Design and Implementation of a Scalable Storage System for Fully-Consistent Replicated Data Logging

Giorgos Saloustros

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisor: Prof. *Angelos Bilas*

Thesis Advisor: FORTH Researcher*Kostas Magoutis*

This work has been performed at the **ICS-FORTH, N. Plastira 100 Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece**

The work is partially supported by the **European ICT-FP7 program through the SCALEWORKS (MC IEF 237677) project**

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδιασμός και Υλοποίηση ενός Πλήρως-Συνεπούς Κλιμακώσιμου Συστήματος Αποθήκευσης για Δεδομένα Συνεχούς Ροής

Εργασία που υποβλήθηκε από τον

Γιώργο Σαλούστρο

ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας: _____
Γιώργος Σαλούστρος, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή: _____
Άγγελος Μπίλας, Αναπληρωτής Καθηγητής, Επόπτης

_____
Κώστας Μαγκούτης, Ερευνητής Ινστιτούτου Πληροφορικής ΙΤΕ, Επιβλέπων

_____
Δημήτρης Νικολόπουλος, Αναπληρωτής Καθηγητής, Μέλος

Δεκτή: _____
Άγγελος Μπίλας, Αναπληρωτής Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Ιούλιος 2011

Abstract

The rapid growth of Internet-scale applications built over commodity hardware for cost-efficiency has created the challenge for IT architects to design storage systems that are scalable and highly available. In addition, an increasingly competitive scalable software market raises the need for rapid prototyping of such systems. In this thesis we show a new approach for rapidly improving the robustness of an existing scalable file system, by designing and implementing a strongly-consistent replication primitive as a re-targetable building block that we then retrofit with minimal engineering effort into the scalable file system. Our replication primitive (ZKRL) extends a general-purpose open-source coordination service (Apache Zookeeper) to maintain consistent data replicas across read and append operations. By cleanly replacing the ad-hoc replication protocol of the Hadoop scalable file system (HDFS) we achieve stronger overall data replication semantics. The resulting system thus becomes a candidate for a wider range of applications than previously possible. This thesis further shows that this improvement need not incur a performance penalty: Our results show that we can achieve performance close to peak I/O (network or disk) capabilities when appending to replicated logs over 1-Gb/s and 10-Gb/s networks.

Περίληψη

Η όλο και πιο διαδεδομένη ανάπτυξη εφαρμογών σε κλίμακα Διαδικτύου εκτελέσιμων πάνω από υλικό γενικού σκοπού, έχουν δημιουργήσει την πρόκληση στους αρχιτέκτονες λογισμικού να σχεδιάζουν συστήματα αποθήκευσης τα οποία να ικανοποιούν ιδιότητες κλιμακωσιμότητας, διαθεσιμότητας και υψηλής απόδοσης. Επιπρόσθετα, μια αυξανόμενη και ολοένα πιο ανταγωνιστική αγορά κλιμακώσιμου λογισμικού εγείρει την ανάγκη για ταχεία ανάπτυξη τέτοιων συστημάτων. Σε αυτήν την εργασία παρουσιάζουμε μια νέα προσέγγιση για την βελτίωση της αξιοπιστίας και της διαθεσιμότητας ενός υπάρχοντος κλιμακώσιμου συστήματος αρχείων, η οποία βασίζεται στην σχεδίαση ενός πλήρως συνεπούς, θεμελιώδους στοιχείου λογισμικού (ZKRL), το οποίο μπορεί να χρησιμοποιηθεί σαν δομική μονάδα ενός ευρύτερου συστήματος λογισμικού. Η υλοποίηση του ZKRL επεκτείνει μια υπηρεσία συντονισμού γενικού σκοπού (Apache Zookeeper) για να κρατάει αντίγραφα δεδομένων. Εν συνεχεία, χρησιμοποιούμε το ZKRL για να αντικαταστήσουμε το πρωτόκολλο αντιγραφής ενός υπάρχοντος κλιμακώσιμου συστήματος αρχείων, του Hadoop File System (HDFS), δυναμώνοντας με αυτόν τον τρόπο την αξιοποστία των δεδομένων. Το τελικό σύστημα διαθέτει πιο ισχυρές ιδιότητες κλιμακωσιμότητας και διαθεσιμότητας κάνοντας το κατάλληλο για ένα μεγαλύτερο εύρος εφαρμογών σε σχέση με το αρχικό. Η εργασία αυτή επιπλέον δείχνει ότι αυτή η βελτίωση δεν μειώνει την απόδοση: Τα αποτελέσματα της αξιολόγησης του συστήματος δείχνουν ότι μπορεί να επιτύχει απόδοση κοντά στο όριο του υλικού Εισόδου-/Εξόδου (δίσκος ή δίκτυο) πάνω από δίκτυα ταχύτητας 1 Gbps και 10 Gbps.

Acknowledgements

Contents

1 Introduction **1**
- 1.1 Thesis contributions . . . 3
- 1.2 Thesis organization . . . 4

2 Background **5**

3 ZKRL Design **11**
- 3.1 Overview of Zookeeper . . . 11
- 3.2 Persisted replicated logs . . . 12
- 3.3 Protocol operation . . . 15
  - 3.3.1 Failure-free Operation . . . 15
  - 3.3.2 Failure Recovery . . . 15
- 3.4 Implementation . . . 19
  - 3.4.1 Quorum packet structure . . . 19
  - 3.4.2 ZKRL Data Path . . . 20
  - 3.4.3 Optimizations . . . 22
  - 3.4.4 ZKRL Read Path . . . 25
  - 3.4.5 Failure scenarios . . . 25

4 Integration of ZKRL with HDFS **27**
- 4.1 Overview of HDFS . . . 27
- 4.2 Benefits of integration . . . 29
- 4.3 Implementation . . . 29
  - 4.3.1 Compressing the ZKRL znode tree . . . 32
  - 4.3.2 HDFS read path modifications . . . 33

5 Evaluation **35**
- 5.1 Performance of ZKRL . . . 37
  - 5.1.1 Write Performance . . . 37
  - 5.1.2 Read Performance . . . 42
- 5.2 Recovery of ZKRL . . . 43

5.3 Performance of HDFS-ZKRL 44
5.3.1 HDFS Write Performance 44
5.3.2 HDFS Read Performance 48

6 Conclusions and Future Work **53**

List of Figures

3.1 Zookeeper replicated state machines. 12
3.2 Transforming Zookeeper into a persistent chain-replicated log. 14
3.3 ZKRL failure free operation 17
3.4 Protocol operation during recovery 18
3.5 Quorum Packet Structure 19
3.6 ZKRL data paths 20

4.1 HDFS - ZKRL integration. 31
4.2 ZKRL metadata in HDFS. 32
4.3 ZKRL tree compression in HDFS. 33

5.1 ZKRL throughput with variable proposal sizes. 38
5.2 ZKRL sync response time 39
5.3 ZKRL mem response time 40
5.4 ZKRL throughput under different system configurations. . . . 41
5.5 ZKRL mem/disk for 10Gbps network 42
5.6 Write throughput for hdfs for 64MB block size 46
5.7 Write throughput for hdfs for 8192MB block size 47
5.8 Write throughput/CPU for (2, 6), (2, 2x3) setups. 48
5.9 HDFS write performance in 10Gbps network 49
5.10 Read performance of hdfs for 1Gbps network 50
5.11 Read Performance of hdfs for 10Gbps network 51

Chapter 1

Introduction

Demands for scalability and performance in storage infrastructure are continuously increasing due to the growth of Internet-scale applications such as e-mail, video streaming, web indexing and stream processing. Moreover, the need for rapid design, prototyping, and deployment of these new Internet applications creates challenges for IT architects that have to design for a variety of trade-offs regarding performance, system availability, and consistency. In the past, an approach to overcome these constraints was the relaxing of data consistency semantics. This was considered to be a reasonable price to pay (especially if you control the applications) for reduced time-to-market, as Google did in the design of GFS [25]. Another way to reduce development time is to use component-driven software engineering in which reusable software components can be synthesized to the desired specs. Decoupling the application logic from the storage layer is one step towards this goal that enables applications to match with their desired storage semantics. Systems such as Boxwood [11] and DDS [17] were designed to offer a toolbox of storage components that can be used to synthesize application-specific designs.

Another issue in the design and implementation of scalable storage systems is the use of commodity hardware for cost-efficiency. However, the failures in this type of environment as mentioned from Google data center reports [32] are high, so there is the need for high availability. For dealing only with disk failures more lightweight approaches can be used such as

RAID but these solutions handle only disk failures and not network, node or power failures. This leads us to a distributed systems solution, that is implemented in software, which is typically achieved by maintaining copies (or replicas) of data in failure-independent devices across network. Another choice to be made is whether the software for high availability will be built in user space or kernel space. Development in user space has the benefits of portability and easier development and maintenance where the kernel space decision may lead to better performance.

In this thesis we combine these two trends, reusable software components for building scalable storage systems and data replication for high availability, to design and implement a new software component we call ZKRL (for Zookeeper Replicated logs). ZKRL is a storage service implementing a log abstraction and API using strongly-consistent data replication. It inherits a distributed consensus algorithm from a coordination service (Zookeeper [34]) designed for distributed synchronization and configuration management.

Zookeeper, in a nutshell, keeps an in-memory hash table structure persistent and consistent among a set of servers. The changes we made to transform this coordination service to a storage service are the following: First, ZooKeeper is a memory-only fixed-size object store that must be extended to handle arbitrary-sized data that live on persistent storage. Second, ZooKeeper's failure-recovery algorithm must be modified to handle state synchronization of the disk-based data across replicas. Third, since ZooKeeper is geared towards small read-mostly transactions we must further optimize it for large write-intensive transactions. This entails removing certain data movement overheads in the data path, performing language run-time and operating system specific tuning, and replacing the tree-communication pattern that is the basis of its atomic broadcast protocol currently to a chain pattern (each node receives from only one node and/or sends to only one other node). While the latter is expected to reduce CPU overhead and wasted network bandwidth, it requires modifications to the ZooKeeper protocol, which relies on a FIFO channels assumption that is no longer directly supported by the topology, to ensure correctness.

To exhibit ZKRL's typical use as a software component towards synthe-

sizing a larger software system, we embed (retrofit) ZKRL into a distributed file system (the Hadoop File System or HDFS), replacing its standard replication primitive. HDFS [14] is a distributed file system that has received significant interest by the scientific and business communities since its introduction in the open-source world in 2007. HDFS is inspired by the Google File System (GFS) [29] and shares its primary goals of being a scalable file system for supporting the Map-Reduce [6] framework and related data repositories [19] [9]. Meanwhile, HDFS has rapidly grown to become a de-facto candidate for applications with disparate requirements such as continuous stream processing [2], application checkpointing [10], machine learning [21], and general-purpose Cloud computing [5], stretching it from its original design goals. HDFS offers weak-consistency guarantees that in many cases can block application progress or produce incorrect results [25]. For example, its relaxed-consistency replication mechanism can under many failure scenarios violate single-copy serializability [27]. Given HDFS's growing popularity across application domains there is a pressing need to strengthen its data-access semantics. At the same time real-world software engineering considerations make it imperative that this improvement incurs minimal change to HDFS's codebase. Our system presented in this work addresses both these needs. The benefit of the synthesis between ZKRL and HDFS is the strengthening of the distributed file system's data semantics with very little integration complexity. The evaluation of the modified HDFS shows that is nearly as efficient as the original HDFS in I/O-intensive benchmarks, showing that one need not sacrifice performance for improved semantics. Our investigation further shows that our system is practical and robust, and that the fact that ZKRL is self-organizing can lead to simplified metadata-node architecture. However, software designers should make the task of software composition easier by using modularity and well-defined APIs into their distributed system designs.

1.1 Thesis contributions

The contributions of this thesis are summarized here:

1. The design and development of a distributed replicated logging system (ZKRL) that achieves high efficiency through aggressive optimizations on commodity hardware.

2. By retrofitting ZKRL into the Hadoop File System we show that HDFS's replication semantics can be improved and its metadata architecture simplified without sacrificing performance.

1.2 Thesis organization

The rest of this thesis is structured as follows: In Chapter 2 we present related work, in Chapter 3 we present our ZKRL design and implementation and in Chapter 4 our integration of ZKRL into HDFS. In Chapter 5 we describe our evaluation and experimental results and finally in Chapter 6 we conclude.

Chapter 2

Background

Replication in storage systems has received significant interest from the research community in the past. There are two main approaches for implementing replication: the primary-backup and the state machine approach [30]. In the primary-backup approach one server is designated as the primary and all the others as backups. If the primary fails, then a failover occurs and one of the backups takes over. In the state machine approach state service is replicated to all servers and the client requests are presented in the same order to all non-faulty servers. Implementing a state machine essentially requires achieving consensus between the servers, which is a fundamental problem in distributed systems that encapsulates the task of group agreement in the presence of faults; the processes propose values and have to agree on one among these values.

Two-phase commit [23] and Paxos [26] are two basic protocols that solve the consensus problem. The two-phase commit protocol (2PC) is a distributed algorithm that coordinates all the processes that participate in a distributed atomic transaction on whether to commit or abort (roll back) the transaction. In a failure-free execution the protocol consists of two phases:

1. The commit-request phase (or voting phase), where the coordinator process attempts to prepare all the transaction's participating processes.

2. The commit phase, in which, based on voting of the participant pro-

cesses, the coordinator decides whether to commit only if all participants have voted "Yes" or in any other case abort the transaction, and notifies the result to all the participants. The participants then follow with the needed actions (commit or abort) with their local resources.

The main problem of the 2PC is that in the presence of coordinator failure the protocol may block until the coordinator is repaired. This problem is solved in Paxos [26] which is an asynchronous consensus algorithm. In Paxos each node acts as proposer, acceptor and learner. Each proposer issues a proposal (which in ordered by an non negative increasing number n) and solicits acceptance from the acceptors. If acceptor has not seen a smaller proposal it accepts it and promises not to accept future proposals lower than n. When a majority of acceptors replies the proposer sends an accept message to the learners. The learners accept the message unless they have seen a greater n value. t If the majority of the learners acknowledge the proposal, it is delivered to the application.

