Persistence and Node Failure Recovery in Strongly Consistent Key-Value Datastore

MUHAMMAD EHSAN UL HAQUE

Master of Science Thesis
Stockholm, Sweden 2012

TRITA-ICT-EX-2012:175
Persistence and Node Failure Recovery in Strongly Consistent Key-Value Datastore

By
Muhammad Ehsan ul Haque

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering of Distributed Systems.

Supervisor & Examiner: Dr. Jim Dowling.

TRITA-ICT-EX-2012:175

Unit of Software and Computer Systems
School of Information and Communication Technology
Royal Institute of Technology (KTH)
Stockholm, Sweden.
December 2012
Abstract

Consistency preservation of replicated data is a critical aspect for distributed databases which are strongly consistent. Further, in fail-recovery model each process also needs to deal with the management of stable storage and amnesia [1]. CATS is a key/value data store which combines the Distributed Hash Table (DHT) like scalability and self organization and also provides atomic consistency of the replicated items. However being an in memory data store with consistency and partition tolerance (CP), it suffers from permanent unavailability in the event of majority failure.

The goals of this thesis were twofold (i) to implement disk persistent storage in CATS, which would allow the records and state of the nodes to be persisted on disk and (ii) to design nodes failure recovery-algorithm for CATS which enable the system to run with the assumption of a Fail Recovery model without violating consistency.

For disk persistent storage two existing key/value databases LevelDB [2] and BerkleyDB [3] are used. LevelDB is an implementation of log structured merged trees [4] where as BerkleyDB is an implementation of log structured B+ trees [5]. Both have been used as an underlying local storage for nodes and throughput and latency of the system with each is discussed. A technique to improve the performance by allowing concurrent operations on the nodes is also discussed. The nodes failure-recovery algorithm is designed with a goal to allow the nodes to crash and then recover without violating consistency and also to reinstate availability once the majority of nodes recover. The recovery algorithm is based on persisting the state variables of Paxos [6] acceptor and proposer and consistent group memberships.

For fault-tolerance and recovery, processes also need to copy records from the replication group. This becomes problematic when the number of records and the amount of data is huge. For this problem a technique for transferring key/value records in bulk is also described, and its effect on the latency and throughput of the system is discussed.

Key Words:

Datastore, Distributed Hash Table (DHT), Atomic Shared Registers, Disk Storage, Consistency, Failure Recovery, Paxos, Group Membership.
Acknowledgements

I would like to express my gratitude towards my supervisor and examiner Dr. Jim Dowling for giving me the opportunity to work on the project. I would also like to thank him for his continuous support and technical guidance throughout the project.

I would like to say a very special thanks to Cosmin Arad and Tallat Mahmood Shafaat for their valuable technical support and guidance throughout the project, without which the project was not possible.

Finally, I would like to thank my family for their love, care, devotion, encouragement and support for me.
# Table of Contents

Chapter 1: Introduction

1.1 CAP Theorem

1.2 Strong vs Eventual Consistency

1.3 Recovery in Replicated Datastore

1.4 Thesis Contribution

1.5 Related Work

1.5.1 PNUTS

1.5.2 Cassandra

1.5.3 Dynamo

1.5.4 Spinnaker

1.6 Roadmap of the Document

Chapter 2: CATS

2.1 CATS API

2.1.1 get(Token key)

2.1.2 put(Token key, String v)

2.2 CATS Architecture

2.2.1 Chord Overlay

2.2.2 Storage Model

2.2.3 Paxos for Consistent Quorums

2.2.4 Failure Detector and Failure Handling

2.2.5 CATS Data Structures

2.2.5.1 Local Range Store

2.2.5.2 Item Store

2.2.5.3 Paxos State

2.2.6 Bootstrap Server

2.2.7 Reads and Writes

2.2.7.1 Read

2.2.7.2 Write

2.2.8 Failure and Join

2.2.8.1 Joining the Overlay Network

2.2.8.2 Joining Groups

2.2.8.3 Node Failure

Chapter 3: Disk Persistence

3.1 Motivation for Disk Persistence
6.1 Experiments with Recovery ................................................................. 36
6.2 Performance Experiments ................................................................. 36
  6.2.1 Load/Insert Performance ............................................................... 37
  6.2.2 Effect of Record Size on Disk Size ............................................... 37
  6.2.3 Effect of Record Size on Latency .................................................. 38
  6.2.4 Latency Variance with Time ......................................................... 38
  6.2.5 Effect of Server Threads on Latency .............................................. 39
  6.2.6 Effect of Varying Load (Client Threads) on Latency ...................... 40
  6.2.7 Scalability with Write Intensive and Read Intensive Load ............... 40
  6.2.8 Bulk Transfer ............................................................................... 41
    6.2.8.1 Effect of Chunk Size on Download Time ................................. 41
    6.2.8.2 Download Time with Data Size ............................................... 41
    6.2.8.3 Effect of Chunk Size on Latency and Throughput ..................... 42
  6.2.9 Elasticity ...................................................................................... 43

Chapter 7: Future Work and Conclusion ................................................. 44

  7.1 Future Work .................................................................................... 44
    7.1.1 Evaluation of bLSM with CATS .................................................. 44
    7.1.2 System Wide Recovery ............................................................... 44
    7.1.3 Merkle Trees ............................................................................. 44
    7.1.4 Thread Pool in Bulk Transfer Component for Sources ................. 44
    7.1.5 Load Balancing ......................................................................... 44
    7.1.6 Distributed Range Queries ......................................................... 45
    7.1.7 Object Storage Model ................................................................. 45
    7.1.8 Distributed Transactions in CATS .............................................. 45
    7.1.9 Using CATS for HDFS Name node ............................................ 45

  7.2 Conclusion ....................................................................................... 45
List of Figures & Graphs

Figure 1.1: CAP Theorem........................................................................................................1
Figure 2.1: CATS API...........................................................................................................6
Figure 2.2: CATS Overlay Architecture...............................................................................7
Figure 2.3: CATS storage model..........................................................................................7
Figure 2.4: Message flows in Get and Put requests.............................................................11
Figure 2.5: Steps involved in node join..............................................................................12
Figure 2.6: Steps involved after detecting node failure.......................................................13
Figure 3.1: CATS Persistence...............................................................................................16
Figure 3.2: BerkleyDB database structure..........................................................................17
Figure 3.3: LevelDB LSM Trees and Compactions.............................................................19
Figure 3.4: Linearizability violation with multiple threads................................................22
Figure 4.1: CAS Failure scenario.........................................................................................25
Figure 5.1: Range download strategy in bulk transfer.........................................................33
Figure 6.1: Insert performance of CATS with DB engines.................................................39
Figure 6.2: Effect of record size on disk size on different DB engines...............................40
Figure 6.3: Effect of record size on latency with different DB engines...............................41
Figure 6.4: Effect of disk cache..........................................................................................41
Figure 6.5: Effect of server threads on latency with different DB engines..........................42
Figure 6.6: Effect of varying load on latency with different DB engines.............................43
Figure 6.7: Scalability of write intensive load with different DB engines............................43
Figure 6.8: Scalability of read intensive load with different DB engines.............................44
Figure 6.9: Effect of chunk size on download time with different DB engines....................44
Figure 6.10: Effect of download size on download time with different DB engines.............45
Figure 6.11: Effect of chunk size on latency with different DB engines.............................46
Figure 6.12: Elasticity of CATS with different DB engines................................................46
The advent of web 2.0 has entirely changed the way internet was being used. It is no more a way of transmitting information from web sites to user but also allows users to interact and collaborate with each other and thus allowing them to add value to the web [7]. Today’s most popular internet applications, which are now considered as services, are Social media networks (Facebook, Twitter, Google+ etc), wikis, blogs, YouTube, flicker, e-Bay, amazon etc. The most common feature, among them, is that each one of them is dealing with a very huge volume of data; therefore the importance of data storage and management is the most crucial aspect for all of them.

The massive amount of data that these applications need to access and the the high rate with which they produce data requires the underlying data storage to be massively scalable. The scaling requirements for such application is beyond the capabilities of enterprise data base systems [8]–[10]. Sharding is considered a common cost effective solution, where each node of the cluster is responsible for part of data and runs a separate instance of the database software [11]. However there have been reported problems with sharding [12].

Apart from scalability modern web-applications need high availability. This is a fact that these applications run in a cluster of several hundred or thousands of nodes with failures happening all the time. To achieve high availability, where failures are norm, some replication strategy is needed to replicate the data across multiple nodes. One common replication technique is synchronous master-slave replication as used by many sharded databases. However master slave replication is not an ideal solution for an internet and cloud scale application [11].

1.1 CAP Theorem

In 2000 Eric A Brewer, in his talk presented the CAP conjecture [1], which was later proved by Seth and Nancy in 2002 [13].

According to CAP theorem, out of the three desired properties: consistency, availability, and partition tolerance only two can be provided at the same time by any replicated data store, as shown in Figure 1.1. At internet and cloud scale network failures have statistically high
probability, and therefore applications at such scale are left to choose between consistency and availability AP and CP.

1.2 Strong vs Eventual Consistency

Strong consistency gives a single server illusion to the clients and guarantees that all replicas appear identical to the clients. CAP theorem presents a real limitation on the large distributed systems and has led to the popularity of eventual consistency [14]. Eventual consistency is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. In eventual consistency the data items may diverge in different replicas due to accessing less than the replica or network partition [1], [13].

