maximum number of packets that can be stored in each branch.

- *Configuration.* This interface allows the controller configure the number of Parking Lot branches, to create and delete branches, and to configure the number number of packets that can be stored at each branch. This interface is also used to switch between different values of time out, after which the packet will be removed from the Parking Lot.

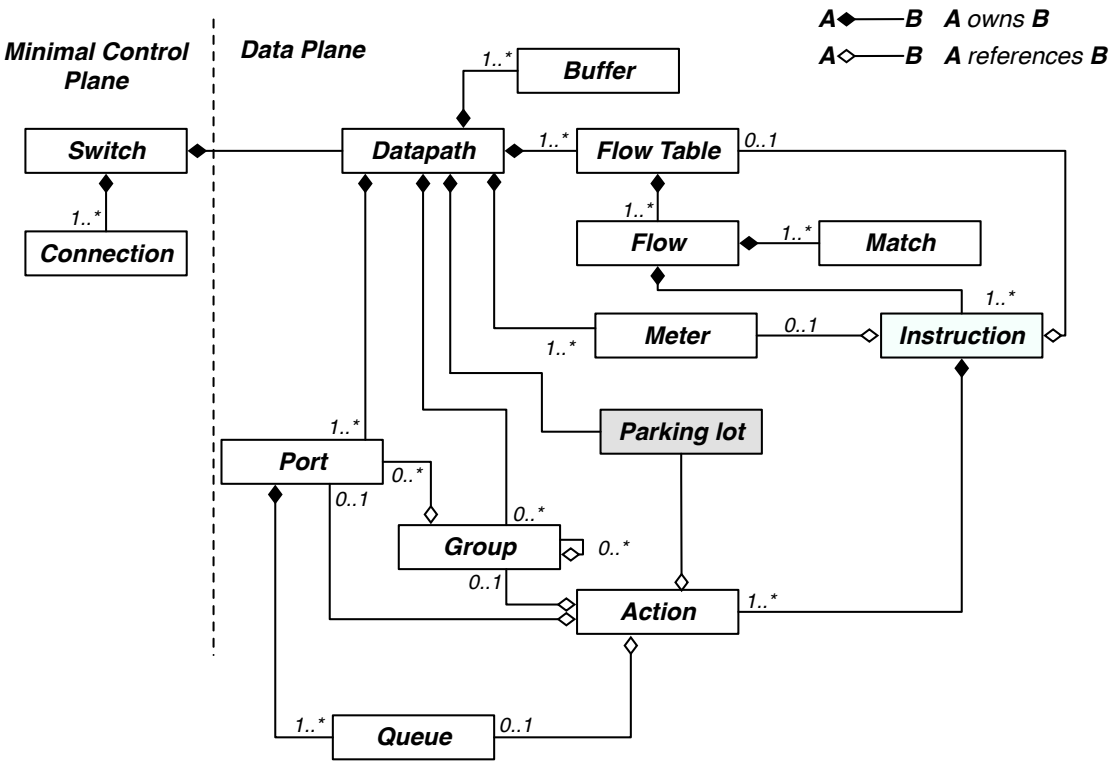

Figure 2.13: Software Switch Dataplane Abstraction

- *Statistics.* This interface allows the controller to gather statistics about the Parking Lot performance, such as the number of packets deleted due to the time out expirations, the average occupancy of the Parking Lot, the average waiting delay and many others.

- *Events.* This interface allows the controller to subscribe to various Parking Lot events, such as time-out expiration and the occupancy reaching a certain limit.

3. IMPLEMENTATION AND TEST RESULTS

3.1 Implementation and Experimental Results

To simulate the Reverse Multicast Protocol and its impact on data storage operations, we created applications to simulate coded reconstruction operations from a storage manager in our theoretical DSS. A simulated storage manager application, data storage node application, and end-user client application were created which used raw sockets to generate and receive RMP packets. For iSCSI based transfers we immediately obtained linerate transfers as TCP scaling is enabled by default in Linux kernel.

For our SDN-enabled switch with network coding capabilities we used an existing Userspace softswitch with OF 1.3 with OpenFlow capabilities and implemented the aforementioned parking lot data structure. We then modified it's Flow match capabilities to detect RMP labels. Using a Nox controller (Zaku version) [22] along with Openflow Extensible Match (OXM) and Action features [23] we added match and action conditions to the controller to support the RMP. The Storage Manager operations were simulated by using a Python based script which we used to perform remote procedure calls which initiate/terminate the RM protocol data transfer.