One instance of a replicated state machine [30] implementation using the Paxos protocol is Chubby [4]. Chubby offers a centralized locking service that decouples synchronization primitives from application code in order to enable rapid deployment. The pros of implementing synchronization as a service and not as a local library are the following: First, this approach can support a wide range of applications, second, the locking service can scale up to serve a big population of clients and, third, it can provide high availability. Chubby is separated into cells, where each cell is consisted of a farm of servers (typically 5) and is responsible for a portion of the entire namespace. Write and read requests are all served by the same master server for each cell, which is elected via the Paxos algorithm [31]. It exposes a simple file-system like API supporting open/close, put/get and acquire lock operations.

Another general purpose coordination service which is inspired by Google's Chubby[4] is Zookeeper [34]. The main differences are that reads can be served by any replica in the group and lock acquisition can be built on top of Zookeeper client API; this is not part of the exposed API as Chubby. Finally Zookeeper uses ZAB [15] as a reliable broadcast protocol whereas Chubby uses an implementation of the Paxos algorithm. ZAB consists of two parts;

the broadcast protocol which is executed by a leader and the leader election protocol, which recovers from a leader failure. It guarantees sequential consistency semantics. Sequential consistency guarantees that the real execution looks to clients like some sequential execution in which the operations of every client appear in the order they were submitted. It does not guarantee that a read of one client returns the latest value written by another client. In addition to the two phase commit protocol abort messages are omitted; followers either acknowledge the message or abandon the leader. Through this simplification, the leader can commit the message as soon as the majority of the followers have acknowledged it. Also in addition to Paxos, there are two key properties that ZAB assumes, which lead in simplifying its design. First, ZAB assume TCP channels so message delivery is guaranteed in FIFO order. Second, it ensures that there is only one message proposal for a given proposal number simplifying its recovery.

In this thesis we use Zookeeper to design a storage service (ZKRL) for keeping consistent logs among a set of servers. Another approach with the same goals is Bookkeeper [20], a storage service designed for write-ahead logging. Bookkeeper consists of a number of servers that store the log files called Bookies. When each Bookie bootstraps it contacts a preconfigured Zookeeper cell and creates a node with its IP address and port for receiving new connections. Clients can choose the degree of replication for their log file and contact the Zookeeper cell to find available Bookies. Then the client chooses the quorum, n out of m Bookies, where n is the number of replicas for the log and m is the number of servers. The replication mechanism used in Bookkeeper is a primary-back up protocol, where the client acts as the coordinator to the Boookies. This raises performance issues because the client must send to each bookie the update request and becomes hot-spot in write intensive workloads as the number of replicas increases.

One problem with the performance of 2PC protocols is the performance bottleneck due to the increased load on the coordinator. One approach to overcome this problem is Chain Replication (CR) [28], a replication protocol for scalable, consistent and high performance storage services. It consists of an overlay chain network topology, where write or update requests from

clients are only served by the head (the first node) and are propagated to the chain. On the other hand, read requests are served only by the tail. In this way, it provides strong semantics and high performance as the load is evenly distributed among storage servers. CR relies on an external directory service for discovering current group membership whereas Zookeepers' membership mechanism imposes a static group known to all peers.

Another approach to solving performance hot-spot in a primary-backup scheme is Mencius [18] which is designed especially for WAN networks. Mensius tackles the following performance problems that arise from Paxos relying on a single leader for choosing request sequence: First single Leader CPU becomes hot-spot. Second, Leader network link bounds the total system capacity do to the tree communication pattern with the followers, and third latency is asymmetric; that is clients that are in the same site with the Leader face lower latency than others in different sites over wide area networks. It solves these problems by proposing a replicated machine protocol optimized for wide area networks with asymmetric links by distributing the leadership among protocol rounds.

Zookeeper's recovery mechanism is based on the record of messages in disk media. On the other hand, systems like Harp follow another approach to achieve replication consistency and performance by using special modules of hardware. In more details, Harp [8] is a fault tolerant file system over NFS that introduces two main mechanisms to support integrity and availability. Uninterruptible power supply supports non blocking writes and availability is supported by a primary back up replication. Harp consists of three main entities a primary server, a secondary server and a server holding a witness where each of them is connected in separate UPS. With the UPS mechanism the primary server can reply to a write operation as long as the write is in the main memory of the primary and the secondary. The witness server takes action on a node failure; the server that can contact the witness becomes the new primary ensuring the consistency of the data in the presence of both site failures and network partitions. Nevertheless, commodity hardware is a preferable solution in large scale systems due to its cost efficiency.

The idea of modular design for storage services [17] has inspired the de-

sign of Boxwood [11]. Boxwood is a system that offers a general purpose storage service. Its design follows a layered approach. In the bottom level a replicated block service is implemented and on top of it a chunk manger module is responsible for the allocation/deallocation of blocks, exported by a global block address space. The B-tree module uses the chunk manager to store/retrieve data. Also it introduces the idea of the design of the synchronization and failure detection mechanisms that are used by the system as separate services. In ZKRL design the same modular design idea is applied.

In the current thesis we use ZKRL to replace HDFS [14] replication mechanism in order to enrich HDFS semantics. HDFS is a large scale file-system such as Niobe [12] and Google File System (GFS) [29]. In particular HDFS is inspired from GFS and is the storage subsystem of the Hadoop MapReduce framework. Google File System is a storage system optimized for append operations and streaming reads.. GFS is consisted of a single master and chunk servers. Master is responsible for the entire filesystem namespace which is flat. It keeps all the metadata in memory and a mapping from file name to a list of block files; each block file is typically 64 MB. GFS can support concurrent appends because in each block write operation one of the typically 3 servers is elected as master, thus serializing the order of the updates. Read requests can be served from any chunk server. A GFS/HDFS client does not participate in the replication algorithm; when it receives a failure it is set to retry the operation until a successful outcome. GFS do not use a distributed commitment protocol and thus failure of a participating node may lead to an inconsistent replica group state that is visible to the client. This state may persist beyond the duration of the failure: file replicas may contain duplicates of data records, or be padded with zeros. Furthermore, a change in the identity of the primary replica may provide a client with a different end-of-file, leading to a skipped write at one or more replicas. Rapid reconfiguration of replica groups would reduce the window of inconsistency but it can take time as it involves the master (as the replica group is not aware of each other), which may at times be overloaded.

The major difference between GFS and HDFS is that HDFS does not support concurrent appends due to the client's design principle to act as

master in the replication protocol. Its recovery mechanism lies its correctness on the ability of the client to calculate offset for write requests. With the embedding of ZKRL into HDFS concurrent appends can be supported. Similar to this work in the direction of strengthening HDFS semantics, IBM has recently published source code for an implementation of the Hadoop filesystem interface over the IBM General Parallel File System (GPFS) [22]. They also considered evolving cluster file systems such as GPFS in the direction of providing appropriate support for the type of applications that HDFS is a good fit for [16]. These approaches are related to ours but are significantly more complex and require investment to a commercial parallel file system technology. GFS and HDFS are used as a the storage back end in the widely used applications of BigTable [9] and HBase [19].

Chapter 3

ZKRL Design

3.1 Overview of Zookeeper

Zookeeper implements a hierarchical namespace of fixed-size objects (referred to as znodes) accessed via a filesystem-like API. The Zookeeper namespace is organized as a hash-table memory structure kept consistent across a set of servers called Quorum Peers (QPs). The set of QPs is referred to as a cell. A cell is organized as a single leader and number of followers as shown in Figure 3.1. Each request (otherwise known as a proposal) towards a znode corresponds to a transaction with a specific ID (referred to as a zxid) eventually committed into a Commit Log. Consistency is achieved via a simple atomic-broadcast protocol [15]. Znodes do not map to persistent locations on disk. Instead, the entire namespace is periodically serialized and snapshotted to disk. Zookeeper can thus recover znode state by loading the most recent snapshot and running its commit log, up to the most recent committed transaction. The total amount of data Zookeeper can store is bounded by the physical memory of the least-provisioned node in the system.

The leader is connected to each follower through direct FIFO channels (implemented as TCP streams) in a tree pattern. Each leader receives proposals from clients, moves them out of sockets, forwards them to all followers and then writes them to disk. Upon receiving a proposal, a follower writes it to disk and then acknowledges with the leader. The leader commits a pro-

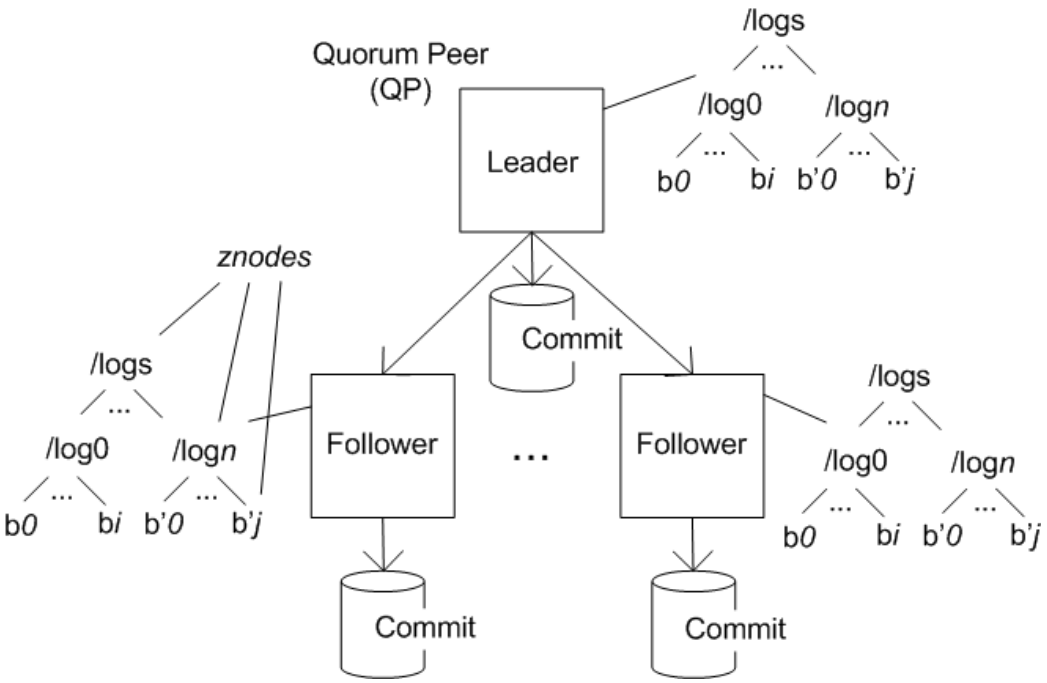

FIGURE 3.1: Zookeeper replicated state machines.

posal and responds successfully to clients only after it has received ACKs from a majority of followers. To avoid overload the leader uses an application-level flow control protocol (i.e., stop receiving data from TCP sockets) to throttle clients when the queue of outstanding (not yet committed) proposals exceeds a configurable threshold. A group commit protocol comes into action to avoid the cost of flushing dirty buffers to disk at each operation.

3.2 Persisted replicated logs

We have extended Zookeeper to produce a persistent replicated-log abstraction (ZKRL) that supports the following operations: create log, delete log, append data to log, and read data from log. Create-log is implemented as the creation of a parent znode whose name is the name of the log (e.g., /logs/log0 shown in Figure 3.1). Append-to-log is implemented as the creation of a child znode ((e.g., /logs/log0/b0 shown in Figure 3.1) using the Zookeeper API SEQUENCE flag, which picks (in agreement across replicas)

the next-in-sequence number and incorporates it in the znode's name. This essentially ensures that the appended data receives the same offset across all replicas. Read-log is implemented as the reading of a child znode at a given offset under a given parent. Delete-log is implemented as the removal of the corresponding parent znode along with all its children.

Although one can construct such an abstraction over unmodified Zookeeper, there are a number of practical limitations: First, Zookeeper does not support disk-backed znodes. Second, Zookeeper focuses on optimizing performance for small proposals (in the order of a few hundred bytes) and thus exhibits performance bottlenecks when used with large messages. We address these problems by extending Zookeeper as described below:

Disk storage for large znodes: We introduced an indirection mechanism that enables storage of large znodes on a persistent structure on disk, called the Data Log (Figure 3.2). To achieve this we store into the znode memory structure a pointer to the corresponding Data Log location (an offset into a file). On every proposal that creates a large znode, ZKRL logs the proposal content to the Data Log (prior to committing the transaction) and set the znode pointer to the offset in the file. For consistency we require that the data referred to by a zxid always be stable on disk before committing the zxid transaction. Note that since it is the data payload and not the znode itself that is stored on disk, ZKRL is still limited by physical memory on the amount of metadata for the replicated logs. Optimizations such as those described in Section 4.3.1 can reduce the amount of metadata, increasing the amount of data that can be stored in ZKRL.

Write-path optimizations: Zookeeper was originally designed for small (in the order of hundreds of bytes) proposals. It is thus not a surprise that its serialization and de-serialization mechanisms and memory copying in the data path introduce considerable CPU and memory overheads when used in large-message settings. In addition, the Zookeeper implementation does extended use of dynamic memory allocation from the Java heap, leading to frequent Java garbage collections that occasionally freeze the application. To address these issues we introduced copy-avoidance optimizations and used static memory allocation (recycling pre-allocated data buffers) to reduce the

overhead of the data path.