Systems like [15]–[17] are eventual consistent and are aimed at extremely high availability. However still there are many other applications like financial applications, electronic health record applications and many others which desire strong consistency and some form of transaction support. Also for applications that run in a LAN environment, opting for strong consistency is more appropriate [18].

1.3 Recovery in Replicated Datastore

In a distributed data store, consistency of the system after recovery of failed nodes is also needed to be considered. For an eventual consistency model the recovered node can be integrated with the system immediately, the requirement is only that the recovered node will become consistent in a bounded time. For a strongly consistent model the recovery algorithm needs to ensure the consistency of the data with other nodes before integrating with the system. Special care is needed as the state of the sources are also changing during the recovery.

Node recovery for a replicated data store with strong consistency have been discussed in the literature [19], [20].

Several methods have been proposed, including disk based and disk less recovery [21]. In disk-less recovery, the node recover with complete amnesia, with no memory of its state or data items and is integrated in the system as a fresh node. The inherent redundancy of the replicated data is used to recreate the state of the recovering node from a clean slate [22]. This may require a lot of network traffic, depending on the amount of data that it needs to take from other nodes. In disk based recovery the conventional mechanism is to maintain a log of operations on a local persistent storage and recreating the state by replaying the logs and only getting the updates from the neighboring nodes, reducing the network traffic to minimum [11].

1.4 Thesis Contribution

The work done in this thesis is an extension of the work done in a research project, called CATS, being conducted at SICS (Swedish Institute of Computer Science). CATS is a distributed key-value store which uses consistent quorums to guarantee linearizability and partition tolerance and is scalable, elastic, and self-organizing which are key properties for modern cloud storage middle-ware [22]. The contribution of the thesis are.
1. To implement a local disk storage for storing the data items on a persistent storage. With a disk based storage we expect the latency of the operations to increase as compared to a memory based storage. However, this will allow the amount of data that can be stored on each node to go beyond the available memory on the nodes and thus increasing the capacity of the whole system.

2. To implement node recovery algorithm, allowing the nodes to recover with an old state and catching up with the nodes with updated states to maintain consistency of the replicated data items.

A major limitation of the memory based model is the impossibility of the system recovery in the event of majority failures. The node recovery algorithm can handle recovery after majority failures without violating consistency. Another use case of the node recovery, not implemented in this thesis, is to allow system wide shutdown and recovery. This has been briefly discussed in the future work section.

3. Optimizations to increase the throughput of read write operations, by parallelizing non conflicting operations.

1.5 Related Work

1.5.1 PNUTS

PNUTS is a massively parallel and geographically distributed database system for Yahoo web application [21]. It is centrally managed and geographically hosted system, providing high scalability, high availability and fault tolerance. The storage is organized into hashed or ordered tables. PNUTS relies on YMB (Yahoo Message Broker) a centralized message broker system as a replacement for the redo logs and as a replication mechanism. PNUTS provides a per-record timeline consistency model, where all replicas of a record apply updates to a record in the same order. Timeline consistency is stronger than eventual consistency but weaker than linearized consistency model. The disk storage for PNUTS is a proprietary disk based hash table and MySQL innoDB for ordered tables. Node recovery is a three step process in PNUTS (i) initiating a copy from remote replica (source) (ii) check-pointing at the source to ensure in-flight operation are applied and (iii) Copying the tablet to destination. The recovery protocol requires consistent boundaries of the tablets across replicas. This is maintained by partitioning the tables, synchronously across the replicas, when needed.

1.5.2 Cassandra

Cassandra is a distributed system for managing very large amount of structured data. Cassandra is highly scalable and uses consistent hashing like chord [23], to dynamically partition the data over nodes in cluster. The data model consists of multidimensional maps indexed by keys and values consist of columns which are grouped together to form column families, like in BigTable [24]. High availability is achieved by various replication strategies; rack aware and data center aware strategies. Read/Write operations are quorum based with a tunable replication. Records are persisted on local file system, a sequential commit log is used for durability and recovery, and an in memory data structure is used for efficient reads. Merges are performed in background like compaction in BigTable [24]. Recovery
takes place by identifying the records and then reading and writing the latest version of the
data. Since the recovering node also responds to the service requests therefore an old
value can be read by the client if read with a lower consistency level. In this respect
Cassandra has an eventual consistency model as the recovering node eventually gets the
latest version of the data.

1.5.3 Dynamo

Dynamo is a highly (always) available key-value data store. In CAP terminology dynamo is
partition tolerant and available and sacrifices consistency under certain failing scenarios
[15] and thus provides eventual consistency. Data partitioning and replication is provided
through consistent hashing [25], and consistency is provided by object versioning [26] and
conflict resolution is performed through vector clocks. Dynamo uses a pluggable storage
engine and has been used with Berkley DB [3] (transactional and Java edition), MySQL
and in memory buffer with a persistent backing storage. Replica synchronization which is
essential for node recovery is handled by Merkle trees [27], which is an anti-entropy
protocol for quickly identifying the deviations and reducing the amount of data transfer
during replica synchronization.

1.5.4 Spinnaker

Spinnaker is an experimental datastore designed to run on a large cluster in a data center
[11]. It provides key-based partitioning and N-way replication and provides transactional
and timeline put/get operations. Spinnaker uses Paxos [6] for consistent replication and
provides availability as long as majority of the replicas are alive. It relies on Zookeper [28]
for storing meta data and failure detection. Spinnaker uses write ahead log which uses
uniquely identified log sequence number LSN for each replication group (called cohort) for
recovery. Committed writes are stored in a memtable which are periodically flushed to an
SSTable and stored on an immutable storage. Each cohort consists of a leader and two or
more followers; the failure of a leader makes the cohort unavailable and new leader is
chosen using leader election. The new leader is chosen such that its log contains all the
writes committed by the failed leader. A followers recovery involves applying local log
records and then catching up with the cohorts leader. Even though Spinnaker claims to be
consistent, it may lose consistency in a particular failing scenario where a leader and a
follower fails permanently in rapid succession.

1.6 Roadmap of the Document

The thesis document is structured as follows. Chapter 2 gives an overview of CATS. It
describes the CATS API and architecture and explains the various components and
building blocks for CATS. It also explains how read and write operations are performed and
how dynamic reconfiguration during node failures and joining of new nodes takes place.
Chapter 3 describes the disk based persistence model implemented in CATS. It covers the
architecture of the two existing key-value datastore, Berkley DB Java Edition and LevelDB,
explaining the implication and effect of the underlying architecture of these DB engines on
CATS read/write operations. The chapter also explains an optimization by introducing
parallel operation to improve throughput. Chapter 4 discusses the failure recovery of the
CATS nodes that relies on the disk persistence layer implementation and explains the
recovery algorithm with a formal proof. Chapter 5 explains the bulk transfer component that is responsible for replicating data across the replication groups and plays an essential role in dynamic reconfiguration of the system. Chapter 6 presents and compares the experimental results obtained by using BerkleyDB and LevelDB as storage engines. Chapter 7 discusses future work and conclusion.
Chapter 2 : CATS

As previously mentioned CATS is a key value distributed data store that provides atomic reads and writes and thus provides an atomic consistent storage. In this section we will discuss the interface provided by CATS for reading and writing key/values, next we will describe the architecture of CATS and in the end we will describe, how read and writes are performed and how nodes join and leave the system.

2.1 CATS API

CATS provides a simple API to store and retrieve string values; for storing complex objects clients need to serialize the objects into string before storing them. The read and write operations are atomic/linearizable. Each value is associated with a key; the key is of type Token and the tokens can be of type string, long and byte arrays. The token type is a system wide property of CATS. Figure 2.1 shows the operations that can be performed using CATS API. The API provides following two functions.

2.1.1 get(Token key)

Returns the last value written with the key “key”. In case if the operation is concurrent, with a write or failed write with the same key “key”, the function may return the last value written or the concurrent value that is being written. In the later case all subsequent calls to the get function for the same key will never return an older value.

2.1.2 put(Token key, String v)

Inserts or replace a value “v” associated with the key “key”. A successful return guarantees that the value has been inserted or updated in the majority of replicas and a subsequent read operation for the same key “key” will never return a value older than the value “v”. A failure return does not guarantee anything i.e. the value may or may not be inserted or updated in the majority of replica.

2.2 CATS Architecture

This section will provide an overview of the architecture of CATS. The goal is to provide enough background to understand the later chapters which are the main contribution of the thesis.

2.2.1 Chord Overlay

The nodes in CATS are connected with each other to form a chord overlay network topology [23]. The chord architecture provides scalability, load balancing, decentralization and availability. The consistent hash service provided by chord is used to lookup nodes.
The actual implementation of chord uses finger table and stores information of $O(\log N)$ nodes and requires on average $O(\log N)$ hops to resolve lookup requests. However for the purpose of CATS, where there are few nodes and all of them residing in a cluster the finger table strategy might increase the latency of lookup request; therefore instead of maintaining a finger table, CATS uses a successor list. The length of the successor list is a system wide property. A successor list of length greater than or equal to the number of nodes in the system will resolve all lookup requests in one hop. The nodes in the successor list are updated during the stabilization and in addition to that some nodes can be evicted from the successor list when the nodes in the successor list are suspected to be failed by the failure detector 2.2.4. The remaining algorithm for stabilization and node joining and leaving from the chord overlay ring is same as described in [23]. Figure 2.2 shows the chord overlay of 8 nodes with a successor list of length 4.