For our testbed we used 8 TAMU ExoGeni based Virtual Machines [24]. One VM served as a central switch, another was used to house the controller and storage manager application, two/three/four served as data storage nodes, and one served as a client. We modified various network characteristics such as link bandwidth and packet drop percentage using Linux utilities such as netem and ethtool [25] in order to more closely simulate bandwidth limits in actual network hardware. Apart from this GENI allows us to modify various aspects of our topology as needed. The VMs came with native Ubuntu 14.04, however needed upgrades/packages for our system to work.

As a baseline, we compared with performance of data transfer via iSCSI during the repair process for a software raid 5 share across the network. Raid 5 share was made possible by using Mdadm [13] (software RAID on linux) [26, 27] and by using iSCSI targets and initiators serving as block devices located on different VMs. To simulate failure we use the native $mdadm\ manage$ commands and setting one of the drives to a "$fail$" status and observed the quantity and throughout of data transferred and recorded our observations during the rebuild process (also initiated by mdadm command). The test setups are indicated by Figures 3.1 to 3.3.

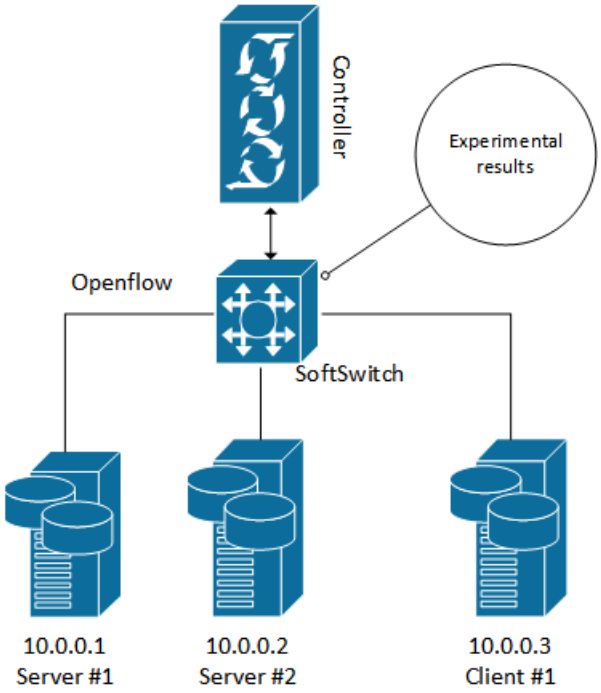

Figure 3.1: Topology for Preliminary Investigation with Two Servers and One Client.

In our experimental testbed we measured the following:

1. Data transfer rate per data storage node or Average Throughput per source over lossy

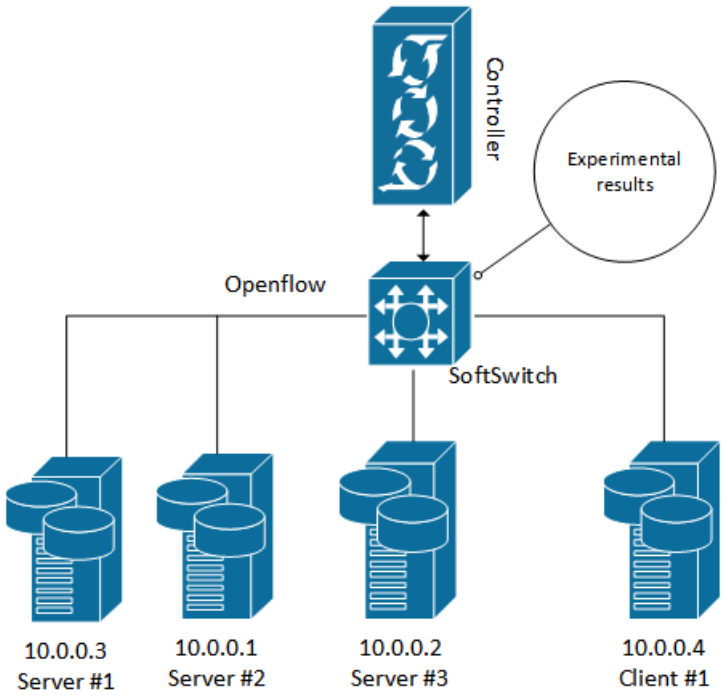

Figure 3.2: Topology for Preliminary Investigation with Three Servers and One Client.

links for various link bandwidths using RMP and TCP transfers. This was tested with 0.01% error rate to measure effects of near lossless and low latency networks and effect of available link bandwidth as well as error-prone networks and this was compared with a TCP based protocol like iSCSI. The mean RTT was observed to be 1.4 ms.