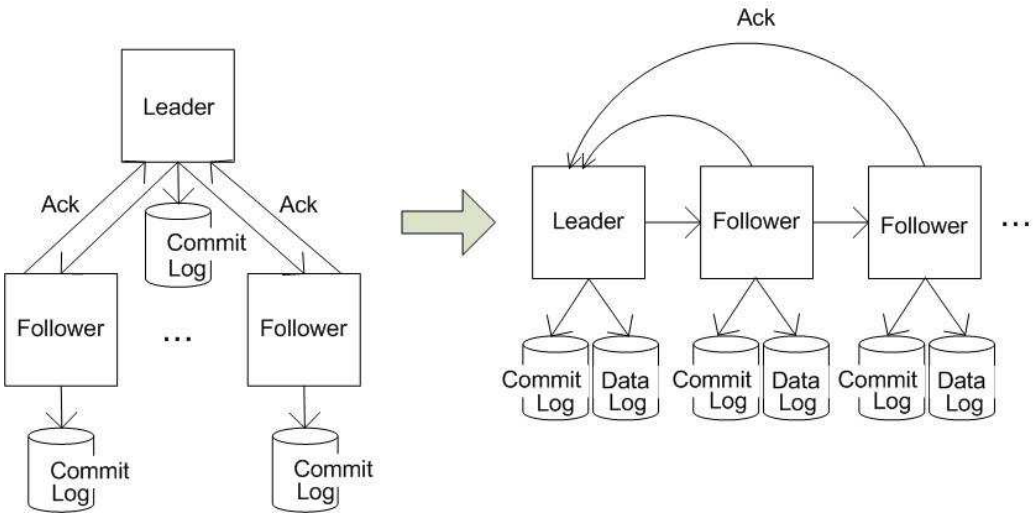

FIGURE 3.2: Transforming Zookeeper into a persistent chain-replicated log.

Chain overlay communication topology: The tree topology between the leader and the followers used in Zookeeper (Figure 3.2 left) suffers from two performance problems when used in a throughput-intensive setup: (a) The leader CPU becomes a hotspot due to having to repeat the work of sending each proposal to each follower; and (b) the effective bandwidth of the leader's output link is divided by the number of followers, putting an upper bound to total system throughput. To overcome these problems we use a chain topology (Figure 3.2 right) along the lines of Chain replication [28]. To ensure that we preserve the correctness properties of Zookeeper we ensure FIFO ordering between the messages seen by each follower even though in reality we use multiple (chain as well as direct) paths between leader and followers.

State synchronization for disk data: The Data Log complicates recovery in that ZKRL needs to synchronize disk as well as memory state. We thus extended Zookeeper's state synchronization mechanism to take into account the Data Log differences between the system state and recovering replicas.

3.3 Protocol operation

The most important departure of ZKRL from Zookeeper is the use of different communication channels and modes between leader and followers (Figure 3.2 right): (a) broadcast from leader to all followers over a chain; (b) direct communication between followers and the leader over separate connections; (c) and unicast from leader to a specific follower over the chain. In what follows we will describe operation in the absence of failures and then during recovery from failures.

3.3.1 Failure-free Operation

The leader is responsible for deciding the configuration of the chain based on its view of the membership (IP addresses and IDs of all followers in the current quorum), which it maintains as soft state. Clients communicate their proposals to the leader, who forwards them to the first QP down the chain via a PREPARE message (Figure 3.3). The message contains the proposal itself as well as the configuration of the chain, namely the IDs and sequence of QPs in the chain. Every QP (including the leader) ensures that the proposal is in-order (looking at the zxid), logs it to storage, updates the routing information on the PREPARE message, and forwards it to the next QP down the chain. When a QP is certain that a write has reached stable storage it sends an ACK for it to the leader through their direct connection. When the leader receives a majority of ACKs it commits the proposal by injecting a COMMIT message in the chain.

3.3.2 Failure Recovery

The leader periodically sends PING messages to all followers and assumes a follower is down if it receives no answer within a timeout period. It then removes the follower from its membership set and releases all resources maintained for it. Similarly, if a follower detects that the leader is down it restarts itself and initiates a new leader-election round [15]. Failure of any QP in the chain may results in loss of messages (all in-transit messages that it had in

memory), impacting QPs downstream in the chain which will receive subsequent messages out-of-order.

When a QP detects an out-of-order condition (comparing subsequent message zxids) it will restart itself, learn the leader (if there is one or try to elect one if not), and re-synchronize with it. Re-synchronization starts by sending the leader a FOLLOWER_INFO message (Figure 3.4) conveying the last zxid that the follower has seen as well as its ID. In response, the leader locks the queue that it enqueues the messages for the chain (FFQ) and sends the follower a NEW_LEADER message containing the last zxid that the leader has seen and FollowerEpoch, a monotonically increasing value (soft state) indicating the incarnation of this follower during the same leadership. A QP will reject messages (regular or synchronization) that belong to a past epoch. In response to the NEW_LEADER message, the follower sends the leader an ACK message to indicate that it is now ready to receive state. This is needed to convince the leader that the follower is ready to receive synchronization messages out-of-band over the chain.

The actual leader-follower exchange during state synchronization depends on the amount of state that needs to be transferred. The leader decides this based on the relation of the follower's zxid to the window of recent transactions [zxid_min, zxid_max]:

1. If the follower zxid is less than the leader's zxid_min, the leader serializes its in-memory hash table structure in a SNAPSHOT message and sends it to the follower over the chain. The follower de-serializes the SNAPSHOT message and installs a consistent hash table in its memory. This snapshot may include pointers to on-disk data that do not yet exist in the follower's storage. This is because of znodes with zxid greater than the latest zxid that the follower has seen. To transfer those files to the follower we use a special RESTORE message (an extension to the Zookeeper protocol). After sending SNAPSHOT, the leader iterates over the hash table structure and finds all znodes whose zxid is greater than the follower's latest zxid. It then sorts those znodes in increasing zxid order, prepares RESTORE messages with a pointer (file,

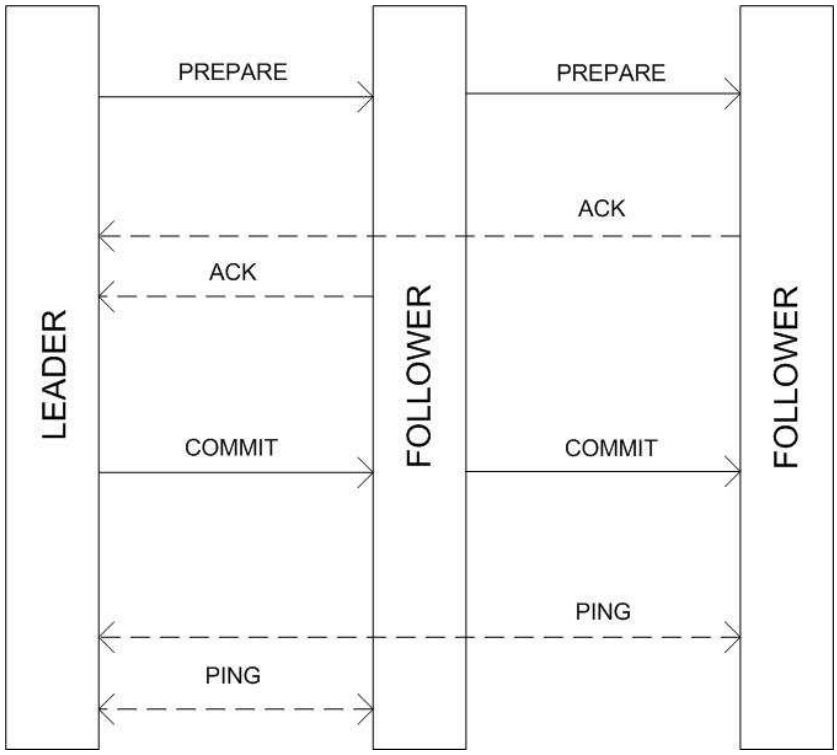

FIGURE 3.3: Failure-free operation. Solid lines indicate broadcast communication over the chain; dashed lines indicate communication over the direct leader-follower connections.

offset, size) to the on-disk data as well as the actual data, and sends them over the chain. Afterwards it sends possible outstanding proposals and commits via PROPOSALTO and COMMITTO messages. These may have appeared in the time gap between the arrival of the FOLLOWER_INFO message and the locking of the FFQ.

2. If the follower's last zxid is within [zxidmin, zxidmax] then we consider that the follower has not lost much state and can be synchronized by just replaying the specific set of missing proposals. The leader sends the follower a DIFF message and then all proposals that the follower missed via unicast PROPOSALTO and COMMITTO messages. These messages (also extensions to the Zookeeper protocol) are functionally equivalent to PROPOSAL and COMMIT with the difference that they

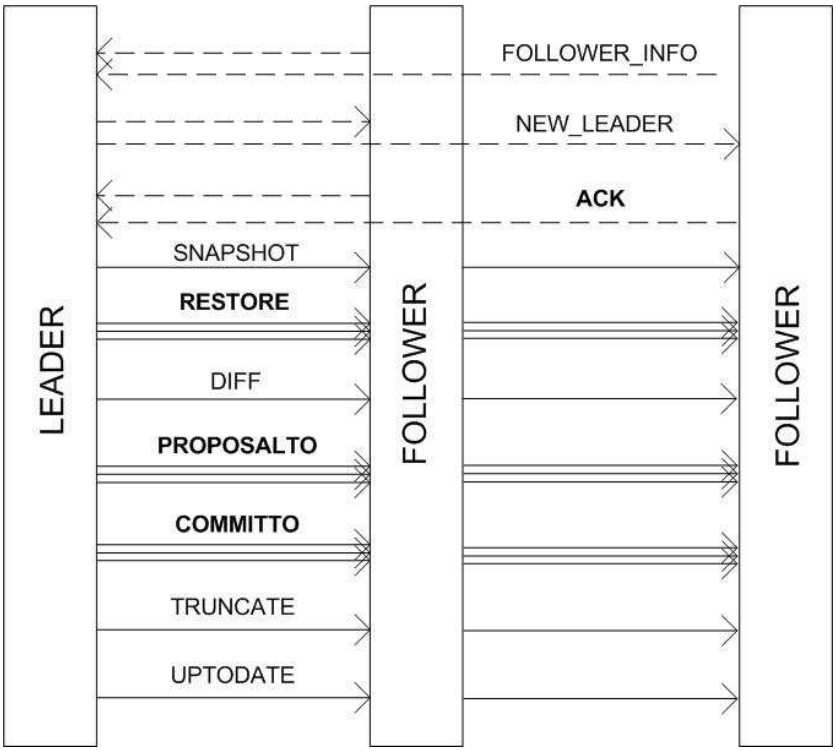

FIGURE 3.4: Protocol operation during recovery. Solid lines indicate unicast communication over the chain, dashed lines indicate communication over the direct leader-follower connections. In bold are our extensions to the original Zookeeper protocol.

are sent over the chain to a single recipient. Just as in the previous case the leader includes data from (file, offset, size) into the DataBlock field.

3. If the follower's last zxid is greater than zxidmax (i.e., the follower has seen more proposals than the leader) the leader sends a TRUNCATE message to the follower, who truncates its logs to the leader's zxid.

Finally, the leader sends an UPTODATE message to the follower to indicate that it can resume processing. The possibility of chain failures before a QP is fully synchronized are complicating recovery and their handling is discussed in Section 3.4.5.

3.4 Implementation

This section focuses on implementation details. We begin by describing the format of protocol messages exchanged between QPs, then provide a close look at our data path implementation and optimizations, and finally describe our failure handling in detail.

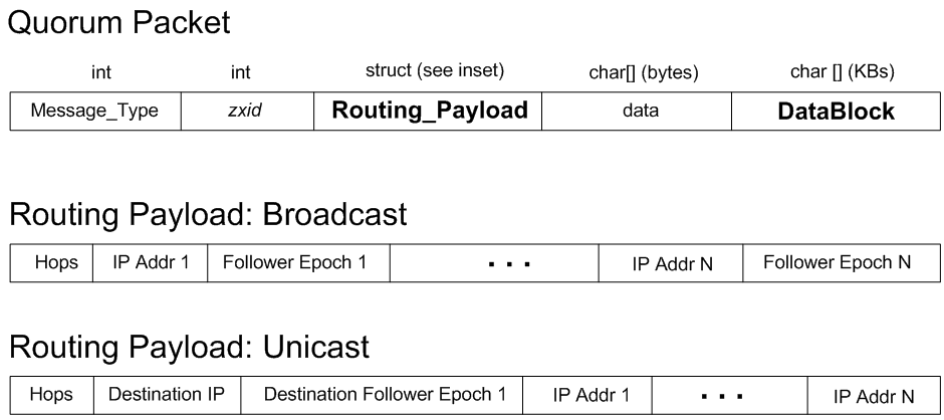

FIGURE 3.5: Quorum Packet. Our extensions to Zookeeper are shown in bold.

3.4.1 Quorum packet structure

The QuorumPacket (Figure 3.5) is the format of messages exchanged between leader and followers. The basic fields of QuorumPacket include the type (as described in Section 3.3); the proposal zxid; the data to fill-in the in-memory structure (order of bytes); the chain configuration (**Routing-Payload**, a ZKRL extension); and the data chunk to be written to storage (**DataBlock**, a ZKRL extension). The structure of RoutingPayload depends on whether it refers to a broadcast or a unicast message. For broadcast messages it consists of: **Hops**, an integer decreased by one at every QP that forwards the message (the message is no longer forwarded when hops reaches zero); **IP_addr[i]**, the IP address of the next QP, where i is the current QP; and **FollowerEpoch[i]**, indicating the FollowerEpoch for the QP at hop i. For unicast messages, RoutingPayload consists of: **Destination Follower**

IP, the IP address of the recipient (the message is no longer forwarded once it reaches its intended recipient, independent of the Hops field); **Destination Follower Epoch**, the FollowerEpoch of the intended recipient; **Hops**, the distance to the intended recipient; IP addr, the IP address of the next QP to route the message to.

3.4.2 ZKRL Data Path

QPs in a ZKRL cell are interconnected via TCP channels between the leader and each follower and also over a chain formed via QP-to-QP connections, as shown in Figure 3.6. TCP connections between QPs in the chain are persistent across proposals. After a failure, the failed node is bypassed by establishing a new TCP connection with the next QP. ZKRL extends Zookeeper's znode structure with a data field, which points to a persistent location within a Data Log. The data field contains the Data Log filename, an offset into it, and the data size.

ZKRL inherits a staged event-driven architecture (Figure 3.6) from Zookeeper. Each QP is structured as a sequence of Request Processors that operate in a pipelined fashion. Each RP is associated with a thread that performs operations on its input passing over the result to the next RP via a shared FIFO queue. The requests are passed by reference through the queues.