### 2.2.2 Storage Model

In CATS the records are partitioned into disjoint connected ranges. Each node is responsible for storing a set of ranges; each range is further replicated into different nodes. The number of nodes where ranges are replicated is defined by a system wide configuration parameter called “replication degree”. A replication degree of 3 means that each range and all the records falling within the range is replicated in 3 different nodes; but due to the majority quorum approach, used in CATS, the records are only required to be written in the majority of the replica, which in this case will be 2 nodes. Also the replication degree must be less than or equal to the length of successor list (described earlier). The replication technique is similar to the “chained declustering” [29], where each range is assumed to be a primary range in one node and secondary range in the subsequent, next replication degree - 1 nodes, on the chord overlay ring. In CATS a node is considered to be a primary replica for the subset of all the ranges from the predecessor node id (exclusive) to the node’s id (inclusive). However the labeling of primary and secondary replica is only superficial; the nodes replicating the ranges are not aware of...
being primary and secondary replica. The term is only used for explanatory purpose. Figure 2.3 shows the storage model of CATS for a replication degree of 3.

### 2.2.3 Paxos for Consistent Quorums

For consistent storage of replicated records, CATS uses the abstraction of atomic shared register [44]. The distributed atomic shared register are implemented using a quorum based algorithm. The quorum based approach ensures atomic consistency/linearizability for the read and write operation. Due to the asynchronous message passing and eventually strong failure detector [30], nodes can be detected inaccurately, therefore the group membership information for the replicated items can get inconsistent which can lead to inconsistency in read/write operations. Maintaining a consistent group membership is a well known consensus problem [31]. A well known solution for consensus is Paxos [6]. In the simplest form Paxos allows different processes to propose values and ensures that:

- Only a value that has been proposed is chosen.
- Only a single value is chosen.
- A process never learns a value that it has been chosen unless it is chosen.

In CATS we use Paxos to achieve consistent group memberships (consistent quorums), which along with the majority based read imposed writes ensures consistency of replicated records.

### 2.2.4 Failure Detector and Failure Handling

Failure detector, monitor system components in order to detect failures and notify registered components when a failure of one or more monitored component occurs [32]. Based on the completeness and accuracy properties failure detectors are classified into various classes [30]. Completeness property ensure that every correct process eventually detects the failure of a crashed process and accuracy property restricts that the decision made by the failure detector can not be wrong. It is also a well known fact that strong accuracy can not be achieved in an asynchronous network; therefore CATS uses an eventual strong failure detector (◇S), which has eventual weak accuracy and strong completeness property. The Failure detector is implemented in the Kompics framework [33] and uses a heartbeat mechanism to monitor other processes. In CATS each node has its own local failure detector and it uses this failure detector to monitor the neighboring nodes which includes, predecessor node, successor nodes in the successor list and the group member nodes of the groups in which the node is participating.

When a node is suspected by the failure detector an event of node suspicion is triggered and the node is removed from the monitoring list. The handling of suspicion event requires different actions to be performed for different nodes.

### 2.2.5 CATS Data Structures

In this section we will describe some of the important data structures/state variables.

#### 2.2.5.1 Local Range Store

As mentioned earlier each running node keeps the group membership information. This membership information is kept in the Local Range Store as a set of Local Range. The
Local range consists of the range, group (which is an array of node references) and the version number. Once a group is modified the version number of the new Local range is increased by one.

2.2.5.2 Item Store

Item Store maintains the records stored in the node as well as it maintains the set of ready and busy ranges. Ready ranges are those ranges for which the node can do read and write operations, while the busy ranges are those ranges for which the node is responsible but is waiting to get the records from the other group member. This usually happens after a view change, when the new node has received a view installation message but the data for that range has not yet been downloaded from other group members.

2.2.5.3 Paxos State

Each node can act as a proposer and acceptor in Paxos rounds [6]. The data for each Paxos role, acceptor and proposer, is maintained. For proposer state, we need to keep all the proposals that the node has proposed in the ongoing rounds. Once a round is completed the proposal is discarded for garbage collection. For acceptor we store all history of accepted proposals that the node has accepted in any round. The storage of these proposal and acceptor states are mandatory for Paxos implementation.

2.2.6 Bootstrap Server

In a dynamic system where nodes join and leave at any time, establishing a precise population size is problematic [34]. Also policy enforcement, whether dynamic or static, in a completely distributed, asynchronous and decentralized system is a challenging problem. Another related problem is membership awareness which require synchronous communication. In order to solve these problems CATS uses bootstrapping. A bootstrap server is a different process (other than the CATS node processes), whose responsibility is to maintain a global view of all the nodes in the system and assist new nodes in entering the system. In CATS there is a fixed threshold policy, in which when the number of nodes in the system is less than a threshold size i.e. \( n < t \), a “below threshold” policy is applied. In this “below threshold” policy the new node entering in the system is only stored and no further action is taken. When the number of nodes in the system becomes equal to the threshold size i.e. \( n = t \) an “equal threshold” policy is applied in which the bootstrap server orders the nodes (based on the node identifiers) and form an overlay chain and inform all the nodes about their predecessor and successors. This special case usually happens when the whole system is started. In a long running system such an event will be very rare. For the “above threshold” policy i.e. \( n > t \) the bootstrap server immediately responds to the newly entering node with a random node in the system. The new node then sends a lookup request for its own identifier with the node provided by the bootstrap. The lookup response returns the node with which the new node should join the system. The joining node then follows the protocol mentioned in Chord [23] to join the system.

The bootstrap server also sends heartbeats to all the current nodes who are in the system to maintain a global view of the system. The view might not be correct at all the time but an always correct view is not required. If the nodes joining the system can re-try after some time, then this will allow the bootstrap server to detect the failure and then redirecting the joining node to a live node. There is also an obvious problem of single point failure as the bootstrap server is a single server maintaining all the information about the participants in the system and every node that joins the system needs to contact the bootstrap server first.
However this bottleneck is only for new nodes joining the system and failure of bootstrap server does not disrupt the normal read and write operations of the CATS.

### 2.2.7 Reads and Writes

In CATS every node in the system can handle read and write requests from the clients for any arbitrary key. For the sake of simplicity we will describe how these operations are performed in a fail-free environment.

Any read or write operation can be invoked by a client on any node in the system. However the operation can be handled and processed only through a coordinator (A coordinator can be any node replicating the range of records containing the requested record.). If the requested node is a member of the replication group, then that node will become the coordinator; otherwise the request is routed to any of the member of the replication group, which will become the coordinator. Further steps involved during Read and Write are described below.

![Message flows in Get and Put requests](image)

**Note:** Blue lines represent request/response from the majority

#### 2.2.7.1 Read

For a read operation the coordinator sends a Read request for the requested record key to all the members of the replication group. Once the coordinator receives the reply from the majority of group members (including the coordinator itself), it first ensures that the timestamp (version) of the records received from the majority is same before sending the response to the client. In case if there is a mismatch between the timestamps of the
responses received from the majority of replication group, the coordinator first perform a write (read impose), with the latest timestamped value, and once the impose is completed in the majority the coordinator replies with the value of the highest timestamp. The read impose ensures the linearizability of the read operations. Figure 2.4(a) shows the message flows for a read operation.

2.2.7.2 Write
For a write operation the coordinator first consult all the group members, the consultation process is a read request sent to all the group members. Once the coordinator receives the replies from the majority of the group members (including the coordinator itself), it then uses the largest timestamp received to generate the next timestamp, and sends the write request to all the group members with the new timestamp and new value. Once the coordinator receives replies from the majority of the group members for the write request it then respond with a success reply. Figure 2.4(b) shows the message flows for a write operation.

2.2.8 Failure and Join
2.2.8.1 Joining the Overlay Network
In order to join the system, the new node first contacts the bootstrap server with its own identifier; the assignment of node id is currently done by the node itself. On receiving the node join request the bootstrap server may apply one of the three possible policies as mentioned in the previous section. In response, the node either receives a reply with a bunch of information including successor list, predecessor and the group membership information that the node is responsible for; this response is received when the number of nodes reach to the threshold size for the first time. The other possible reply that a joining

---

**Figure 2.5: Steps involved in node join**

Note: View shown in blue are before the joining of new node, while the views in red are after the joining of new node. Step 8 is shown with altruistic behavior. In optimistic behavior the new node will only initiate the view change for split view while the view change for range (525,650] will be initiated by node 650 and view change for (400,525] will be initiated by node 525.
node can receive from the bootstrap server will contain any randomly chosen nodes. The
remaining algorithm for the joining node to be part of the overlay network is described
previously and explained completely in Chord [23].

2.2.8.2 Joining Groups
Once the node joins the overlay network it is still not able to participate in the read and
write operations, as the node is not responsible for storing any data ranges.

During the group join phase the joining node sends a group request message to the
successor node and in response the successor node reply with all its group membership
information. The joining node then proposes new group view for the groups in which it
should enter as a new member. There are two variations of group joining algorithm;
optimistic and altruistic. In optimistic behavior the new node only proposes new group view
for those group where the node id of the joining node falls in between the group record
range; whereas in altruistic behavior the joining node proposes in all the groups where the
node id is between the node ids of any two nodes in the group. Once the new group view is
installed to a member of old group, that node then pushes all the records for that particular
range from its local storage to the new joining node. Once the majority of the members of
old group have pushed the records to the new node and the new node has also installed
the new group view it is then able to participate in the read and write operations. Figure 2.5
shows the basic steps performed during a node join.