2. The overall Peak/Mean CPU usage difference of our modified Softswitch from with its base version, and see the overall impact on a switch's cpu usage. This measurement is important as a switch often has to handle large quantities of flows and should have processing power to be able to handle additional encoding operations

3. Average time taken to delete entries in the parking lot to see how efficient the data structure is at encoding packets and forwarding them along the pipeline. Measuring

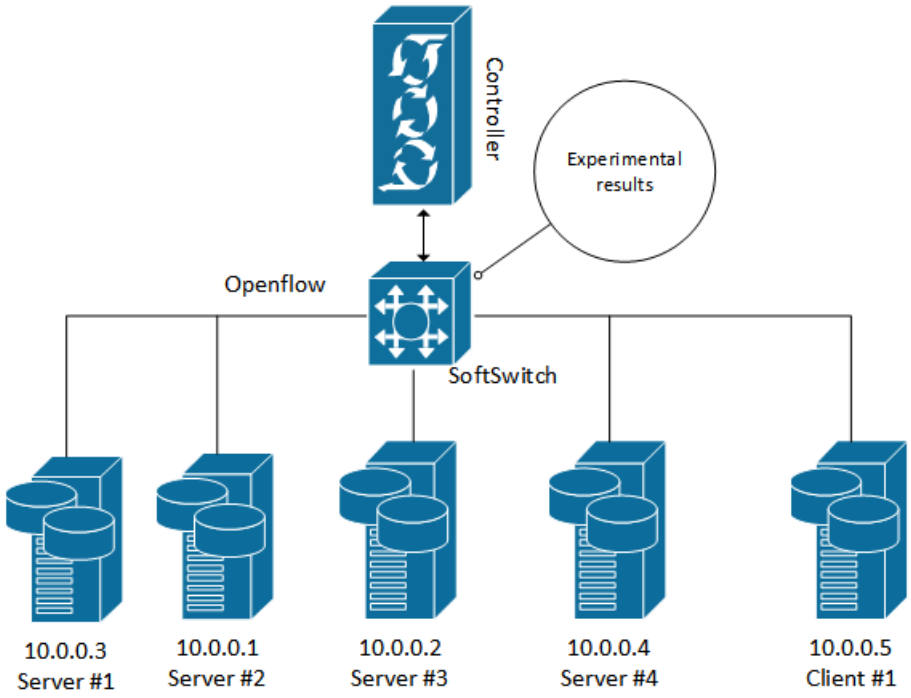

Figure 3.3: Topology for Preliminary Investigation with Four Servers and One Client.

Parking Lot performance in-terms of average lookup and encode operation, average random/sequential insertion time and giving us a feedback on overall parking lot performance. It is essential for a high hit data-structure in switch to perform to at its highest functionality. This was measured on Setup 1 only.

4. Average reduction in time (in %) taken to transfer repair data over bottlenecked network.

We measured the bandwidth available to each data storage node while transferring/encoding 1 GB of data (each node has 1 GB of data as the Exogeni VMs had net storage of 10 GB each) to the requesting client. We also limited the available bandwidth on the requesting client's link using netem, tc and ethtool linux utilities.

The average processing time for sequential and random insertion events, removing

parking lot entries, and entry lookup followed by the bitwise xor operation is illustrated in 3.4. **Note:** We also found by measuring TCP traffic (without including the parking lot structure) that the average packet processing time (for simple forwarding operations and limited controller triggering) for a packet inside the softswitch was **67**u**s** and more complex operations involving heavy traffic (including regular triggering controller at 1 ms from switch) took **2046** u**s** . Since the average perpacket service time (for the parking lot) is far less than 67us as shown in 3.4, this additional processing time has a very marginal impact on overall switch performance. Thus *we can add our custom parking lot feature within the switch without significantly increasing processing time*.