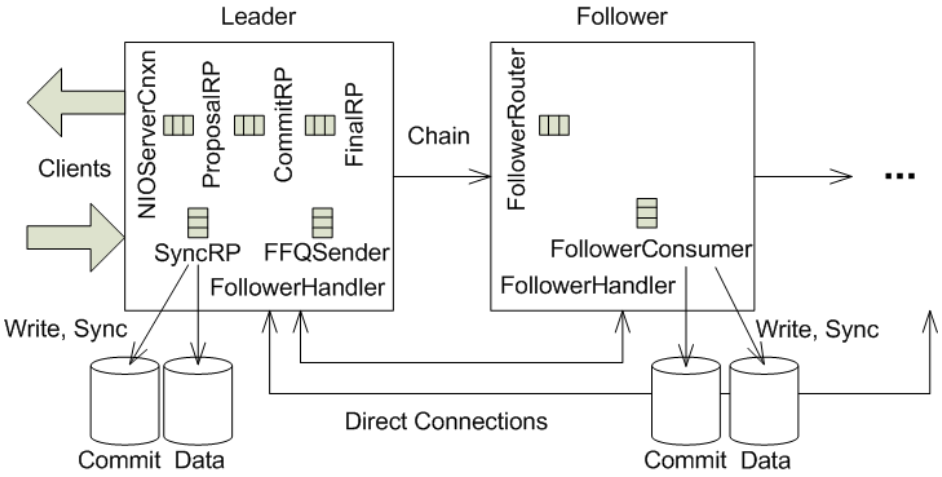

FIGURE 3.6: ZKRL data paths

The lifecycle of a proposal (consider a log-append as an example) comprises the following steps: The client sends a request to the leader by invoking the CreateTxn API. A thread (NIOServerCnxn) allocates a new buffer in which it copies the proposal's data payload. The leader may decide to throttle clients if it is running low on the number of available pre-allocated buffers. The leader queues the request into the FIFO queue of the PrepRequestProcessor stage. The PrepareRequestProcessor encapsulates the client request into a Quorum Packet and queues it into the queue of ProposalRP which run in the same context. ProposalRP will broadcast or unicast the quorum packet over the chain using another thread (FFQSender). Prior to sending the packet, ProposalRP marks RoutingPayload with the IPs and Epochs of the QPs in the chain. FFQSender parses RoutingPayload, decreases Hops by one and tries to send the message to the next peer. The proposal is then passed to SyncRP, which appends it to the Commit and Data Logs. Following the writing of the Data Block to the file system, SyncRP inserts the associated pointer (filename, offset and size) to the in-memory hash table. SyncRP periodically flushes written proposals to the disk using group commits. SyncRP passes the proposal on to CommitRP, which is responsible for counting ACK messages sent by QPs for this proposal. QPs may ACK a proposal only after they ensure that it has been successfully flushed to disk. When the leader receives a majority of ACKs it sends a COMMIT message to all QPs over the chain and the proposal is applied on the receiving of the COMMIT message. When a proposal is committed, the FinalRP stage sends a reply back to the client.

Followers have a simple structure in which a FFQRouter thread performs equivalent tasks to the leader's FFQSender. FFQRouter is spawned when the follower receives a NEW_LEADER message and is responsible for listening for new incoming TCP connections and for propagating messages in the chain. It passes its output to the FFQConsumer thread, which decides if this message should be consumed or not. If it is a UNICAST message it checks the IP and FollowerEpoch fields. If they both match its own, it consumes the message, otherwise it forwards it. If it is a BROADCAST message the FFQConsumer checks the FollowerEpoch field. If the message

is from a previous epoch (i.e., stale), it rejects it, otherwise it consumes it. If the message is a PROPOSAL or PROPOSALTO message it appends the DataBlock to the Data Log and the metadata in the Commit Log and then sends an ACK message to the leader through the direct connection. If the message belongs to any other category it appends it to the commit log and performs the appropriate proposal-specific action to it.

Upon receiving a RESTORE message the follower QP attempts to locally open the file indicated by RESTORE and creates it if it does not exist. It then seeks into offset and writes into it the data from RESTORE's DataBlock field. If a follower needs to restart before the process is complete, it will disregard all stale RESTORE messages in its next incarnation by means of the FollowerEpoch field. Finally, a FollowerHandler thread (an instance of which is spawned by the leader for each direct connection) is responsible for handling the messages corresponding to the dashed lines in Figure 3.4, except ACK which is handled by CommitRP. FollowerHandler is also responsible for the follower QP bootstrap and state synchronization protocols.

3.4.3 Optimizations

We have introduced a number of language (Java), runtime, and operating system specific optimizations to the base Zookeeper code to enable a high-throughput write-intensive replicated log system. One such optimization is the pre-allocation and subsequent management of a pool of large direct memory buffers [1] to avoid data movement costs in the JVM as well as frequent heavy-duty garbage collector invocations that significantly impact performance [17]. Each time the application needs a buffer for Quorum Packet's DataBlock field it tries to use one of the pre-allocated buffers. If none are available, it waits until a buffer becomes available and throttles clients by stopping to receive data from TCP sockets. This throttling mechanism differs from (but is equivalent to) Zookeeper's original mechanism based on a threshold to the number of outstanding (not yet issued plus not yet committed) operations.

Another optimization focuses on reducing data movement costs (such as

unnecessary memory copies), which result into wasted CPU and memory bandwidth and thus decreased system throughput. Our methodology was to replace the tree communication structure by a chain and all memory copies between Request Processors (Figure 3.6) by exchanges of buffer references. With these modifications data are undergoing a minimal number of copies (3) in the application space: One copy from the TCP socket buffer into an application buffer, another one into the output TCP socket, and a third copy into the filesystem buffer cache. After that, the buffer can be returned to the pool if allocated for it (e.g., if it was a large buffer) or left to be reclaimed by the garbage collector. Further reducing data movement costs in the I/O path requires specific system support [13][7] that is not widely available in commodity environments.

Writes to the data log take use the group-commit mechanism that Zookeeper applies to the commit log. To achieve high I/O throughput the system needs to ensure efficient operation all the way to the disks. In other words, in a heavily write-intensive setup the filesystem buffer-cache should be continuously writing to disks to avoid stalling for too long at the periodic sync operation. The standard operating system setup (in Linux and other general-purpose operating systems) is to delay writes. We thus had to change the standard behavior by tuning the pdflush thread (responsible for de-staging data from the buffer cache to disk) to be invoked whenever there is anything in the buffer cache to be written (by appropriate tuning of the dirty_background_ratio kernel parameter). Note that this optimization requires platform-specific knowledge and is thus testimony to the fact that despite improved support [1] it is not always possible to achieve fully platform-independent systems software [17] [24] in Java alone. Besides choosing an aggressive cache-to-disk de-stage policy we also decouple write and sync operations to separate threads of control. In the initial version a SyncRP thread (Section 3.4.2) groups write-requests and flushes them to disk if they reach a threshold of 500 or if it is otherwise idle. Our optimization was motivated by the first observation that serializing write() and sync() operations within SyncRP interrupts the write pipeline when waiting for completion of sync() and thus reducing overall disk throughput. The second observation is that in

a ZKRL Quorum of size greater than one the followers and especially the last follower in the chain which has the lowest CPU load among the team issue more frequent sync operations than the leader due to the sync algorithm of Zookeeper as described above. In our approach the SyncRP thread writes proposals to disk whereas a newly introduced FlushRP thread issues sync() calls on groups of proposals or when a control message explicitly forces a sync() In this approach the whole quorum issue sync operations to almost the same number of proposals.We use the latter technique to ensure rapid commitment of latency-sensitive operations.

Writes to the Data Log must take place before the corresponding writes to the Commit Log. To ensure that the actual writes to disk preserve the order of the issued write() operations is a challenging task in an environment where the application does not control when the file system's buffer cache is de-staged to disk. Thus to ensure ordering we treat writes differently between the two logs: in writing to the Data Log we invokes the write() call. However, in writing to the Commit Log we defer the actual write() and instead store the corresponding data into an application buffer. When flushing to disk, we first sync() the Data Log and afterwards write() and sync() the Commit Log.

There are two policies that we can use for the Data Log: Implementing it as a single file (along the lines of pure log-structured designs) or as a set of files (file per proposal or file per group of proposals). We have experimentally tested that the former approach is nearly always superior to the latter and therefore use it as the default layout. We also believe an extent-based file system such as xfs is a good match for the type of workload produced by ZKRL.

Finally we had to increase the TCP window size from its default value in Linux kernel and increase the socket buffers. Before this optimization we noticed not optimal performance due to TCP flow control.

3.4.4 ZKRL Read Path

The read path of ZKRL evolves these steps. The client that wants to read a file A at offset C can calculate the name of the znode that contains the pointer to the data without issuing a get children operation which is expensive especially for big logs because it must fetch the names of all the children of a znode. By diving C with B (proposal size) and keeping the integral part of the division, say znode_id it can construct the name of the znode by doing a concatenation of A|znode_id.

If we don't keep a znode for each proposal, due to memory space optimization, we can keep a znode for a log after it is closed as described in details in 4.3.1, then we must expand the Zookeeper client api to serve read requests. We could introduce an operation read(file, offset) where the zkServer lookups it's internal tree structure to locate the corresponding znode. For each znode we keep it's total size in disk so the zkServer can decide if the client offset is valid inside the file.

3.4.5 Failure scenarios

We consider two cases: leader failure and follower failure. All nodes are able to detect a leader failure (via PINGs) and kick system re-configuration via electing a new leader and synchronizing global state. A follower failure will be detected by the leader (also via PINGs) and by the follower that is upstream in the chain, since the FFQSender or FFQRouter will fail to forward messages downstream. In such cases the sending node will repeatedly try to send until timing out. Subsequently, it will decrement the Hops field and forward the message to the next node in the chain, bypassing the failed node. If that node also seems to be down, the sending node will repeat the procedure (that is, try the following node down the chain) until the Hops value reaches zero. When the leader detects a failed follower it removes its IP from the routing vector (RoutingPayload), taking out the node from the chain topology.

It may be that messages carrying the old chain configuration are still in the chain after the leader has decided to reconfigure it. This is not a problem since the node that is upstream of the failed knows how to bypass the failed

one. If the failed node restarts, it will be assigned a new Follower Epoch by the leader and thus will reject proposals from a previous epoch. Another issue is with the proposal messages that the failed node was processing and thus took down with it. Followers that are downstream from the failed node in the chain may receive a zxid x without having seen a proposal with zxid y for x $>$ y. Once a node detects violation of the FIFO condition it restarts so as to re-synchronize with the leader. Finally, a related failure case is when a follower receives proposals prior to completing state synchronization (i.e., before receiving an UPTODATE message). In this case synchronization messages must have been lost and the follower should restart to re-synchronize.

The follower takes a snapshot when it receives an UPTODATE message. Every a time a QP loads the ZK tree from its snapshot and replays its commit log, it calculates the total size of each log kept on disk. It then compares with the actual file size and if necessary truncates the logs to the size pointed by the ZK-tree.

Chapter 4

Integration of ZKRL with HDFS

4.1 Overview of HDFS

HDFS nodes can be either a namenode or a datanode. A namenode is responsible for maintaining the file system namespace, the file-to-block mapping, and the location of replicas. Datanodes store replicas of blocks as local UNIX files. To access a file, a client contacts the namenode to get a file handle and information about the names and locations of block replicas. The default block size is 64MB. Each block consists of 1024 64KB packets. HDFS replicates blocks of a file using a configurable number of replicas. An HDFS replica group for a new block is decided by the namenode and organized (e.g., connections setup between the datanodes) just prior to the client accessing it.

The standard HDFS replication mechanism works as follows for a write: Once the client has the list of block replicas it contacts the first datanode in the list, which contacts the second node in a pipelined fashion all the way to the last node. After the chain is set up, every packet written to the block is mirrored across all participating datanodes. Reads are satisfied from any (usually the nearest to the client) replica.

Each datanode dedicates a thread (the receiver) to handle incoming packets and another thread (the responder) to send acknowledgments to the sender of each packet (a datanode or the client). The receiver's actions

for each packet are as follows: it reads the packet from the network into a memory buffer, places an ack for it in the responder's input queue, forwards the packet down to the next datanode in the pipeline, computes a CRC on the packet, writes the CRC to the CRC file and appends the packet data to the block file on disk,. The receiver executes these actions in a loop for all packets of a block. The responder's actions are as follows: once an ack from a downstream node is received, it forwards the ack to its upstream node in the pipeline. As a result, when the client receives an ack for the nth packet it is certain that the packet lies on the application buffers of all datanodes.

HDFS's replication consistency model works as follows: If a datanode crashes before the block is closed the error will be propagated back to the client and the pipeline between the datanodes and the client will close. The client will contact the namenode to start block recovery and the namenode will increase the version of the block and send it back to the client. The client keeps two queues, one with packets that have not been propagated to the replica group, and another with packets that have started propagating but not yet acked. The client will contact the replica group using the new block version retransmitting packets in the unacked queue. Each packet has an associated offset calculated by the client. This is to ensure that retransmission of packets that may already have been written does not cause duplicate writes (in other words writes are idempotent). This is however a reason why HDFS in its current form cannot support concurrent writers as for example GFS [29] does. It guarantees exclusive write-access to a file by not issuing a new write lease until either the file is closed or the previous lease has expired and its owner has not renewed it.

However readers are allowed to read the last block of a file and thus may see different results temporarily in case of a datanode failure if HDFS is in the middle of its recovery process. If the client and/or a datanode crash, the inconsistency may last much longer because a new writer will have to wait until the lease of the previous writer expires. Following expiration of the write lease the namenode will wait for a fixed amount of time and will then close the file so it could be available for future read/write operations. Each datanode will report to the namenode the size of the last (not yet closed)

block that they have on disk. The namenode will pick the minimum among the reported sizes and ask datanodes to truncate their copies to that size and increase their version of the block. A recovered client will discover the current end-of-last-block by contacting the namenode and will start writing from that point on. The last block of a file is not guaranteed to be consistent across replicas until closed (i.e., all replicas have received the entire block).