2.2.8.3 Node Failure
In CATS node failure in any case, either graceful termination or abnormal termination are
handled uniformly, and in any case the failing node does not need to do anything before
leaving the system. When a node stops responding to the messages for some duration, the
node is considered to be dead by other nodes. Failure detectors and the process of failure
detection is explained in 2.2.4. Here we will describe the different actions taken by a node
when failure of other node is detected.

In the event of a suspicion of predecessor node the link to the predecessor node is reset
for allowing other node to connect with. In the event of a suspicion of successor node the
next successor from the successor list is selected as the successor and a link with the new
successor is established.

The event of failure handling of group members are done in one of two possible ways,
optimistic and altruistic. In optimistic behavior only a special node, which is usually the first
node of the group in the network overlay order of nodes, is responsible for taking actions
on detecting the failure of group members. In altruistic behavior any node in the group can
take action on detecting the failure of group members. The action which is taken (in any
case, altruistic or optimistic) is that the node which detects the failure of group member
starts a new Paxos round and proposes a new view for the group(s) in which failure was
detected; the new view is chosen after removing the failing node and adding the next
suitable node from successor list which can replace the failing node. Figure 2.6 shows the
basic steps involved during and after a node failure.

Both the methods have pros and cons. In altruistic behavior any node can detect the
failure, and thus there is more chance of collision because several nodes will be trying to
propose at the same time. It can become even worse when the different nodes try to
propose different view, which can happen because of the unsynchronized successor list at
each node. The effect would be that there might be premature view changes which will
result in a delay to stabilize the system. On the other hand optimistic behavior is more susceptible to multiple or majority failures when the responsible node fails before detecting the failure of previous failed node.

Figure 2.6: Steps involved after detecting node failure.

Note: View shown in blue are before node failure, while the views in red are after recovering from failure. Step 6 is shown with altruistic behavior. In optimistic behavior node 650 will initiate the view change for (525,600) and node 525 will initiate the view change for (400,600).
Chapter 3: Disk Persistence

Before starting this section, it is important to mention that we will be referring in memory storage as the storage on any volatile storage which loses all the data on process crash or power failure. Similarly we will be referring disk storage for any kind of non-volatile storage this can be any SSD or HDD or any other permanent storage.

3.1 Motivation for Disk Persistence

CATS was originally designed as an in-memory data store. As CATS is implemented in JAVA, the natural and straightforward selection for storing records was to use JAVA Maps to store key/value pairs. The in-memory storage provides very high throughput and low latency for performing read and write operations, and can be used in several applications. However, there are still reasons for using disk-based storage instead of memory. Few such reasons are:

- Although memory is becoming cheap; still disks are more cheaper.
- Disks have more capacity than memory (RAM), and allow the size of the database to grow beyond the memory size.
- More chances of data loss in memory as compared to disks.
- Depending on the size of the data that a node needs to store, the recovery of process after failure will take longer time and will also increase the network traffic during recovery as the recovering process will need to collect the data from neighboring nodes.
- In systems like CATS, which support strong consistency, uses quorum-based reads and writes, and relies on replication for fault tolerance makes recovery from majority failure impossible without persisting the state variables onto a disk.

The main goal of having disk persistence in CATS is to store large amounts of data which can not be stored in the local memory of the nodes, and to store the state variables in order to enable the nodes to recover from failures.

In this section we will first look at the CATS design for having persistence, after that we will describe the architectures of two off the shelf solutions (BerkeleyDB JE and LevelDB) with which CATS persistence is implemented and evaluated, how they will impact on the read and write operations and recovery of nodes. In the last we will discuss about a slight improvement in the design using thread pools to improve the throughput of the system. All the experiments and their results to evaluate the performance of CATS with different DB implementations will be discussed in the Experiments and Results.

3.2 CATS Model for Disk Persistence

The Persistence layer for CATS has been designed in order to support any kind of underlying storage. This can be any RDBMS (e.g. Postgres, MySQL or Oracle) or any
document DB (e.g. MongoDB or CouchDB) or any local key/value datastores (e.g. LevelDB, BerkleleyDB, Tokyo/Kyoto Cabinet or Bitcask). A RDBMS or a document DB will not make sense to be used as an underlying storage for CTAS; therefore the most suitable and sensible choice is to use any local key/value datastore. Each node can be plugged with any local storage and any key/value storage, provided the CATS persistence interfaces have been implemented with the storage API. This pluggability ensures that the other CATS modules are totally independent of the underlying storage and also allow to test the performance of CATS with different off the shelf solution without impacting the code base. Different key/value datastore provide similar interfaces with little difference to interact with the datastore; however their underlying mechanism/algorithm to store records might be entirely different. Some of the well known mechanism log structured B+ trees [5], Log structured Hash Tables [17], and Log Structured merge trees [4]. The overall performance of the CATS will be highly dependent on which storage engine is used.

In order to have pluggable storage engine, we have created an abstract layer for persistent storage. The abstract layer consists of two interfaces which expose basic services which can be used by CATS for storing records and state variables. The interfaces has been implemented with BerkleyDB, LevelDB and with Java Maps. The Java map implementation is done for backward compatibility and to do comparison with other implementations. One interface in the abstraction layer is for reading and writing key/value records, and as well as for local range queries. The other interface exposes services for storing and retrieving various state variables in the underlying storage. CATS uses Java collections for maintaining the state variables, we have continued the same approach. Additionally, variables are also stored in the permanent storage whenever the state variables are updated. For in memory implementation the state persisting interface implementation is empty. Figure 3.1 shows abstraction layer for persistence and dependency of different CATS modules on the persistence layer. We have briefly described the RangeStore, Paxos and ItemStore modules in the previous section, and BulkTransfer module will be described in the later section.
3.3 BerkleyDB JE Architecture*

BerkleyDB JE is an ancestor of BerkleyDB, which is an embedded DB. BerkleyDB provides persistent, application specific data storage in a fast scalable and easily administered package [3]. BerkleyDB run in the application address space and there is no need of a separate installation. From the underlying architecture point of view databases in BerkleyDB JE are implemented as B+ Trees [5]. Every node in the database tree can be any one of the type of leaf node (LN), internal node (IN) and bottom internal node (BIN). Each of these are explained below.

![Figure 3.2: BerkleyDB database structure.](image)

**Leaf Node (LN)**

Every leaf node in the tree represents a key/value record and contains two piece of data. The key is the logical part which is actually referenced by the parent of leaf node and the value is stored as a reference of the log record, log sequence number (LSN) in a log structured storage.

**Internal Nodes (IN)**

Internal nodes are nodes that references other nodes. These nodes maintain a fixed size array of key/value pairs. The value at index “x” points to a subtree containing records for the range \([\text{key}[x], \text{key}[x+1])\).

**Bottom Internal Nodes (BIN)**

BINs are special type of internal nodes, these are the lowest internal nodes and each reference in the BINs point to a leaf node. Figure 3.2 shows the structure of the BerkleyDB database.

3.4 LevelDB Architecture*

*LevelDB provides a persistent key value store. Keys and values are arbitrary arrays. Keys are ordered and the records are stored according to a user-specified comparator or by a

---

*Complete documentation of BerkleyDB JE can be found in [3].

*Official implementation notes of LevelDB can be found in [2].
default comparator [2]. LevelDB is also an embedded DB and is plugged into the application and runs in the address space of the application, without any need for a separate installation. From the underlying architecture point of view LevelDB is an implementation of LSM (Log Structured Merge) trees [4], and the implementation uses a similar concept of a single tablet in Bigtable [24]. The records on the disk are stored in the key sorted order into a sequence of levels.

Every update (write, modify and delete) is written in a log file, which is converted into an SSTable [24] when it reaches a threshold size. An in memory copy of the log file, which is called memtable is also maintained and all read operations first consult the memtable. The newly created SSTables are added in the top level (level 0) of a B-Tree. When the number of level 0 files exceed a certain limit all level 0 files or some level 0 files with overlapping ranges are merged with all the overlapping level 1 files to produce new level 1 files and discarding all older files. The important distinction between level 0 files and files at other levels is that the files at level 0 may contain overlapping keys, whereas files at higher levels only have distinct non overlapping keys. For all levels > 0 when the total size of the data in the level exceed $10^4$ (MB) then a file in level L is picked and merged with all the overlapping files in level L+1 to produce new level L+1 files and discarding older files. During compaction a new SSTable is created when the current SSTable exceeds a threshold size or the number of overlapping entries of the current level L+1 file overlaps more than 10 files in level L+2, this is needed to lower the files to be merged for compaction of level L+1 files. In each level L > 0 the compaction is performed from a starting key after the ending key of the last compaction of level L. The compactions are managed in the background by separate threads. If we throttle the DB with write operations during the compaction, then we might produce a lot of level 0 files before moving them to level 1, which will increase the read latency as every read operations will need to consult the level 0 files. Figure 3.3 shows how SSTables are managed in LSM tree and how compaction is done.

![Figure 3.3: LevelDB LSM Trees and Compactions.](image)

### 3.5 BerkleyDB & LevelDB Implementation in CATS

In this section we will list implementation details of BerkleyDB and LevelDB in CATS.
During startup, every node opens or create two database instances. One is for maintaining state variables and other is for storing key/value records. In both DBs a database corresponds to a directory on the file system and all the contents are stored in this directory. The database name (directory) can be changed from a configuration parameter.