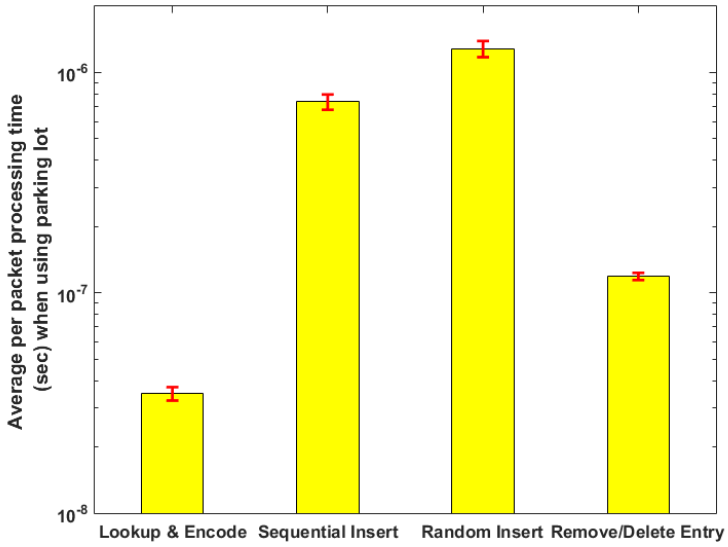

Figure 3.4: Parking Lot Data-Structure Performance Evaluation.

The overall rate of data transfer as measured by the rate useful data is received at the client with various link speed settings is given in Figures 3.5 to 3.9 and is graphed in the following sections. In our tests, RMP outperforms TCP for any given link resource setting

and is able to fully use it's available resources in the client's bottlenecked link for receiving decoded data. In the TCP case, two/three and four TCP connections shared the same link resources and were unable to transfer data at a higher rate without causing packet loss.

3.1.1 Results for Setup 1 for Various Link Speeds

The results for Setup 1 with three Storage Clients (one being the reciever and two others being the transmitters) have been showcased in Figure 3.5.

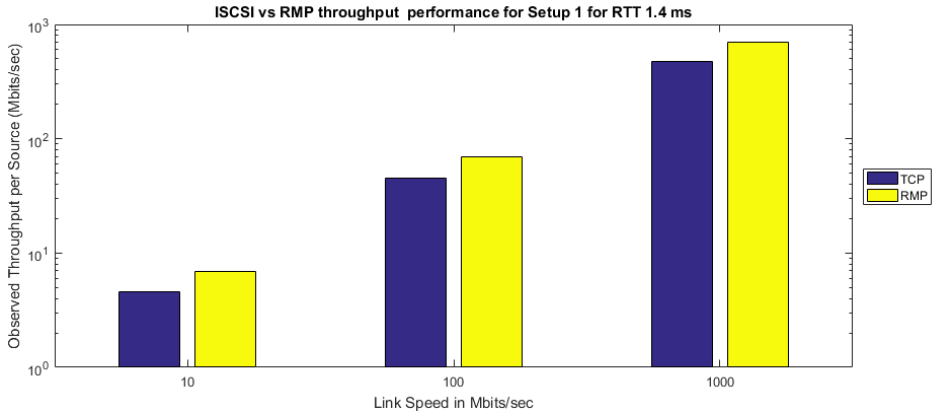

Figure 3.5: Performance of RMP vs ISCSI in Setup 1.

3.1.2 Results for Setup 2 for Various Link Speeds

The results for Setup 2 with four Storage Clients (one being the reciever and three others being the transmitters) have been showcased in Figure 3.6.

3.1.3 Results for Setup 3 for Various Link Speeds

The results for Setup 3 with 5 Storage Clients (one being the reciever and four others being the transmitters) have been showcased in Figure 3.7.

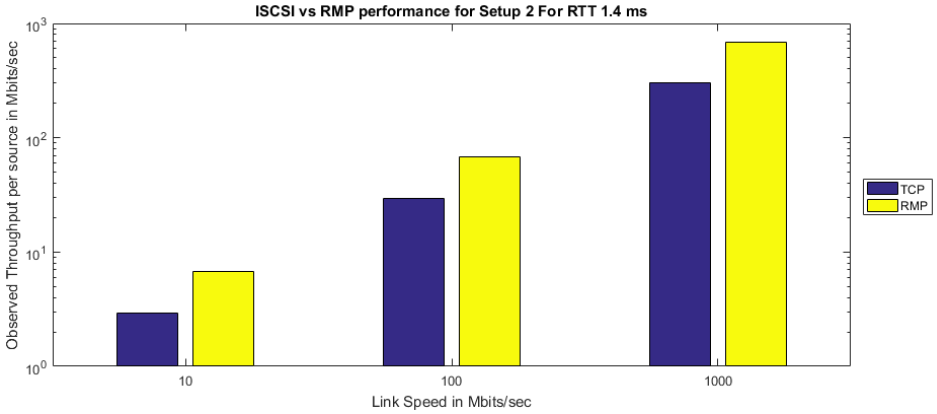

Figure 3.6: Performance of RMP vs ISCSI for Setup 2.