4.2 Benefits of integration

Integrating ZKRL into HDFS as an alternative replication mechanism provides more intuitive semantics at reduced complexity. Another structural benefit to HDFS from this integration is the offloading of replica management from the namenode. Activities such as replica leader election, synchronization, and organization can be taken care of by each ZKRL cell itself and need not be performed by the namenode, reducing its workload. This is especially important in architectures with a single metadata node such as GFS and HDFS. The metadata node need only be responsible for the allocation of file blocks to ZKRL cells and the identity of the primary node in each cell (to which each client should be talking for reading/writing blocks). Policies for performance and high availability at the block level (such as placing replicas in different racks, otherwise known as rack-aware placement) can now be encapsulated into ZKRL. Finally, the ability of HDFS users to choose the replication factor for a file can be extended to ZKRL by enabling different quorum sizes for each cell.

4.3 Implementation

Our integration of ZKRL into HDFS has minimal impact on the HDFS design. The on-disk data structures as well as the read path remain unchanged, in other words HDFS blocks are still created in the same on-disk locations and with the same format as in the original HDFS, which is critical for interoperability with other HDFS functions (such as the block report daemon).

The write path however now goes over ZKRL. We decided to map an HDFS block on a ZKRL log. Each HDFS packet maps to a ZKRL PROPOSAL, which expands the block's size we have.

HDFS keeps CRCs for every block file. In our integration we disabled HDFS CRCs in order to separate the performance impact of ZKRL to that of the CRC mechanism, which is orthogonal to our design. In our implementation the HDFS client still computes CRCs and sends them to the datanode but the datanode ignores them. We also slightly altered the DataBlockScanner, a daemon that periodically scans a datanode's disk and reports to the namenode the blocks that are available in the datanode as well as if they are corrupted by checking their CRCs. Our modification was to not compute CRCs or report bad blocks to the namenode.

ZKRL is initiated during HDFS datanode bootstrap. The namenode knows only the datanode that is the ZKRL leader and has no interaction with ZKRL followers, which also interact only with the ZKRL leader. If the datanode (ZKRL leader) fails, the new leader will boot the datanode code and will contact the namenode reporting the block files that lie on its disk. All the read and write requests are served only by the datanode. The client sets the replication factor to one to prevent HDFS from creating a standard replica group using its own mechanism.

On receiving a new client request the datanode parses the HDFS protocol headers and creates the appropriate files in its configured directory exactly as in the original HDFS with the only difference that CRCs are now disabled. After parsing the HDFS Datanode does not write to disk directly (as in the standard HDFS implementation) instead forming a ZKRL request using as metadata the filename and the data payload and enqueueing it to the ProposalRP queue bypassing the NIOServerCnxn thread. For allocating data buffers we leverage the static pre-allocated buffer mechanism of ZKRL. If there are no buffers left the thread is blocked until a buffer becomes available. The number of the pre-allocated buffers set the threshold after which the Datanode throttles client requests. ZKRL operates as described in Section 3.4.2 with the only difference that after committing a proposal, FinalRP frees the buffer it has allocated for this request but does not respond directly to

the client (since HDFS handles that).

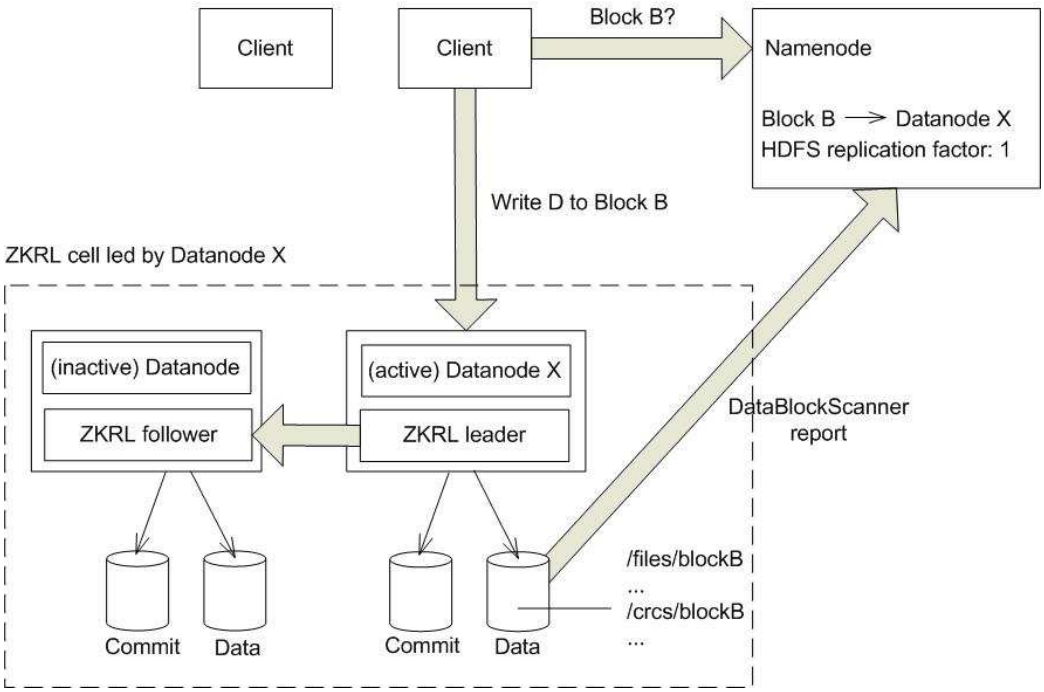

FIGURE 4.1: HDFS - ZKRL integration.

Each ZKRL cell in HDFS stores blocks (64MB) that grow in packet-size (64KB) appends. Each block consumes a (parent) znode and each packet another (child) znode. Looking at the example in-memory zk-tree of Figure 4.2 we see the root of the ZK tree at height; a znode for every block in the cell (assume N is the total number of blocks) at height 1; and a znode for each packet appended to the block at height 2. Height-2 znodes contain (besides the zxid of the proposal, which is common to all znodes), the offset inside the file and the size of on-disk data for this proposal. Assuming block size B and proposal size P bytes we have B/P children for every znode at height 1. For a concrete calculation, assuming block size of 64MB and proposal size of 64KB (1024 children for each node at height 1) each block requires about 140KB of meta-data. This means that 200MB of physical memory can support about 182GB on disk. To improve this ratio we use the technique described in the next section.

4.3.1 Compressing the ZKRL znode tree

Our solution to the problem of compressing the size of metadata is to remove height-2 znodes altogether after the finalization of a block in the datanodes, summarizing and storing their information at parent znodes at height-1. Our technique is based on the observation that every node at height 1 has necessarily the smallest zxid between all its children at height 2. This is because the node at height 1 was created first and children at height-2 are added in future proposals. Also, the rightmost child has the largest zxid across all siblings and their parent. Thus instead of creating a new znode at height 2 for every proposal that expands the block, we can keep the latest zxid of the proposal that expanded the block and the current size of the block at height-1 as shown in the example of Figure 4.3.Through this mechanism about 182TB of data on disk can be addressed via 200MB of physical memory at each cell, which is considered adequate for most uses.

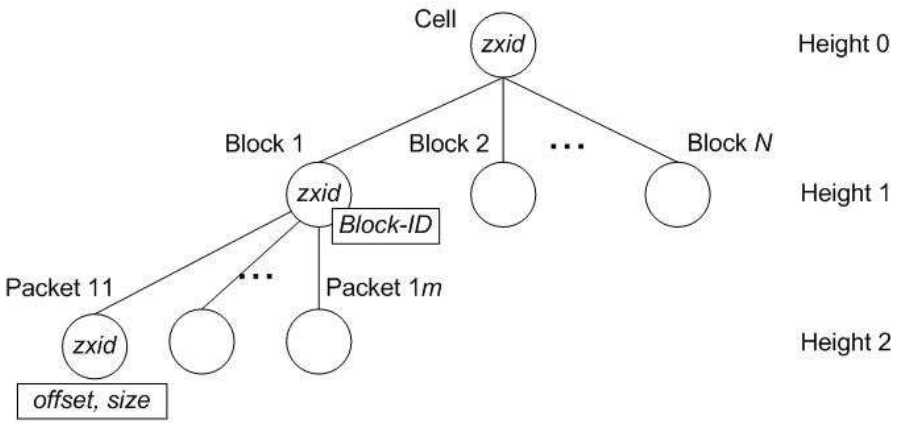

FIGURE 4.2: ZKRL metadata in HDFS.

This optimization requires small modifications to the replica synchronization mechanism described in Section 3.3.2 and may result in increased state-synchronization time in the worst case. To introduce these modifications by means of an example, suppose that a follower has seen proposals up to zxid 6. The leader (after sending the snapshot to the follower) knows that the follower has the entire first block (because $6 > 5$). It also knows that the

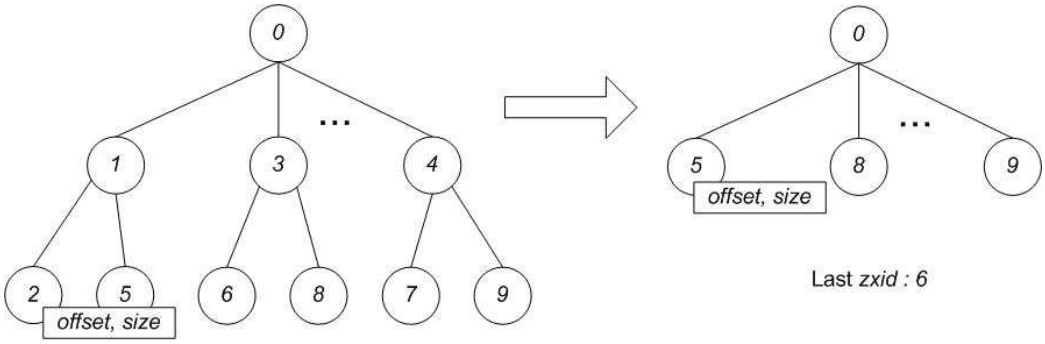

FIGURE 4.3: ZKRL tree compression in HDFS.

follower has not seen the entire second block (because $8 > 6$). Suppose in this example that the follower has missed proposal 8 only—however due to the compression the leader does not know exactly which segments of the block the follower is missing. The leader thus has no alternatives except for sending the entire block. For the third block the leader holds $9 > 6$ so it also sends the entire block to the follower. The drawback of our compression methodology thus is to send more data to the follower during the resynchronization phase than it is actually missing. Note that the read path is not affected by this change since the default read path bypasses ZKRL altogether. The worst case could happen when multiple clients are concurrently writing to different blocks in a file, resulting in scattering consecutive zxids in different branches at height 1 of the tree.

4.3.2 HDFS read path modifications

HDFS allows concurrent writer and readers. In order to fully inherit the semantics of ZKRL in HDFS-ZKRL a minor change must be applied in HDFS read path. Each datanode before issuing a read operation it must first look up it's zk tree to guarantee that the data to be read are commited by all the followers. This must be done in order to exclude the scenario where a proposal has been appended to disk but not commited yet. Datanodes in hdfs keep an in memory hash table structure of the blocks that are kept in this datanode. In hdfs-zkrl integration this structure could be replaced

entirely by the zk tree.

Chapter 5

Evaluation

We run our experiments in two sets of machines. The first set of machines is commodity hardware whereas the second set lies in the category of high performance.

The first set, which we will refer to as 1GBPS setup, is consisted of dual socket CPU AMD Opteron 1.6GHz servers with 2GB memory connected over a 1-Gbps switched Ethernet network using Jumbo (9000-byte) frames. Each server runs Linux Redhat Enterprise 5 with kernel version 2.6.18 and is equipped with three 500-GB Western Digital WD5001AALS-00L3B2 SATA-II disks, in addition to their system disk connected to a PCI-X adaptec SATA controller. Two of these disks are organized as a RAID-0 array using the Linux md driver and are used to store the data replicas in all systems. The third disk is used as a commit log in the case of HDFS-ZKRL. The RAID-0 array measured with the dd tool for raw sequential write IO reaches a throughput of 186 MB/sec whereas the commit log disk reaches 60 MB/sec. In our experiments we use the xfs filesystem on top of these disks, with nobarrier option enabled and block allocation size of 512MB. We run iozone benchmark on top of RAID-0 array and the commit log disks for sequential write workload, writing a 8GB file in records of 64KB issuing a sync operation before file close. With this setup we measured a throughput of 181 MB/sec for the RAID-0 array and 56 MB/sec for the commit log. Each server is equipped with two 1-Gbps network interface controllers (NICs) based on

the Tigon3 chipset with checksum offload enabled. We use both NICs –one sending and another receiving data— to achieve a maximum full-duplex rate of 107MB/s measured using iperf or 104.5 MB/sec measured using ttcp [33]. We chose this configuration to compensate for a performance hit when using a single NIC in full-duplex mode in our servers (limited at about 80MB/s).

The second set of machines, which will refer to as 10GBPS setup, are dual socket CPU quad-core Xeon Servers running at 2.26GHz with hyper-threading technology (resulting in 16 logical cores), 12GB of main memory and are connected over a 10Gbps switched network. Each server has a 10Gbps intel card with jumbo frames enabled that measured with ttcp gives a performance of 1117 MB/sec. The disks are the same used in the the first set connected to the on-board SATA controller. Each Xeon Server runs Centos 5.5 with 2.6.18-164.el5 Linux kernel. The md RAID-0 array measured with dd tool for raw sequential write IO gives a performance of 164 MB/sec whereas the commit log disk gives 82 MB/sec. In our experiments we use again the xfs file-system mounted with the same parameters as in the previous set. Measured with file-system benchmark iozone, for sequential write of a 24GB file written in records of 64 KB including a sync operation before file close, RAID-0 array gives 157 MB/sec whereas the commit log disk gives 78 MB/sec.