All write operations on both databases are done with immediate sync mode, which means that every write operation is committed to disk before replying. This behavior can be changed by setting “cats.db.sync = false”. By setting sync = false the data will only be written on the OS buffer and will be committed to the disk later by OS. The drawback of sync = false is that during the event of system crash, some recent writes will be lost. However no write will be lost in simple process crash (i.e. the machine does not reboot). The sync = true in both DBs have similar crash semantics to a “write()” system call followed by “fsync()”.

Each record, referred in CATS as item, is composed of three attributes; key, timestamp and value. The key of the underlying datastore is the byte array of the item key and the value of underlying store is the concatenated byte array of timestamp and value.

Every record is serialized into byte arrays before inserting in the DB. We have used kryo serialization [35] for serializing the records into byte array. BerkleyDB provides its own object serialization, however we have experienced that kryo serialization does better as compared to BerkleyDB’s object serialization, and it gives a uniformity in using any DB.

Bulk writes provide high throughput by batching several writes in one operation. In BerkleyDB they are done using transactions, while in LevelDB bulk writes are done using WriteBatch, which ensure atomic updates.

Range queries are provided for data stored locally on each node. In LevelDB range queries are implemented with LevelDB iterators, whereas in BerkleyDB range queries are implemented with cursors. The range queries can also be asked to return only the Key and timestamps of the record through a boolean parameter. The range queries in CATS are used for transferring data in bulk (discussed in Bulk Transfer) across the nodes. The total size of the records in requesting range can be very large and may require very large buffer to store the result. To avoid this problem range query only return the records starting from the first record in the requested range till either the last record in the requested range or till the total size of records exceed a threshold, whichever happens first. The result also includes the returning range [key of the first record in result, key of the last record in result], this allows the caller to request further records if required.

3.6 Performance Comparison of BerkleyDB & LevelDB

With our usage of LevelDB and BerkleyDB we have found some cases where these datastores have shown performance degradation.

For BerkleyDB we have noticed that the write throughput drops gradually over the period as the size of the datastore grows. We suspect that there are several reasons for decrease in write throughput. i) Writing random records causes much higher disk I/O load. This
becomes evident when the working set exceeds the size of the cache. ii) Writes are durable by default, which forces a lot of extra disk flushes. iii) Transactional environment is being used, in which case checkpoints cause a slowdown when flushing changes to disk. The other obvious problem is the performance of key scans; because of the log-structured design cache misses are more expensive. A cursor traversing the records, which misses in the cache will be forced to issue an I/O for any items not found in the cache, whereas a log-structured merged design or traditional on disk structure will fetch several items in the cache, amortizing the number of seeks for later reads.

For LevelDB we have also noticed that there are certain periods where there are huge spikes of latency while inserting the records. We suspect that this behavior is due to the partition scheduler [36] which LevelDB uses for scheduling merges. Partition scheduling leverages write skew to prevent long pauses during compaction [37], however they does not absolutely prevent long pauses in writes [37], [38] and shows huge peaks in latencies. Other obvious problem with LevelDB is that its architecture is designed to support efficient scans and high throughput on write while sacrificing on the performance of random reads. This is because in LevelDB keys and values are coupled together and causing the cache to be filled with keys and record unlike the BerkleyDB where values are decoupled from the records which resides in cache. This way LevelDB fails to give a stringent bound on the read latencies as often read operation might need more than one disk seeks. Another reason for high latency is that if there are lots of background operations while compaction, the compaction process can take substantially longer, which in turn might result in producing several level-0 files, and each read operation will need to incur an overhead of merging these level-0 files.

Performance results for LevelDB and BerkleyDB with CATS are discussed in Experiments and Results.

3.7 Improving CATS Throughput with Concurrency

In the initial results with LevelDB and BerkleyDB we found a significant difference from the in memory based model. We realized that there was a significant drop in the CPU utilization when used with LevelDB and BerkleyDB, and the difference in performance is due to the fact that the CATS operations (read or write) were mostly waiting for disk I/Os. We also noticed that with our initial non-thread design the I/O bandwidth of the system was very much under utilized. We have approached this problem as it is solved in other database systems by using threads to improve the overall throughput and utilize more I/O bandwidth. Java provides a complete set of utilities for creating concurrent applications. In our implementation we have used the Executor framework for invoking, scheduling and executing asynchronous read/write operations. Each CATS process creates a thread pool during initialization and assigns each read/write operation to one of the available threads. During these operation the executing thread makes an I/O operation and reply with the result asynchronously. Once the thread execution is complete the thread is returned to the thread pool. The size of the thread pool is also an important parameter to configure to achieve maximum throughput, however the optimum thread pool size may vary on different systems. The result section shows CATS throughput with different thread pool sizes.
3.8 Concurrency Pitfall in CATS and Solution

On one hand, concurrent operations have helped to increase the overall throughput of CATS significantly when used with a disk based storage (LevelDB/BerkleyDB) but on the other hand we have to be very careful as a negligent use of thread can violate the strong consistency property of CATS. Figure 3.4 shows an example where linearizability is violated when using a simple thread pool in each CATS process to handle read/write operations. In this example a single client sends two PUT requests one after the other (second request was sent after getting the response of the first request) for the same key $K$. In the first write $W1$ a value $V1$ is written in the majority of the replicas $R1$ and $R2$, while still in progress in $R3$ in a thread $T31$. After receiving the response another write request $W2$ for the same key with value $V2$ is initiated. $W2$ is completed in $R1$ and is done in parallel to $W1$ with value $V1$ in $R3$ in a thread $T32$, while in progress in $R2$. Assuming $W1$ completing after $W2$ in $R3$ ($T31$ writing after $T32$), will violate the linearizable consistency because a subsequent read request which gets a majority from $R1$ and $R2$ with read being parallel to $W2$ in $R2$ can return an old value $V1$.

![Figure 3.4: Linearizability violation with multiple threads.](image)

To guarantee the linearizability property in CATS, we argue that there should be no more than one parallel write operation with a particular key in all the running threads in a CATS process.

In order to ensure that there is at most one write operation for a particular key at any instance, we use Java concurrent hash map maintaining a lockable queue for each key $K$ of an ongoing write operation. Every write operation tries to acquire a lock on a lockable queue which is mapped by the write operation key. If the lock is already taken (in case of an ongoing write operation with the same key) the incoming write operation will block until the already running write operation releases the lock. This implementation allows concurrent writes with different keys to execute in parallel, while synchronizing only the write operations with same keys.
Chapter 4 : Failure Recovery

This chapter will discuss the failure recovery of CATS processes. First we shall look at the diskless recovery mechanism of CATS and see its limitations and then we will present the disk based recovery in which we will show the augmentations done in the existing algorithm to store the state variables and read them on recovery of process. Retransmission of messages are also added in order to cope up with message losses. We will also discuss a technique for improving liveness and in the end of the section we will formally prove that the disk based recovery holds the invariants for consistent quorums [22].

4.1 CATS Diskless Recovery

CATS diskless recovery mechanism relies on the inherent redundancy of the data items within a consistent quorum. When CATS is running on a diskless storage then all the state and stored data is lost, once the CATS process crashes. On recovery, these processes join the system as new nodes. In the first stage the joining node becomes part of the chord overlay network. Next a number of Paxos reconfigurations are initiated [22] either by the joining node or old members and in the end of each reconfiguration the joining node gets all the data from the old members of the consistent quorum. The joining node receives the data for each reconfiguration from a majority of old members and store the highest timestamped data in its local storage. Once the data for the reconfiguration has been stored the node then mark the downloaded range set as ready to serve for further read/write operations.

4.1.1 Limitation in Diskless Recovery

The ABD atomic register algorithm [39] and CATS reconfiguration algorithm relies on majority based quorum; once the majority is lost, neither reads/writes can be performed; nor any reconfigurations can take place. The major limitation for diskless system is that it will not be able to recover from a majority failure without sacrificing consistency. Other limitation is the need to download all the data items from neighboring nodes during recovery.

4.2 CATS Disk-based Recovery

The aim of CATS disk-based recovery is to eliminate the above mentioned limitations of diskless recovery. The disk-based recovery will allow the system to recover from any arbitrary number of failures, provided that enough failing nodes recover with all their disk persisted data items and states intact. The disk-based recovery is very similar to the recovery from network partition, as in both the cases processes have data and states preserved as before partition/failure 4.3.3.

The CATS disk-based recovery is divided into two portions. One portion deals with the persistence and recovery of the data items and the other is for state variables. The recovery of data items is straightforward. Each write operation is persisted on the disk
before replying and every read operation reads directly from the disk. With such a method of synchronous writing, the data items are readily available as they were before process crash. We will be focusing on the recovery of the state variables in the rest of the chapter.

4.2.1 Assumption

Every CATS process executes a deterministic algorithm, unless it crashes. Once crashed, process stops the execution of the algorithm until it recovers. After recovery the process can resume the execution of the same deterministic algorithm. A Process is also aware that it was crashed and is also allowed to execute a recovery procedure. We also assume that each process has a volatile memory and as well as access to a stable storage. The processes use unique primitives \( \text{store(key, item)} \), \( \text{retrieve(key)} \rightarrow \text{item} \) and \( \text{remove(key)} \) to write, read and remove data from the stable storage. The storage abstracts a Map where the items are updated if an item with the same key is already exists.