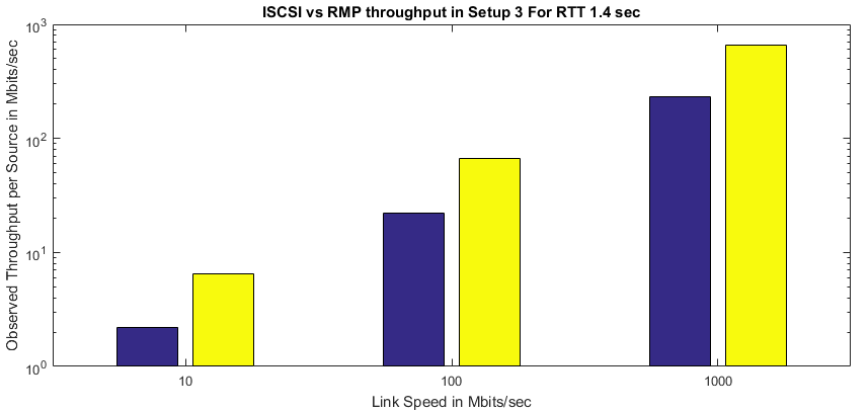

Figure 3.7: Performance of RMP vs ISCSI for Setup 3.

3.1.4 Repair Time Reduction

3.1.5 Throughput Results and Repair Time Reduction Discussion

This test was used with netem simulating 0.005% packet loss on each of the client and data node links(for a single hop system this would be 0.01% overall packet loss probability for each source-switch-destination link). This had a relatively minor impact on the TCP performance. However, due to the nature of the NAK system in our RMP protocol, every

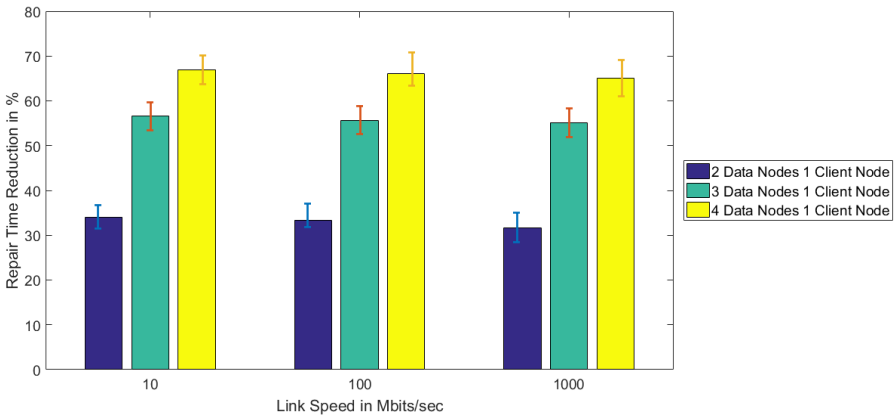

Figure 3.8: Reduction of Time Taken to Repair Failed Single Node from Multiple Setups.

packet loss meant that we needed to retransmit packets from all data storage nodes/or wait for time out at the switch and wait for response to corresponding NAK data storage nodes to recreate the encoded-packet for the client. This caused a more significant drop in the overall data transfer rate for the RMP protocol. Despite this, the RMP protocol still outperformed the TCP protocol in these tests, as shown in 3.5 to 3.7 attempting transfers with more than two data storage nodes will cause TCP to outperform our RMP implementation in more lossy networks (higher packet drop rate).

However, in a low-latency datacenter environment, this will not be an issue and as 0.01% is an acceptable failure rate in data centers (which normally guarantee 99.99% success rate of packet transfer).

In the next section with Figure 3.9 and 3.10, we look at how the switch CPU performed, by measuring the difference in CPU usage % between the base SoftSwitch and our modified switch over multiple setups and link speeds. The % usage is calculated by the operating system's process or task scheduler. For example, if CPU usage is 10% it means that the task has been actively running for 10% of the task scheduler's time units.