For ZKRL we use a custom java client application based on the zookeeper client middleware that appends to a log file. For HDFS evaluation we use the standard DFSIO [3] benchmark that tests the I/O performance of HDFS. It does this by using a MapReduce job as a convenient way to read or write files in parallel. Each file is read or written in a separate map task, and the output of the map is used for collecting statistics relating to the file just processed. The statistics are accumulated in the reduce phase, to produce a summary. In our evaluation we run DFSIO write benchmark for one file.

In HDFS experiments we evaluate hdfs-crc, hdfs-nocrc and hdfs-zkrl. Hdfs-crc is the unmodified HDFS, hdfs-nocrc is HDFS with crc disabled; for each block file a crc file is created of zero length. Hdfs-zkrl comes from the integration of HDFS with ZKRL as we described in chapter 4 and does not keep crc as hdfs-nocrc. The replication factor of hdfs-crc and hdfs-nocrc

is 3 and for hdfs-zkrl is 1 but the block is replicated to a ZKRL cell of size 3.

5.1 Performance of ZKRL

5.1.1 Write Performance

We deployed ZKRL in 1GBPS setup for the following experiments. In our first experiment we measure ZKRL write throughput under various proposal sizes with a quorum of size three. In this experiment ZKRL client creates a log file and appends a 100MB file for 1KB proposals, 800MB file for 8KB proposals and 8GB file for 64KB, 128KB 256KB and 512KB proposal. ZKRL syncs proposals to disk and followers reply with an ack message to ZKRL leader when the number of proposals reach a threshold T or instructed from the client to force sync. We call the T parameter as group commit window size. The leader window refers to outstanding proposals, i.e. proposals for which the leader has not yet received an ack message of the majority. The size of the leader window after which the ZKRL leader stalls the client equals two times the group commit window size. The group commit window is 1500 for proposal sizes 1KB, 8KB, 64KB and 128KB, 1200 for 256KB and 800 for 512KB.

Figure 5.1 depicts the overall throughput in terms of MB/sec bandwidth (left vertical axis) and operations per second or IOPS (right vertical axis). Reported figures are averages over five runs. Small proposal sizes (1KB) feature high IOPS, low bandwidth, and a saturated CPU around 180% (out of 200) due to the high per-I/O overhead. As the proposal size increases we observe an increase of throughput reaching a plateau of about 95 MB/sec after a proposal size of 64KB with a corresponding decrease in CPU utilization around 173%.

In the second experiment that is depicted in Figure 5.2 and Figure 5.3 we measure response time and throughput, for variable leader window size, for ZKRL quorum size of 3 and proposal size of 64KB. Response time is measured as the time gap between a client request and the arrival of the ack message from the leader. In Figure 5.2 we have the full ZKRL system where

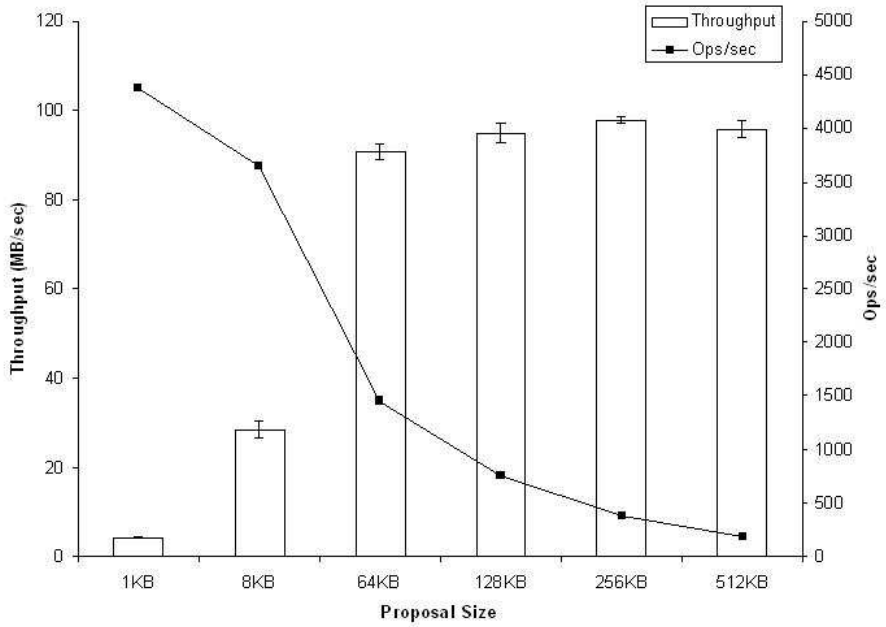

FIGURE 5.1: ZKRL throughput with variable proposal sizes.

in Figure 5.3 we have ZKRL mem, that is the ZKRL system without any disk interaction. The group commit window size is half the leader window size. As we can see for ZKRL sync with leader window of size 2 we have a response time of 19,2 msec, a throughput around 9 MB/sec and low CPU utilization. As the leader window increases the response time, throughput and CPU utilization increases. The size for which we have the minimum response time (283 msec) with the maximum throughput (86 MB/sec) is for leader window 600. After that value as the leader window increases the overall throughput slightly increases around 3 MB/sec and the response time reaches from 500 msec up to 950 msec for 3000 window. In ZKRL mem we measure a response time of 4,8 msec for leader window 2 and for leader window 200 we reach throughput of 96MB/sec for the minimum possible response time of 78 msec.

To identify the impact of different ZKRL components (memory-only vs. full-path) and quorum sizes on performance, we compare 8 different ZKRL configurations: ZKRL-memory and ZKRL-sync at quorum sizes 1, 2, 3, 5, respectively. In ZKRL-memory nothing is written to disk whereas ZKRL-

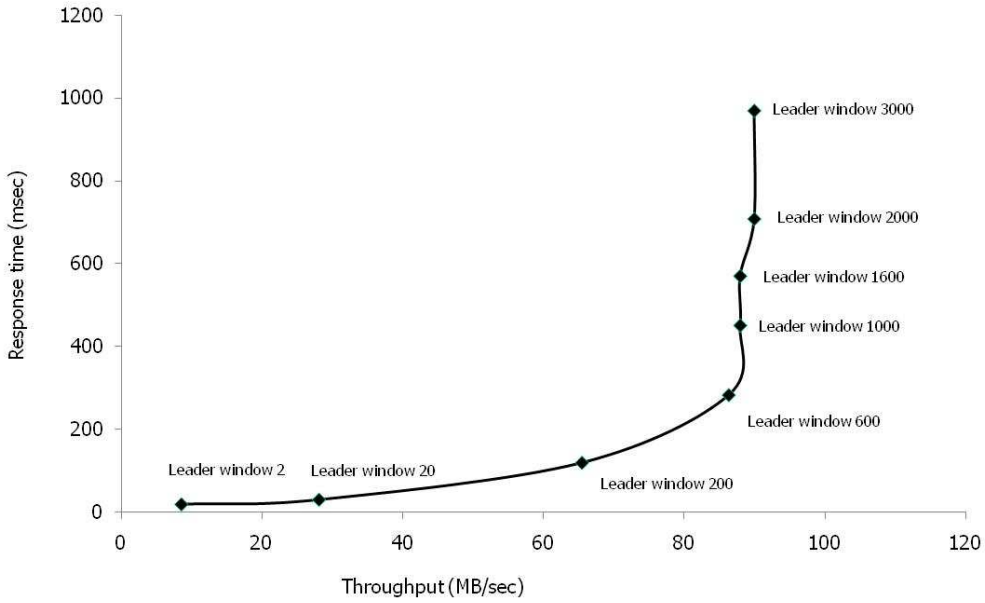

FIGURE 5.2: ZKRL sync response time and throughput for variable leader window size.

sync represents the full ZKRL system. The client has identical behavior to the previous experiment appending 8GB of data to the log in 64KB proposals with a sync operation issued every 1500 proposals.

Looking at quorum size 1 in the memory-only configuration of Figure 5.4 we see that ZKRL is able to saturate the network link consuming 75% of total CPU. The full ZKRL path in the same quorum gives slightly less throughput (106MB/s) at 147% of total CPU. We were able to achieve comparable throughput in the latter case only after decoupling the write and sync activities in SyncRP as described in Section 3.4.3. Without this decoupling the observed throughput drops to about 88 MB/sec. The original Zookeeper syncs to disc when the number of dirty requests reach a threshold or the SyncRP thread becomes idle causing uncoordinated sync operations between the nodes. For synchronization of sync operation between the nodes we altered this algorithm so each node syncs only after it reaches a minimum configurable number of proposals. Clients through a special message can explicitly issue a sync operation. A quorum size of 2 yields a throughput of

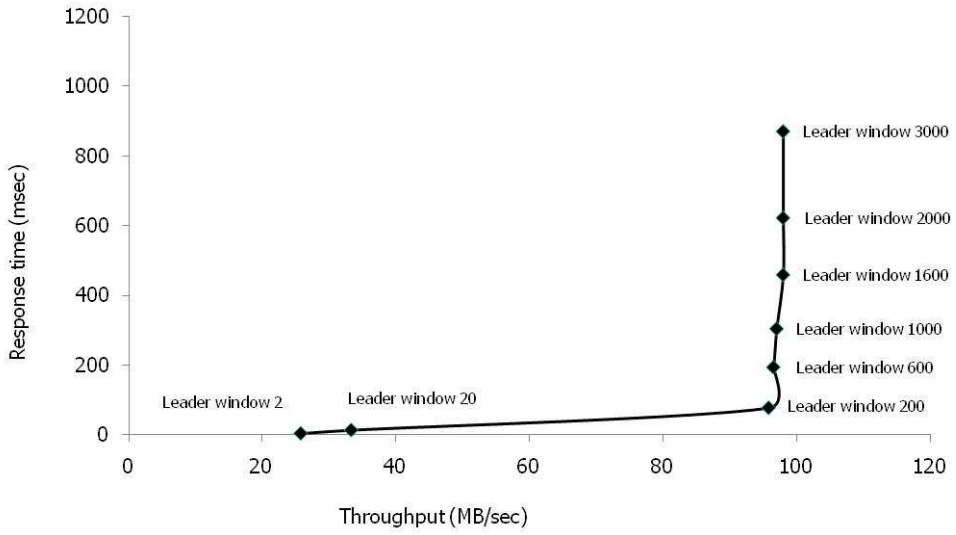

FIGURE 5.3: ZKRL mem response time and throughput for variable leader window size.

98MB/s (memory) and 93MB/s (disk) for a total CPU consumption of 135% and 185% respectively.

Finally for quorum size 3 and 5 ZKRL is able to maintain its aggregate throughput to a nearly steady level. CPU load on the leader increases across configurations but remains nearly constant within a given configuration as quorum size grows. This is because the incremental overhead at the leader for every additional follower is low, a key benefit of the chained replication pattern of ZKRL.

In the next experiments we deployed ZKRL in 10GBPS setup and we performed the ZKRL mem/disk experiment for quorum sizes 1, 2 and 3 respectively. The client creates and appends a log of 24GB in proposals of 64KB.

In Figure 5.5 we repeated the ZKRL mem/disk experiment for quorum sizes of 1 2 and 3 respectively. As we can see for quorum size 1 ZKRL mem saturates the network link reaching throughput 1102,4 MB/sec at 192% (out of 1600) aggregate CPU load at the leader. For ZKRL mem and quorum size 2 we reach a throughput of 827MB/sec with CPU load of 277%. The

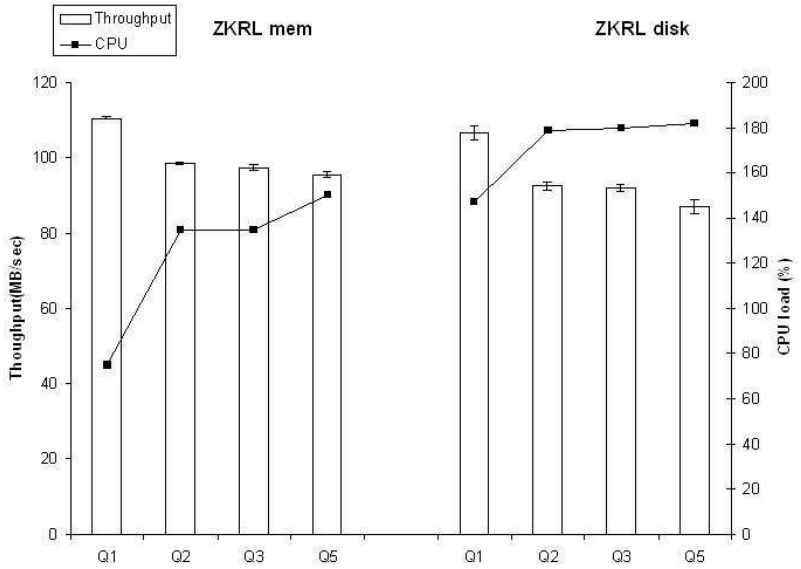

FIGURE 5.4: ZKRL throughput under different system configurations.

observation we make from this setup is that the FFQSender thread is saturated, because in the above setup it's core is at 95% (out of 100%) load. The parallelization of the FFQSender thread will benefit ZKRL scalability but must be designed in order not to violate the FIFO property which is essential for protocol correctness.

In ZKRL/mem quorum 3 setup we initially observed that the FFQRouter thread of the middle follower in the chain is saturated. The FFQRouter thread is responsible for receiving and sending messages through the chain. We observe one core at 100% CPU load and a performance of about 500 MB/sec. We parallelized the FFQRouter thread into FFQRouterReceiver and FFQRouterSender without violating the FIFO property. This gave as a performance of about 767 MB/sec with CPU load of 336%.

For the ZKRL disk setups we again issue a sync operation after 1500 proposals. As we can see in these setups the disk bounds the systems performance reaching a plateau of 125 MB/sec with CPU load around 145%.