4.2.2 Recovery Implementation

We have augmented the Paxos reconfiguration algorithm in order to store the Paxos state variables on the disk. In addition to the Paxos state we also store all the local configurations (in which the process is a member), the list of ready and busy ranges and the history of configurations which have been installed by the process. On recovery the state variables are retrieved from the stable storage and loaded in the memory based data structures. In algorithms [1,2,3] we have highlighted the augmentation needed to store the Paxos state for recovery and also for retransmission of messages. The retransmission of messages is necessary to counter message losses, either due to network failures or due to the destination process failure. The retransmission ensures message delivery when the network is healed and/or destination process recovers. Algorithm 4 shows the recovery procedure that the CATS process execute. This recovery is straightforward where all the state variables are retrieved from the stable storage and loaded in the memory based data structures.

4.2.3 CAS Failure & Problem on Relaying Coordinator

Retransmission

CAS (compare-and-swap) is an atomic primitive to achieve synchronization. We use CAS primitive in CATS to ensure the linearizability of reconfigurations in a key replication group. A reconfiguration \( (v \Rightarrow v') \) is installed only when the current configuration is \( v \).

![Figure 4.1: CAS Failure scenario.](image)

CAS Failure occurs when a member of a previous consistent quorum receives a P3A (Algorithm 2) for a reconfiguration \( (v \Rightarrow v') \), but due to any reason it has not yet received one or more previous reconfigurations \( (u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_N \Rightarrow v \text{ where } N \geq 0) \). In
order to maintain linearizability, the current received reconfiguration \((v \Rightarrow v')\) is buffered until the previous reconfigurations \((u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_N)\) are installed in the correct order.

CAS Failures can occur due to several reasons; different network latencies between different nodes, message losses and node failures. CAS Failures are normally solved with retransmissions. The coordinator of the reconfiguration periodically retries, to cope with message losses as shown in Algorithm 1. However in some cases, only relying on the coordinator for retransmitting the reconfiguration messages might not be enough to guarantee liveness. We give an example of such a case below, where an old reconfiguration is stuck on a node, waiting for coordinator retries.

Consider an initial stable configuration with 5 nodes 0, 2, 3, 4, 5 as shown in Figure 4.1. A new node with node id 1 joins the system. The joining node proposes a split range configuration change from \((0,2\{2,3,4\}) \Rightarrow (0,1\{1,2,3\} \mid (1,2\{2,3,4\})\). Assume the reconfiguration is installed on node 2 and 3 while the message to node 4 is lost, and the coordinator node 1 dies, leaving node 4 with an old configuration \((0,2\{2,3,4\})\). After some time node 2 dies as well; the failure of node 2 is detected by node 3, which proposes a reconfiguration \((1,2\{2,3,4\}) \Rightarrow (1,2\{3,4,5\})\). However since node 4 is stuck with an old configuration, the new configuration will be kept on hold while waiting for the old reconfiguration, which may not come if node 1 is permanently dead. Such a situation weakens the liveness property of the system and is more prone to leading towards unavailability of consistent quorums.

Algorithms \([5,6]\) show the view installation procedure and handling for CAS recovery. The main idea for recovering from a CAS failure is to actively pull the older reconfigurations and install them pro-actively rather than only relying for the coordinator to resend the lost messages. We will include these consideration while re-stating the liveness properties next.

### 4.2.4 Algorithms

**Algorithm 1: Reconfiguration Coordinator**

**Init:** p1Acks[i] ← ∅, p2Acks[i] ← ∅, p3Acks[i] prop[i] ← 0,

\[\text{rec[i] } \perp \quad \text{∀ consensus instance } i\]
\[\text{ph } \leftarrow 0 \quad \text{∀ current Paxos phase 0-4}\]

1: \text{on (Propose:}(v \Rightarrow v')\text{)) do}

2: \text{rec[i] } \leftarrow (v \Rightarrow v') \quad \text{∀ proposed reconfiguration}

3: \text{i } \leftarrow v, \text{ph } \leftarrow 1

4: \text{send (P1A: i, prop[i]) to all members of } v

5: \text{scheduleTimeOut()}

6: \text{on (P1B: i, ACK, ts, rec, v)} do

7: \text{p1Acks[i] } \leftarrow \text{p1Acks[i]} \cup \{(ts, rec, v)\}

8: \text{if } v = \text{consistentQuorum(p1Acks[i]) then}

9: \text{ph } \leftarrow 2

10: \text{r } \leftarrow \text{highestProposedReconfig(p1Acks[i])}

11: \text{if } r \neq \perp \text{ then rec[i] } \leftarrow r

12: \text{send (P2A: i, prop[i], rec[i]) to all nodes in } v

13: \text{scheduleTimeOut()}

* Algorithms \([1,2,3]\) are taken from \([22]\). Changes for storing state variables and message retransmission are in blue.
on (P1B: i, NACK; v) V (P2B: i, NACK, v) do
prop[i]++  ▷ retry with higher proposal number
ph ← 1
send {P1A: i, prop[i]} to all members of v
scheduleTimeOut()
on (P2B: i, ACK, v) do
p2Acks[i] ← p2Acks[i]  {⋃ v}
if v = consistentQuorum(p2Acks[i]) then
ph ← 3, store(i, coordinator)
send {P3A: i, rec[i]} to all members of v
scheduleTimeOut()
on (P3B: i, v) do
p3Acks[i] ← p3Acks[i]  {⋃ v}
if v = consistentQuorum(p3Acks[i]) then
ph ← 4, store(i, coordinator)
send P4A: ⟨ v' ; rec[i] ⟩ to new member (v' / v)
on (TimeOut) do  ▷ proposer
resendCurrentPhaseMessages()
on (Recovery: c) do  ▷ proposer
reinitialize(c)  ▷ re-initialize vars with t
resendCurrentPhaseMessages()
Algorithm 2: Current Group Memebhr
on (P1A: i, p) do  ▷ acceptor
if p ≥ rts[i] ∧ p ≥ wts[i] then
rts[i] ← p  ▷ promised
store(i, acceptor)
reply {P1B: i, ACK, wts[i], rec[i], view}
else reply {P1B: i, NACK, view}
on (P2A: i, p, ( ⟨ v ⇒ v' ⟩ )) do  ▷ acceptor
if p > rts[i] ∧ p > wts[i] then
wts[i] ← p
rec[i] ← (v ⇒ v')  ▷ acceptor
store(i, acceptor)
reply {P2B: i, ACK, view}
else reply {P2B: i, NACK, view}
on (P3A: i, ( ⟨ v ⇒ v' ⟩ )) do  ▷ learner
installView(v ⇒ v')
storeLocalRange(), storeViewLog()
reply {P3B: i, view}
send {Data: v', data} to new member (v' / v)
Algorithm 3: New Group Member
on (P4A: v', rec) do
installView(v'), storeBusy(v')  ▷ makes v' busy
storeLocalRange(), storeViewLog()
on {Data: v', data} do  ▷ from old members
dataSet[v'] ← dataSet[v'] ∪ {data}
reply {DataAck: v'}
if G = consistentQuorum(dataSet) then
storeHighestItems(dataSet)
removeBusyStoreReady(v')  ▷ makes v' ready

Algorithm 4: Process Recovery

Init: acceptors ← ⊥, proposers ← ⊥, views ← ⊥, busyR ← ⊥, readyR ← ⊥, viewLog ← ⊥, waiting ← ⊥

1: on (Recovery) do ▷ main process
2: while p ← retrieveProposer() ≠ ⊥ do ▷ proposers
3: proposers ← proposers ∪ {p}
4: trigger (Recover: p)
5: while a ← retrieveAccepter() ≠ ⊥ do ▷ acceptors
6: acceptors ← acceptors ∪ {a}
7: while v ← retrieveView() ≠ ⊥ do ▷ local view
8: views ← views ∪ {v}
9: while b ← retrieveBusyR() ≠ ⊥ do ▷ busy ranges
10: busy ← busy ∪ {b}
11: while r ← retrieveReadyR() ≠ ⊥ do ▷ ready ranges
12: readyR ← readyR ∪ {r}
13: while l ← retrieveViewLog() ≠ ⊥ do ▷ view log
14: viewLog ← viewLog ∪ {l}
15: while w ← retrieveWaiting() ≠ ⊥ do ▷ waiting view
16: waiting ← waiting ∪ {w}

Algorithm 5: View Installation with CAS Recovery

1: on <installView(v=>v′)> do:
2: if v=>v′ ∈ viewLog then return ▷ installed
3: if v /∈ views then ▷ CAS Failure
4: waiting ← waiting ∪ {v=>v′}
5: send (CASRecoveryReq: v) to (v / self)
6: else
7: install(v′)
8: if v′=>? ∈ waiting then
9: installView(v′=>?)
10: on (CASRecoveryResp: u=>v) do ▷ CAS Recovery process
11: if v /∈ viewLog then ▷ not already installed
12: proposeReconfiguration(u=>v)

Algorithm 6: CAS Recovery Handler

1: on (CASRecoveryReq: v) do ▷ Old members
2: if ∃ u / u=>v ∈ viewLog then
3: reply(CASRecoveryResp: u=>v)
4: else reply(CASRecoveryResp: null)

4.3 Proof for Correctness

4.3.1 Safety

Lemma 1:
After a successful (terminated) reconfiguration (v => v′), at most a minority of nodes in v may still have v installed.

Lemma 2:
For a particular key replication group, there cannot exist two disjoint majorities (w.r.t. view size N) with consistent views, at any given time.

1 Taken from Lemma-1 from [22].
2 Taken from Lemma-2 from [22].
Corollary 1
In a sequence of reconfigurations of a replication group (\(v \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \ldots \Rightarrow v_n\), where \(N \geq 1\)), no two views will be same.