3.1.6 Results for Peak and Mean CPU usage (%) in the Central Encoding Switch for Different Workloads

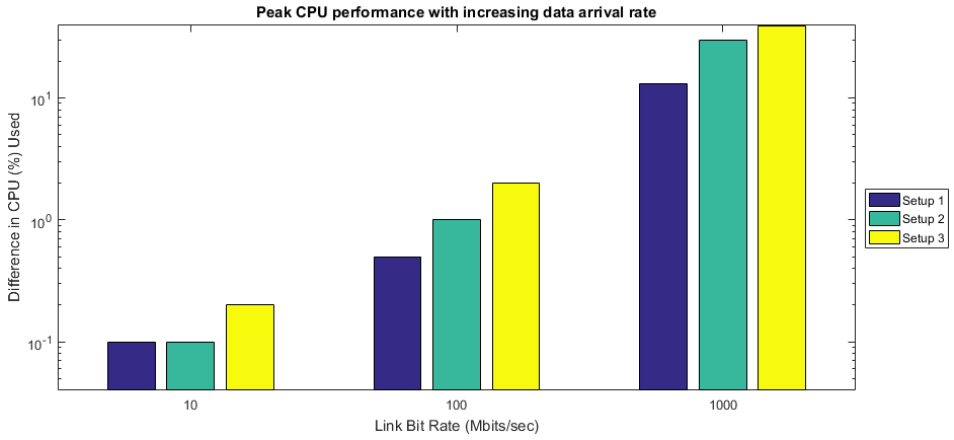

Figure 3.9: Mean CPU Performance Difference Between Base Switch and Modified Switch in 3 Setups.

We measured CPU usage (a process' share of elapsed cpu time) by recording the % CPU utilization measurement in top utility in linux and for average CPU usage and we recorded the average CPU utilization by our modified switch using ps utility. ps records a "snapshot" of current CPU usage, thus allowing us to record peak usage by recording multiple snapshots. top is a monitor we used to record mean usage.

Here we observed that CPU usage increases observed for normal low-loss low latency

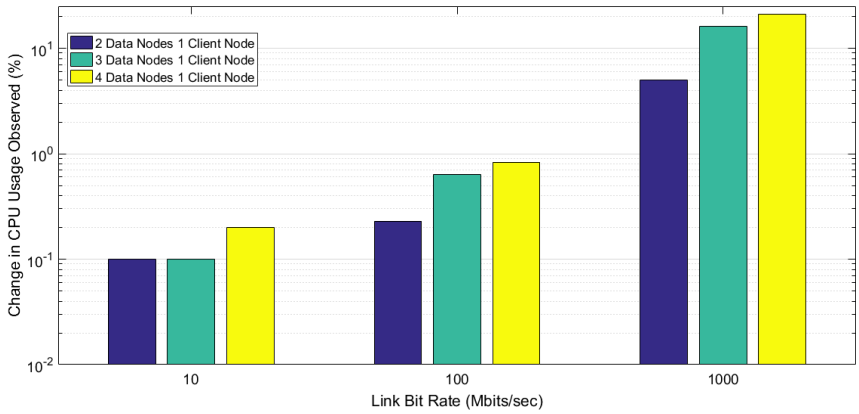

Figure 3.10: Peak CPU Performance Difference between Base Switch and Modified Switch.

operations in 3.9 did not have much of a noticeable impact modified switch even in case of Gigabit link. However, we did observe stray peaks every now and then recorded our observations in the Figure 3.10.

This increase in CPU usage can be attributed to the additional encoding, action and field extraction operations we are now performing within our modified switch.

4. SUMMARY AND CONCLUSIONS

In our work we have designed and developed the Reverse Multicast Protocol and showcased its potential benefits over regular unicast protocols in data intensive applications such as DSS, when used with supporting policies from an SDN controller to improve repair transfer throughput. While there are still short-comings for RMP (such as performance drops due to packet loss over a lossy channel and over high latency links) this work showcases the possibility of a principled approach to performing in-flight packet processing by the network. Furthermore we demonstrate that it can be done without suffering a major performance penalty at the switch. However, peak CPU usage and resilience against packet drops are still major areas of concern especially with supporting line rate transfers in gigabit links.