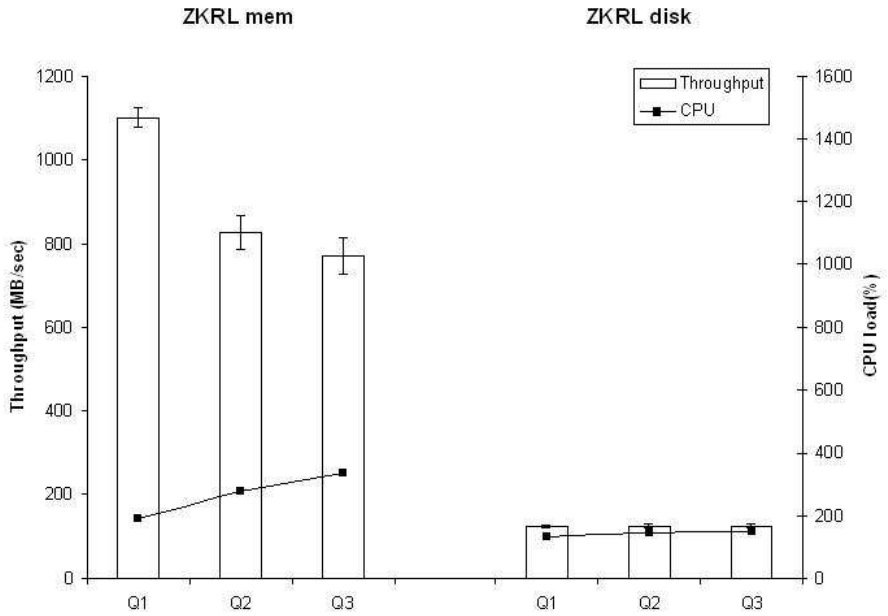

FIGURE 5.5: ZKRL mem/disk for quorum sizes of 1 2 and 3 respectively for 10Gbps network.

5.1.2 Read Performance

In this set of experiments we evaluated ZKRL read performance. Client creates a 8GB log and then reads it sequential. Client requests each 64KB znode forming its name by concatenating znode's filename and proposal number (derived from the integral part of the division of offset by proposal size). The server from which reads are done measured with dd for raw sequential read IO gives 166MB/sec whereas iozone benchmark for serial read gives a performance of 158,92 MB/sec. In 1GBPS setup, we saturate the network link as the read performance of ZKRL reaches 111 MB/sec with CPU load of 144% (out of 200) with one core at 95% load. In the 10GBPS setup we measured performance of 158,48 MB/sec with CPU load of 114% (out of 1600) with one core at 96% CPU load.

5.2 Recovery of ZKRL

The time of system unavailability during failure depends on the position of the failed node in the chain. If the failed follower is at the end of the chain recovery time is minimal because no message is lost and the system still passes new proposals (however the recovered node will have to receive all state that it missed in order to be reincorporated in the chain). If the failed node is the first node upstream in the chain it will block quorum progress (majority lost) and force the entire team to resynchronize with the leader, which is essentially equivalent to running leader election. If no proposals were lost, there will be minimal synchronization time between nodes. In this set of experiments we used a single NIC for sending and receiving data (at a peak throughput of about 80MB/sec measured via ttcp).

We examine two corner case scenarios during which a client writes at full speed. In the first scenario we have a quorum organized in the chain L - F1 - F2 - F3 - F4. The crash of follower F1 causes the loss of messages that F1 did not yet propagate further down the chain, mainly PROPOSAL and COMMIT messages. The leader's FFQSender will detect F1's failure (through TCP) and will bypass it forwarding messages to F2. When the leader realizes F1's failure (through heartbeats) it removes F1 from the routing table so future FFQ messages will not include it in their address vector. Follower F2 will receive messages out of order and thus (the FFQConsumer thread) will detect that the system has entered in an inconsistent state. The FFQRouter thread continues to propagate messages further down the chain. The same process takes place in F3 and F4 and they also restart themselves. When the leader detects that it has lost majority it also restarts itself, with the whole system entering leader election.

The current leader will be re-elected since it has been accepting requests and logging them to disk during the failure detection period, becoming the node that has seen the highest zxid. After re-election, it will start the synchronization phase sending to each follower the proposals it has missed. Our measurements show that F2, F3 and F4 detecting the out-of-order message to restarting and be detected by the leader as failed varies between 350 to

750 msec. When F2 detects that it has entered into an inconsistent state it updates a flag to indicate to all of its threads to shut down. So the above time gap includes the notification time and the time of shutting down the system. The leader's own restart plus the election round lasts about 1.7sec. During the synchronization phase each follower will have to sent about 200 proposals of 770KB each. The total state to be transferred through the chain is thus around 450 MB (each follower receives separately 200 proposals) lasting 6.5 sec for an aggregate throughput of about 72MB/sec with a peak network bandwidth of 80MB/sec since we use one NIC for sending and receiving data in this experiment. The total system downtime is 8.5sec. The current client implementation does not handle seamless failover to the new leader, so it simply exits. As we can see in this scenario a node failure that is near the leader causes a full system restart which is expensive in terms of downtime.

In the second scenario we have the same setup except that now we inject a crash of F3. In this case only F4 receives proposals out of order so there is a majority that continues to pass new proposals. The only penalty is the state transfer time to follower F4 during which period new proposals are stalled. The failure detection time varies from 147ms to 400ms and the synchronization lasts about 5.6 sec (35MB/sec). This time includes passing of lost proposals and the wait time in the FFQ for passing preceding proposals to F1 and F2 before the failure.

5.3 Performance of HDFS-ZKRL

5.3.1 HDFS Write Performance

In this set of experiments we evaluated vanilla HDFS (hdfs-crc), HDFS with CRCs disabled (hdfs-nocrc), and HDFS-ZKRL (hdfs-zkrl) in the first set of machines running DFSIO write benchmark. The set of experiments for hdfs-crc and hdfs-nocrc consists of two different setups (we use the notation (c, d) to refer to the number of clients (c) and datanodes (d): (1, 3) and (2, 6). All setups involve a single namenode. The client in these experiments is appending to a file with replication factor set to three. For hdfs-zkrl we used

the following setups (here we use the notation (c, lxn) to refer to the number of clients (c) and cells (l), each cell consisting of r nodes): (1, 1x3), (2, 2x3). The HDFS replication factor in the case of hdfs-zkrl is set to one (but each block is replicated three times within each cell).

Unmodified HDFS client middleware throttles by default when the sum of queued hdfs packets, that is packets that have been queued to sent, and packets that have been sent to the Datanode but not yet acked, reaches the value of 80. This value in the version 0.21.1 of hdfs is hardcoded and cannot be changed for networks with high throughput and high latency. For hdfs-crc, hdfs-nocrc and hdfs-zkrl we increased this value to 600. This values corresponds for hdfs-zkrl performance as depicted in Figure 5.2 to a leader window a little lower than 600. Finally in hdfs-zkrl we issue a sync operation after 300 proposals (each hdfs packet act as a proposal) except in the case where we receive the last packet of the block where a commit operation is forced. In all setups DFSIO creates a file and writes 8GB of data into it. The write packet size is always 64KB.

Figure 5.6 shows that unmodified HDFS reaches a throughput of about 75 MB/sec spending about half of the available CPU capacity (total of 200%). HDFS without CRCs shows slightly improved throughput and CPU usage. Hdfs-zkrl reaches a throughput of about 78,4MB/s at CPU usage of 170%. An explanation for the increased CPU in the case of HDFS-ZKRL lies in its co-location of HDFS and ZKRL code, ZKRL's complex data path (with additional tasks such as AckRP), the extra messages in hdfs-zkrl's write path (COMMIT messages), tree processing, and its failure detection mechanisms.

As we observe, none of the three systems reach saturation of any hardware resource of CPU, disk or network bandwidth. Client stops writing every 64MB and contacts the Namenode in order to allocate a new block creating thus small gaps in the data path. For this reason we decided to pull the Namenode out of the datapath so we increased block size to 8GB which turns maps a 8GB file to a single block allocation, completely eliminating namenode accesses besides the initial look up.

As we can see from figure 5.7 hdfs-zkrl reaches around 85MB/sec with CPU load of 176% (out of 200%), deducing in this case that we are CPU

bound. On the other hand, hdfs-crc reaches 79 MB/sec whereas hdfs-nocrc reaches 86 MB/sec. From these we can see that the write of the CRC on disk slightly affects performance. However hdfs nor hdfs-nocrc reaches its hardware limitations. From further experiments we observed that hdfs-nocrc with replication 1 saturates network link. Continuing, we removed disk from hdfs-nocrc data path, that means hdfs packets are only mirrored to the next node, nothing is written to disk. In this case hdfs-nocrc with replication 3 reaches network speed, 104 MB/sec.

Hdfs uses in each datanode one thread (BlockReceiver) for sending data to the next node and then writing them to disk without issuing any sync to disk operation. We observed that the block receiver thread periodically waits for a write to disk operation, when the buffer cache is full, in different random times in each datanode causing small bubbles in the pipeline degrading thus overall performance.

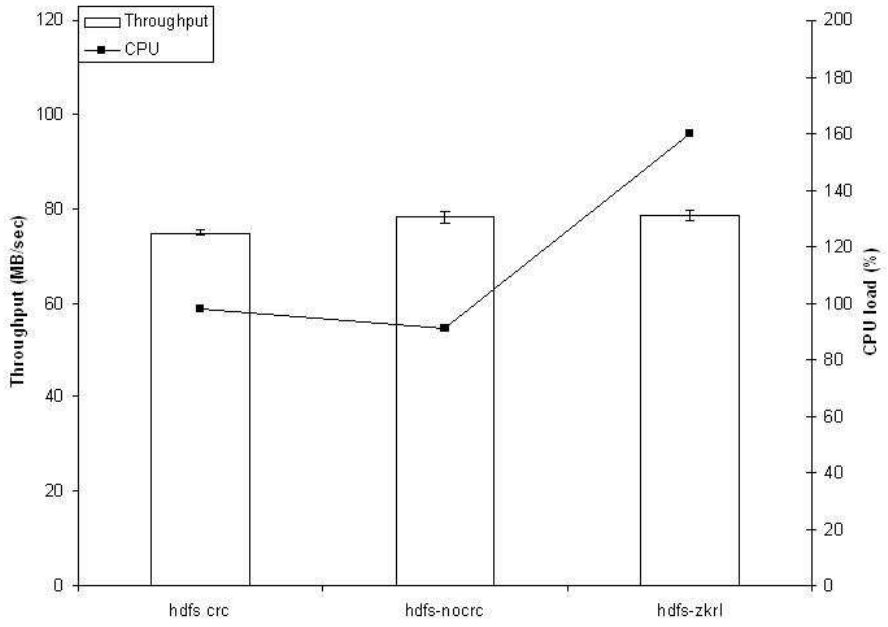

FIGURE 5.6: Write throughput and CPU for (1, 3) and (1, 1x3) and block size of 64MB

When going to two clients in the case of the (2, 6) and (2, 2x3) setups

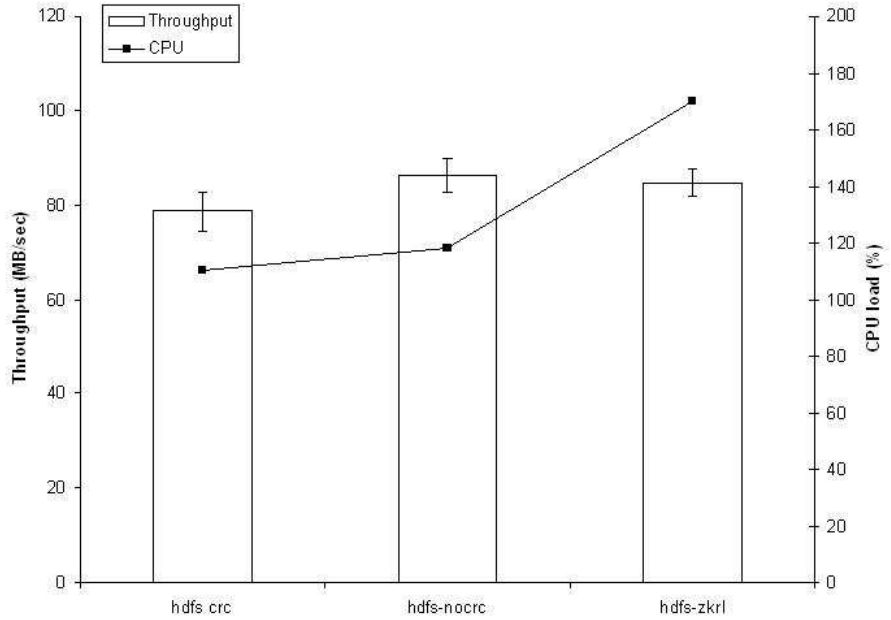

FIGURE 5.7: Write throughput and CPU for (1, 3) and (1, 1x3) and block size of 8192MB.

(Figure 5.8), we see that the aggregate throughput of the system for two clients is about 30% higher compared to the throughput with one client. With 6 datanodes one could hypothesize that two independent streams would be paralleled by the namenode and thus observe an increase of the aggregate throughput in comparison to the first setup by up to an order of two. However this does not happen. Our explanation for this (which we validated by analyzing traces of namenode actions over the experiments) when the namenode configures replica groups for consecutive-block allocations, it seems to be oblivious of the fact that the blocks may be concurrently be written and thus often allocates them on overlapping replica groups (sharing one, two, or sometimes three nodes).