Proof
views consists of a tuple \(<g, i>\), where "g" is the set of nodes in the view and "i" is the version number.

For consecutive configurations: A re-configuration only takes place in the event of new nodes joining the system or nodes leaving the system. A reconfiguration (\(v \Rightarrow v'\)) is proposed by the coordinator only if \(v.g \neq v'.g\) (i.e. the group of nodes in the proposing configuration \(v'\) is different from the group of nodes in the current configuration \(v\)).

For non consecutive configurations: Version numbers are monotonically increasing, so even if the group of nodes for two non contiguous configurations are same their version numbers will always be different.

Corollary 2
In a key replication group, a node recovering with an old view (\(v\)) will either belong to a minority or there was no reconfiguration (\(v \Rightarrow v'\)).

Proof
A successful (terminated) reconfiguration (\(v \Rightarrow v'\)) ensures that \(v'\) is installed in majority of the correct nodes and at most a minority including the failed ones will have \(v\) lemma-1. By corollary-1 there can never be a sequence of reconfiguration (\(v \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \ldots \Rightarrow v_N \Rightarrow v\)), thus the recovering node will either be part of a minority or possibly be in majority but only if there was no reconfiguration (\(v \Rightarrow v'\)).

Lemma 3
In a key replication group, any arbitrary node failures and recovery with an old view (before failing), will never lead to disjoint consistent quorums.

Proof
The reconfiguration algorithm essentially requires a majority of correct nodes with an old view \(v\) for a reconfiguration (\(v \Rightarrow v'\)) to succeed and after termination, a majority of nodes will have \(v'\) installed while at most a minority, including the failed ones, will have \(v\).

Let \(v'\) be the view of last successful (terminated) reconfiguration in the replication group.

Case 1 (\(v = v'\)): In this case there are no reconfigurations while the recovering node was dead, then from lemma 2 there can not be disjoint consistent quorums.

Case 2 (\(v \neq v'\)): In this case there has been a reconfiguration while the recovering node was dead, then from corollary-2 the recovering node will be in at most one minority group, with an old view installed, in any of the reconfigurations (\(v \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \ldots \Rightarrow v'\)).

From lemma 1- 3 and corollary 1 and 2 we prove theorem 1 which is enough to show linearizability.

Theorem 1
For any key replication group, no two disjoint consistent quorums may exist simultaneously. Therefore, any two consistent quorums always intersect.

Restatement of Lemma-3 from [22].
4.3.2 Liveness

**Lemma 4**
Provided a consistent quorum of the current view \( v \) is accessible, a group reconfiguration \((v \Rightarrow v')\) will eventually terminate.

**Lemma 5**
*In a totally connected network.* If \( u \) be the current view for a key replication group at some node \( p \) and provided that, majority of nodes in each of the linearized sequence of views \((u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_N \Rightarrow v)\) where \( N \geq 0 \) are available, a group reconfiguration \((v \Rightarrow v')\) on \( p \) will eventually terminate.

**Proof**

**Case 1 (\( u = v \)):** From algorithm 5 reconfiguration will be applied immediately making \( v' \) the current view on \( p \).

**Case 2 (\( u \neq v \) and a majority in each of the sequence of installed views \( u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_N \Rightarrow v \) is accessible):** From installView \((v \Rightarrow v')\) in Algorithm 5 if \( v \) is not installed in the node then it will initiate the CAS recovery, in which it will eventually propose the reconfiguration \((u_N \Rightarrow v)\) after receiving CAS recovery response. Given that majority of nodes available in \( u_N \) the coordinator \( p \) will be able to receive reconfiguration for \((u_N \Rightarrow v)\). Recursively the node will eventually be able to get reconfiguration \((u \Rightarrow u_1)\), which it will be able to install. Once \( u_1 \) is installed the node will also be able to install all the pending views up till \( v \), and eventually getting able to install reconfiguration \((v \Rightarrow v')\), resulting in \( v' \) being the current view on \( p \).

**Lemma 7**
Provided a consistent quorum is accessible, put and get operations will eventually terminate.

From Lemma 4-7, we have theorem 2.

**Theorem 2**
*In a totally connected network; For any key replication group, provided a consistent quorum is accessible, any put and get operations issued, and any reconfigurations will eventually terminate.*

4.3.3 Network Partition/Merge ~ Node Failure/Recovery

In our implementation network partition and node failure are similar events and similarly network merge and node recovery (with all its state before crashing) are similar events. Node failure(s)/recovery can be considered as a special case of network partition/merge and vice versa. This fact can also be understood by understanding partition tolerance (P) in CAP. According to Gilbert and Lynch [13], “In order to model partition tolerance the network will be allowed to lose arbitrary many messages sent from one node to another”.

Using this definition we first compare network partition and node failure by understanding “what causes partitions?” [40]. Basically two things cause network partition, the first is obvious - a network failure, for example a faulty switch can cause the network to partition such that the nodes in two partitions are unable to communicate. The second is less

---

4 Taken from Lemma-4 from [22].
5 Taken from Lemma-5 from [22].
obvious, but fits the definition, and that is node failures. A node failure will also render messages sent to the failed node as lost messages. In an asynchronous network there is no bound on the time between a message sent and message received, and thus making it impossible to distinguish between node failure and lost message. A single node failure will partition the node from the rest of the network and failure of several nodes partition all of them from the rest of the network.

Now to compare the network merge and recovery, we see that in network merge after partition both the merging partitions have nodes with certain states, these states may or may not have been evolved since the network was partitioned. Similarly a node recovery with its state before crashing will be exactly same as a network merge after partition, where states of the nodes in one partition (the live nodes) might have evolved and the state of the node in the other partition (failed nodes) will be the same.
Chapter 5: Bulk Transfer

In CATS, replication is used for fault tolerance and recovery. In order to maintain a consistent replication group in the event of node failures and joining or leaving of nodes in the system, data records need to be transferred across the nodes; Bulk transfer component is responsible for maintaining a consistent replication group in the event of nodes failure and joining of new nodes in the system.

This chapter will start with providing details for the range query and selected items query that are added in the CATS persistence layer to get data items in bulk from other nodes; next bulk transfer component design is explained with details of the working of the component; the recovery mechanism for the bulk transfer component is explained next; an implementation details of the component describing the data structures involved in designing the component; an algorithm for bulk transfer component is presented and finally some thoughts on how consistency is maintained during the downloads and also during the recovery of nodes after failure.

5.1 Addition in CATS API

5.1.1 Local Range Query

The local range query is a wrapper that uses the APIs provided by the underlying storage to perform range queries. The data structure (TreeMap) or key/value data items (BerkleyDB, LevelDB) used for storing the data items, provide inherent support for performing range queries in some way. In all kinds of storage that have been used in CATS, the data items are stored in key sorted order. BerkleyDB provides cursor; LevelDB provides iterator and Java TreeMap provides iterator and sub-maps which are used to perform range queries.

The local range query in CATS allows a node to make range queries only on its local data. The support for making range query across the nodes is being done in a different project 7.1.6. The local range query currently is not provided as an external API, because it is only being used internally by the bulk transfer component.

The API for the local range query is:

\[
\text{RangeResult getItemsInRange(Token start, Token end, IntervalBounds bound, boolean isValReq, long maxSize, long time)}
\]

This method tries to get the data items stored in the local storage which lies between the start and end token. The bound parameter specifies whether the start or end or both are inclusive or exclusive in the result; isValReq parameter controls whether key-timestamp (false) or key-timestamp-value(true) are required. The last two parameters provide extra restrictions; maxSize imposes that the number of bytes of the resultant records, key + timestamp + value (if required), do not exceed the maxSize and the time parameter restricts the method to return after the given amount of time has expired. The restriction parameters are very critical in performing a range query which may take hours and may...
consume all the memory, depending on the number of data items and their aggregate size in the requested range. The restriction parameter, prevents these situation by controlling the amount of memory that will be used for the result buffer and time that will be spent on running the query. This also means that the result of the range query will not always return the whole requested range.

The RangeResult structure includes the range that is being returned from the method along with a list of data items for the returning range. The returning range enables the caller to check whether the result is a partial range or a complete range. In case of partial result the caller can call the method sequentially over and over again, each time with subsequently smaller sub-ranges (from the last record return in the last call till the last record requested in the last call), until the resultant range is equal to the requested range.

5.1.2 Query for a bunch of selected Items

In addition to the local range query, another method has been added to get the data items for a given bunch of keys. The API for this query is:

\[ \text{List<Item> getSelectedItems(Collection<Token> tokens, long time)} \]

This method tries to get the data items for the given collection of keys within the given interval of time. If the given time period expires and the method is unable to collect all the data items for the given collection of keys from the underlying storage, only a partial result is returned as a list of data items. It is the responsibility of the caller to ensure that all the data items for requested keys are returned and calling the method again with the remaining collection of keys.

5.2 Component Design Details

Bulk transfer component is implemented in CATS as a Kompics component. The component has a dual responsibility, as a provider (sending data items to the requesting nodes) and as receiver (downloading data items).