4.1 Further Study

In the near future, we plan on exploring more complex packet encoding strategies within the switch to support regenerative codes proposed by academia and industry alike as shown in [10]. We also intend to explore methods to further improve the robustness of the RMP, especially in cases where packet loss takes place as well as look for further improvement in optimization. We can also think of strategies to augment exisitng storage managers and implement a full scale Storage Manager prototype to perform storage operations. Finally we plan on stress testing our architecture's adaptability to changing network topologies, workloads, and different link failures. Furthermore, we are also looking at improving performance at switches by using different data structures such as google dense_hash, google sparse_hash datastructure etc [28] for memory optimization and make our solution more robust and be able to handle 10 or 40 gigabit link traffic with enterprise level workloads. Further we plan on crafting all our future solutions based on P4 [29] and

other Hardware switches.

REFERENCES

[1] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 19, ACM, 2010.

[2] 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.

[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] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree Filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, p. 9, 2013.

[5] Veritas, "Veritas Global Databerg Report Finds 85% of Stored Data Is Either Dark, or Redundant, Obsolete, or Trivial (ROT)," Mar 2016.

[6] H. Geng and M. Kajimoto, "Lessons Learned From Natural Disasters and Preparedness of Data Centers," *Data Center Handbook*, pp. 659–667, 2015.

[7] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang, "NCCloud: Applying Network Coding for the Storage Repair in a Cloud-Of-Clouds," in *FAST*, p. 21, 2012.

[8] D. Carteau, "Three Interconnected Raid Disk Controller Data Processing System Architecture," 2001. US Patent 6,330,642.

[9] 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.

[10] 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.

[11] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters.," in *FAST*, vol. 2, pp. 231–244, 2002.

[12] O. Rodeh and A. Teperman, "ZFS-A Scalable Distributed File System Using Object Disks," in *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pp. 207–218, IEEE, 2003.

[13] D. Greaves., N. Yeates, "Mdadm," *RAID Wiki*, (Date last accessed 11-Feb-2017). [Online]. Available: https://raid.wiki.kernel.org/index.php/Linux_Raid. Published: 2016.

[14] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," in *USENIX Annual Technical Conference*, vol. 15, 1996.

[15] R. Koetter and M. Médard, "An Algebraic Approach to Network Coding," *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 5, pp. 782–795, 2003.

[16] C. J. Casey, A. Sutton, and A. Sprintson, "TinyNBI: Distilling an API From Essential OpenFlow Abstractions," in *Proceedings of the Third Workshop on Hot topics In Software Defined Networking*, pp. 37–42, ACM, 2014.

[17] J. Satran and K. Meth, "Internet Small Computer Systems Interface (ISCSI)," (Date last accessed 11-Feb-2017). [Online]. Available: https://tools.ietf.org/html/rfc3720 Published: 2004.

[18] T. Speakman, D. Farinacci, S. Lin, and A. Tweedly, "Pragmatic Group Multicast (PGM) Transport Protocol Specification," tech. rep., Internet Draft, Work in

Progress, 1998.

[19] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[20] F. Németh, Á. Stipkovits, B. Sonkoly, and A. Gulyás, "Towards SmartFlow: Case Studies on Enhanced Programmable Forwarding in OpenFlow Switches," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 85–86, 2012.

[21] N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 2012.

[22] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[23] F. Hu, *Network Innovation Through OpenFlow and SDN: Principles and Design*. CRC Press, 2014.

[24] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "GENI: A Federated Testbed for Innovative Network Experiments," *Computer Networks*, vol. 61, pp. 5–23, 2014.

[25] Linux Foundation, "NetEm." (Date last accessed 11-Feb-2017). [Online]. Available: https://wiki.linuxfoundation.org/networking/netem. Published: May, 2013.

[26] "Distributed RAID Over Network," (Date last accessed 15-Feb-2017). [Online]. Available: http://unix.stackexchange.com/questions/222686/how-to-implement-raid-6-over-different-nodes-on-the-network. Published: May, 2012.

[27] "Distributed Raid Over Network Linux Setup," (Date last accessed 14-Feb-2017). [Online]. Available: https://www.mylinuxplace.com/distributed-raid-over-iscsi/. Published: Feb, 2015.

[28] V. Alvarez, S. Richter, X. Chen, and J. Dittrich, "A Comparison of Adaptive Radix Trees and Hash Tables," in *2015 IEEE 31st International Conference on Data Engineering*, pp. 1227–1238, IEEE, 2015.

[29] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.