More specifically, our experiment of the (2, 6) hdfs-crc/nocrcs setup showed that 23% of the blocks had no collisions, whereas 35%, 36%, and 6% of concurrent writes where to replica groups sharing one, two, and three datanodes respectively. This explains why the aggregate throughput is not

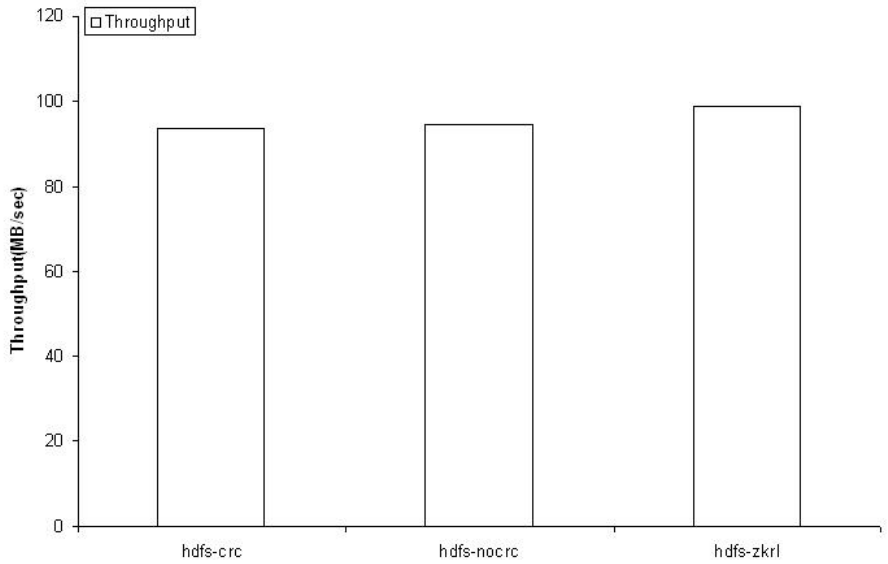

FIGURE 5.8: Write throughput/CPU for (2, 6), (2, 2x3) setups.

doubled in this case. In 59% of all hdfs-zkrl concurrent writes, blocks went to independent replica groups whereas 41% went to the same ZKRL cell.

In the following set of experiments we run again DFSIO benchmark for a 24GB file in the second set of machines. DFSIO and the Namenode run on the same machine.

Again we can see the same behavior for hdfs-crc and hdfs-nocrc. Block-Receiver thread periodically freezes stopping thus the entire pipeline and dropping performance at around 100MB/sec with CPU load of 53% (out of 1600) for hdfs-crc and 49% (out of 1600) for hdfs-nocrc. Hdfs-zkrl reaches a throughput of 110 MB/sec with a CPU load of 160% (out of 1600). Again hdfs-zkrl syncs every 300 proposals.

5.3.2 HDFS Read Performance

In this experiments we evaluated hdfs crc, hdfs nocrc and hdfs zkrl read performance. In 1GBPS setup we run DFSIO read benchmark. An 8GB file is read sequentially and every request is served by the same datanode. As

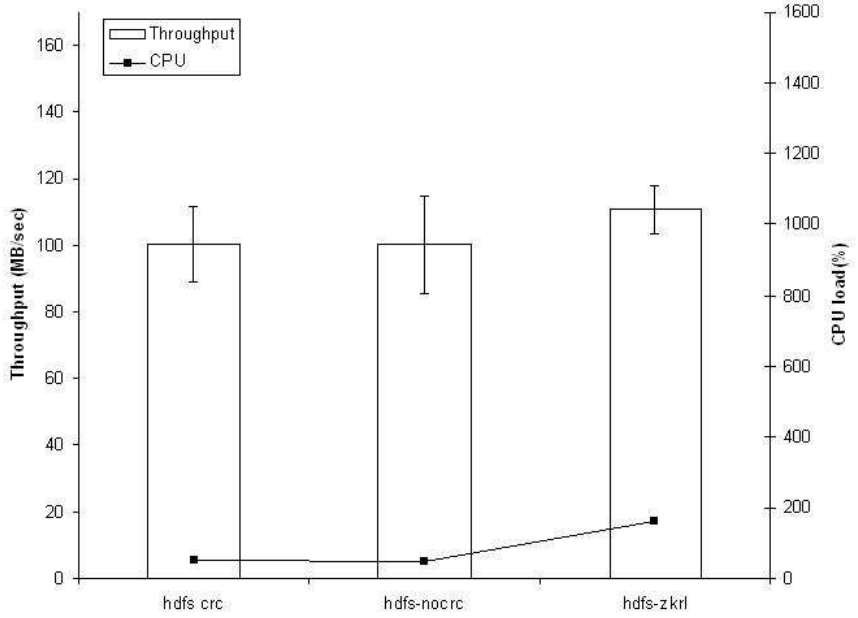

FIGURE 5.9: Hdfs-crc, hdfs-nocrcs and hdfs-zkrl write performance for 64MB block size in 10Gbps network

shown in Figure 5.10 hdfs read path saturates 1 Gbps link (102 MB/sec). As we can see the extra stage in hdfs-zkrl with the additional look up in the zk tree does not affect performance. For the three setups we observe one core at 95% load.

We repeated the same experiment for the 10GBPS setup, as shown in Figure 5.11, running again DFSIO read benchmark. Hdfs-crc gives a performance of about 141 MB/sec whereas hdfs-nocrc and hdfs-zkrl reach 157 MB/sec again with one core in the three setups at 95% load. The slight drop of performance in hdfs-crc is due to reading except from the block file the crc file. This difference in performance arise as we reach disk's speed in the 10GBPS setup.

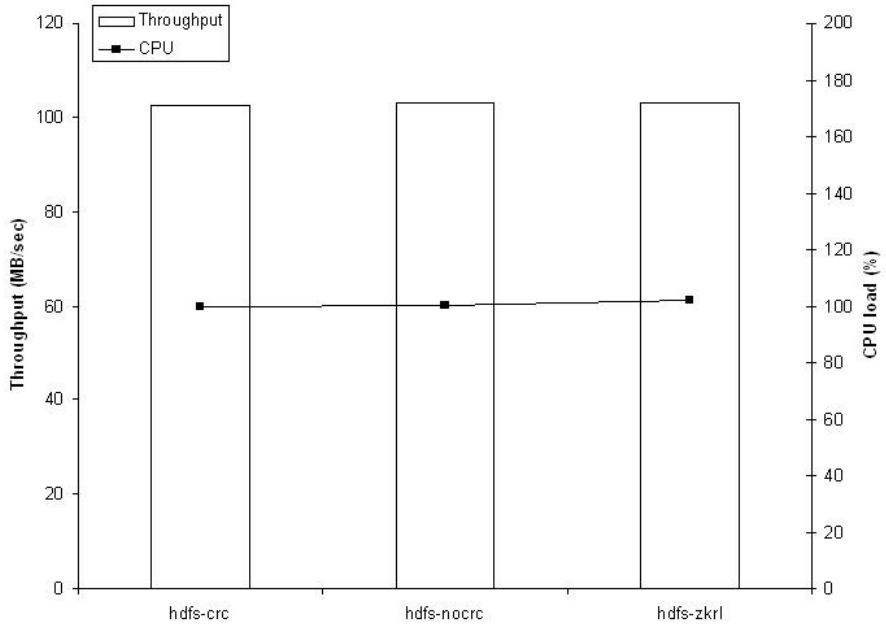

FIGURE 5.10: Read performance of hdfs-crc, hdfs-nocrc and hdfs-zkrl for 1Gbps network.

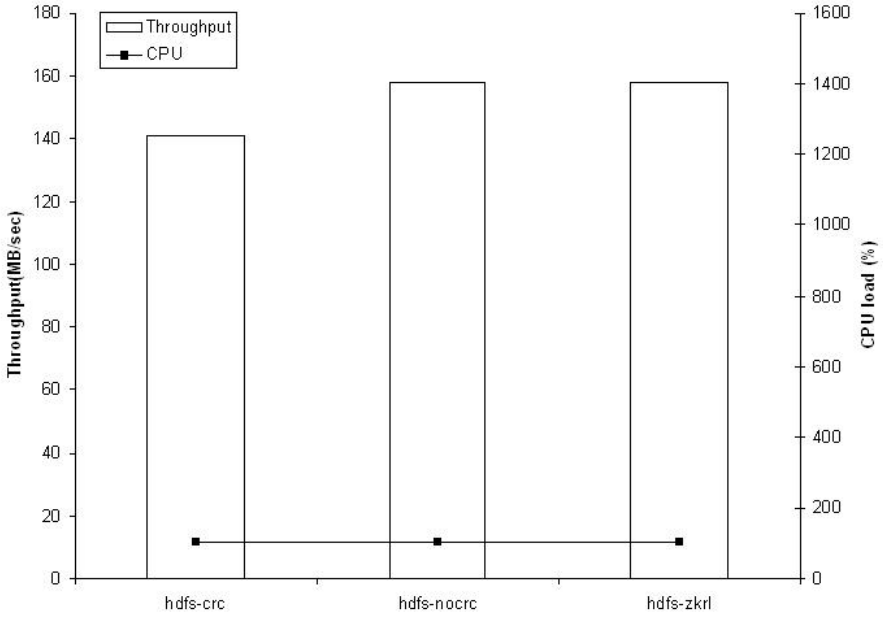

FIGURE 5.11: Read performance of hdfs-crc, hdfs-nocrc and hdfs-zkrl for 10Gbps network.

Chapter 6

Conclusions and Future Work

In this study we presented ZKRL, a scalable storage service for keeping replicated logs consistent among a set of servers. ZKRL can be used either standalone as the storage layer of an application or as a software core for the rapid design of more complex storage infrastructures. Our evaluation has shown that ZKRL can achieve performance close to peak I/O (network or disk) capabilities when appending to replicated logs over 1-Gbps and 10-Gbps networks. The use of ZKRL as a component integrated into HDFS strengthens HDFS's semantics without degrading its performance. The trade-off for achieving this is the increased CPU load compared to unmodified HDFS in 1Gbps (35%) and in 10 Gbps (6%) networks. This transformation makes HDFS applicable for a wider range of applications.

There are a number of interesting issues that are subject to further research. Regarding ZKRL, follower failures inside the chain may result in large system downtime whereas in the tree topology the system remains unaffected from follower failures as long as a majority of followers are alive. In this direction a mechanism could be introduced for retransmission of recent lost proposals so the need for system restarts is diminished. Also as we have shown in ZKRL evaluation over 10Gbps networks the FIFO property (which is necessary for protocol correctness) limits parallelism. Exploiting ways to increase parallelism in ZAB protocol is one way to improve performance.

HDFS forbids concurrent access to the same file for write operations. We have altered HDFS for this purpose and performed preliminary evaluations.

However more work remains to be done. Finally in HDFS the client can explicitly decide about the replication factor of a file. The namenode will choose a replica group of the appropriate number of datanode servers, which it will report to the client. In hdfs-zkrl, ZKRL cells are preconfigured, so the replication factor of a file cannot be decided dynamically. An alternative architecture allowing for ZKRL cells with different number of quorum sizes can be exploited to provide the same flexibility to applications.

Bibliography

[1] Oracle Java New I/O API and Direct Buffers. http://download.oracle.com/javase/1.4.2/docs/guide/jni/jni-14.html.

[2] Y. Kwon M. Balazinska and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated filesystem. In *Volume 1, pp. 574-585. VLDB Endowment*, 2008.

[3] DFSIO benchmark. http://answers.oreilly.com/topic/460-how-to-benchmark-a-hadoop-cluster/.

[4] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating System Design and Implementation (OSDI'06)*, November 2006.

[5] Cloudera. http://www.cloudera.com/ and http://www.dbms2.com/2009/10/10/enterprises-using-hadoo/.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, December 2004.

[7] K. Magoutis et. Making the most out of direct access network-attached storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'03)*, April 2003.

[8] B. Liskov et al. Replication in the harp file system. In *Proc. of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[9] F. Chang et al. Bigtable: A distributed storage system for structured data. In *Proceedings of Seventh Symposium on Operating System Design and Implementation (OSDI'06)*, November 2006.

[10] J. Bent et al. Enabling scientific application i/o on cloud file systems. In *Poster in 2010 USENIX Conference on File and Storage Technologies (FAST'2010)*, February 2010.

[11] J. McCormick et al. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, December 2004.

[12] J. McCormick et al. Niobe: A practical replication protocol. In *ACM Transactions on Storage, v.3 no. 4, article 14*, February 2008.

[13] K. Magoutis et al. Structure and performance of the direct access file system (dafs). In *Proceedings of USENIX Annual Technical Conference, Monterey*, June 2002.

[14] K. Shvachko et al. The hadoop distributed file system. In *Proceedings of IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'2010)*, 2010.

[15] P. Hunt et al. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[16] R. Ananthanarayanan et al. Cloud analytics: Do we really need to reinvent the storage stack? In *HotCloud '09: Proceedings of USENIX Workshop on Hot Topics in Cloud Computing*, June 2009.

[17] S. Gribble et al. Scalable, distributed data structures for internet service construction. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI'2000), San Diego, CA, October 2000.*, October 2000.

[18] Y. Mao et al. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, December 2008.

[19] Apache Software Foundation. http://hbase.apache.com.

[20] Apache Software Foundation. http://wiki.apache.org/hadoop/bookkeeper.

[21] Apache Software Foundation. Mahout http://mahout.apache.org.

[22] Apache Software Foundations. https://issues.apache.org/jira/browse/hadoop-6330.

[23] Jim Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course, volume 60 of Lecture Notes in Computer Science*, pages 393–481, 1978.

[24] A. Cox J. Shaffer, S. Rixner. The hadoop distributed file system: Balancing portability and performance. In *2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'10)*, March 2010.

[25] S. Quinlan K. McKusick. Case study gfs: Evolution on fast forward. In *ACM Queue Magazine*, 2009.

[26] L. Lamport. The part-time parliament. In *ACM Trans. Comput. Syst.*, pages 16(2):133–169, 1998.

[27] C. H. Papadimitriou. The serializability of concurrent database updates. In *Journal of the Association for Computing Machinery, Vol. 26, No 4, pp 631-653*, October 1979.

[28] F. Schneider R. van Renesse. Chain replication for supporting high throughput and availability. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, San Fransisco, CA.

[29] S.-T. Leung S. Ghemawat., H. Gobioff. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, December 2003.

[30] F .B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. In *ACM Comput. Surv.*, pages 22(4):299–319, 1990.

[31] J. Redstone T. Chandra, R. Griesemer. Paxos made live - an engineering perspective (2006 invited talk). In *in Proceedings of the 26th Annual ACM Symposium*, 2007.

[32] Underneath the Covers at Google: Current Systems and Future Directions. http://sites.google.com/site/io/underneath-the-covers-at-google-current-systems-and-future-directions.

[33] ttcp Benchmarking Tool. http://www.pcausa.com/utilities/pcattcp.htm.

[34] Apache Zookeeper. http://zookeeper.apache.org/.