The provider role is fairly simple. In this role the component is responsible to respond the range downloading requests and downloading of selected key items, from other nodes. On receiving the range download requests, the component fetches the requested range from the persistent storage using the local range query 5.1.1 and respond to the downloading request with the data items returned from the local range query on the requested range. The range download request contains a parameter showing whether key-timestamp-value is required or only key-timestamp without values are needed. On receiving the request for selected items, the component fetches the key-timestamp-value of the requested items from the persistent storage using API for fetching selected data items 5.1.2 and reply with the requested data items. It is important to note here that the reply of the range queries from API may not contain the whole requested range for reasons explained in 5.1.1. Same is the case for the query for selected data items 5.1.2.

As a receiver the component is responsible for initiating the downloading and fetching of data items from the group members. The component receives the download notifications from nodes that have completed a reconfiguration. The notification message is to notify the node, that it has been included as a new member in a group. The notification message contains the reconfiguration which tells the component which data range the node needs to
fetch from the previous group members (group before reconfiguration). The actual downloading of the data items from the group members is deferred until notifications from majority of the group members are received for a particular reconfiguration. Once the notifications from the majority of the group members have been received, it is safe to initiate the download. The simplest way here could be to start downloading the range from all the members but this might cause a huge amount of redundant data traffic across the network. In order to avoid redundant data across the network, the range is divided into smaller sub-ranges. In the current setting, the range that needs to be downloaded is divided into a number of disjoint sub-ranges equal to the replication degree (or the number of sources from where the data needs to be downloaded). In order to maintain consistency of the replicated data items in the system, the data items must be received from the majority before installing in the current node. In the normal case most of the data items will be consistent among the replicas and downloading the complete data items (key-timestamp-value) from majority still causes a lot of redundant network traffic. This problem has been addressed with a two phase download solution. In the first phase, the complete downloading range is divided into 3 smaller sub-ranges, where 3 being the replication degree. For each sub-range, key-timestamp-value is pulled from only a single source, while only key-timestamp is pulled from the rest of the sources. This way network traffic is lowered significantly and only key-timestamp is downloaded redundantly. This download phase is shown in Figure 5.1.

![Figure 5.1: Range download strategy in bulk transfer. Note: Range download for a replication degree of 3.](image)

Downloading values from single source, however may lead to inconsistent state of data items in the system. To avoid the inconsistency, another phase of download is performed. In this phase all the key-timestamps of every item in every sub-range, from all the sources are compared and the ones with highest timestamps are picked. If any of the item with highest timestamp was not received from the source where value for that data item was received, then that data item needs to be fetched again from the source which returned the highest timestamp. In case, there are more than one such sources with highest timestamp, anyone can be picked. For each sub-range, a set of data items which needs to be downloaded again from other sources is sent again in the next phase. Once the reply of these picked data items are also received for all the sub-ranges of a particular reconfiguration range, the download is considered complete and the downloaded range becomes ready to serve the requests for data items in that range.
5.3 Failure and Recovery

5.3.1 Recovery of Sources During Download

The Bulk transfer component is fault-tolerant and can handle failures like the main CATS component. The component is equipped with an eventual consistent failure detector (◇S) [41]; all the nodes from which the component is downloading, are monitored by the failure detector. In addition to detecting failures, the detector also detects the recovery of a failed node.

In the event of failure of a source, a reassignment is required for all the active key-timestamp-value downloads of the sub-ranges, going on the failing node. No reassignment is needed for the active sub-ranges downloading key-timestamp, as other sources are already downloading key-timestamp or key-timestamp-value for those sub-ranges. In order to make a reassignment, all the downloaded sub-ranges that have not yet been completed are discarded. Although, discarding of downloaded data will require to download the same data items again from a different source but it allows to keep the design of the component simple. A new source among the remaining group members is picked to download a sub-range, that was assigned to the failed node. The selection of the node for reassignment is done by selecting the node that has least number of key-timestamp-value downloads going. If there is no source available, which will mean all the group members for the particular downloading range has failed, the reassignment also fails and the downloading stops. To continue the downloads one or more failed sources need to restart again. An important case here would be when, as an example, for a replication degree of 3 two sources fail subsequently and causing all the ranges getting downloaded from only one source. This however will not be able to complete the download, as for the download of a sub-range to complete, at least key-timestamp must be downloaded from a majority of the group members with at least one of them must have downloaded key-timestamp-value.

In the event of a source recovering after a crash while the sub-ranges originally assigned to it are not yet downloaded, the design choice that has been taken is to reassign the downloads back to the recovering node. This decision is taken again for a simple design for the component by having less if-then-else choices in the implemented code.

5.3.2 Recovery of Downloading Nodes

In order to support failure recovery of nodes it is very essential to recover not only the Paxos states but also the state of the bulk transfer component before the node failed. The minimum, that is required for the bulk transfer component for recovery is to restart the ongoing downloads and recover the download notifications that it has replied before crashing. This is necessary otherwise the notifications for download will be completely lost and rendering the reconfigurations and replicated state machine to stuck. For the sake of simplicity any ongoing downloads for a reconfiguration, will be restarted on the recovery and all records that were downloaded for that reconfiguration before failure will be downloaded again.

5.3.3 Message Losses

The previous section described the actions taken in suspicion of permanent failure of a source node; another kind of failure that can happen is message failures. Due to the
downloads in a request response manner in smaller chunks, a single message loss from
the source will render the download to halt completely as the receiver will be waiting for the
reply of the request that has been lost. To solve these kinds of message loss problems,
message retransmission is used with gracefully handling duplicate replies.

5.4 Implementation Details

The Bulktransfer component is implemented as a Kompics component in CATS. The
component is initialized by the main CATS component during the initialization of the node.
The component maintains several data structures and is configured to listen several
events. The main data structures that the component needs to maintain are.

5.4.1 Data Structures

Monitoring Nodes
A structure for failure detector, keeping track of which nodes are being monitored by the
FD.

Reconfiguration Log
A list of reconfigurations that have been completed by the component to avoid duplicate
reconfiguration notifications.

Transfer Instances
A Hashmap , storing the notification messages against the reconfiguration instance. This is
needed to keep track of the sources that have sent a download notification for a particular
reconfiguration. The downloading is started once notification is received from majority.
Once all the data for a reconfiguration has been downloaded the corresponding entry is
removed and a corresponding entry in the reconfiguration log is added.

Downloading Sources
A Hashmap, storing the list of sources against the reconfiguration. This is needed during
the reassignment of downloads.

Data Downloads
A slightly complex data structure composed of nested data structures to maintain the
ongoing downloaded data. On the top level this structure looks like
Hashmap<reconfiguration, Hashmap<sub-range, RangeDownload>>. The internal
Hashmap stores the downloaded status of the sub-ranges that are currently in progress for
a particular reconfiguration.

The RangeDownload itself is a complex data structure which keeps track of what data
items have been received from which source for a particular sub-range download.

5.4.2 Events

The events that the component process is shown in the Bulk transfer algorithm 5.5.

5.5 Algorithm

Bulk transfer algorithm ensures that the consistency of the data items will not be violated
and also ensures that the download will be completed provided the node is connected with
the sources for long enough time. The properties for the bulk transfer algorithm are.

1. Initiation
In a reconfiguration, the new node can start to download data items once majority of the
group members have installed the reconfiguration.

2. Termination
Provided enough sources are available and connected for long enough time, the download
will terminate.

3. Linearizability
All the data items installed should maintain linearizability.

Modification in Algorithm 2 Reconfiguration: Current group member

18: send (NotifyDownload: self, v=>v') to v'/v

Algorithm 7: Downloading Node

Init: xfers ← ⊥, kts ← ⊥,
1: on (Recovery) do
2: while p ← retrieveXfers() ≠ ⊥
4: if isMajority(xfers[p.v]) then ▷ Notif. from majority
5: pullRange(p.v) ▷ chunk and pull
6: on (NotifyDownload: s, v=>v') do
7: if v ∉ xfers then
8: xfers[v] ← xfers[v] ∪ {s}
9: else xfers[v] ← xfers[v] ∪ {s}
10: store(v, xfers[v])
11: if isMajority(xfers[v]) then
12: pullRange(v) ▷ chunk and pull
13: on {KTResp | KTVResp:s,v,R,kt,data} do
14: kts[R] ← kts[R] ∪ {(s,kt)}
15: if data ≠ null then installData(data)
16: if R.end = r.end then ▷ Got complete range
17: fetchNewItems(v, kts[R]) ▷ High timestamped
18: else send {KTReq | KTVReq: (r.end, R.end, v) to s
19: on (NewItemsResp:s,v,kt,data) do ▷ new items
20: installData(data)
21: removeBusyStoreReady(v) ▷ makes v ready
22: on (NodeSuspicion:node) do ▷ source suspicion
23: reassignKTV(node)
24: on (NodeRecovery:node) do ▷ source recovery
25: takeoverKTV(node)
26: pullRange(v)
27: subR = chunkRange(v) ▷ {(src, subrange})
28: for ∀ s ∈ v.group
29: send {KTVReq: self, subR[s], v} to s
30: send {KTReq: self, subR[s], v} to (v.group / s)

Algorithm 8: Source Node

1: on {KTReq | KTVReq:s,R,v} do
2: \( (r, kt, data) \leftarrow rangeQuery(R) \quad \triangleright \text{might be a partial response} \\
3: \text{if KTReq then} \\
4: \quad \text{reply} (\text{KTResp}: self,v,R,kt, null) \text{ to } s \\
5: \quad \text{else reply} (\text{KTResp}: self,v,R,kt, data) \text{ to } s \\
6: \text{on (NewItemReq} : s,v,\text{keys) do} \\
7: \quad (kt, data) \leftarrow \text{fetchSelectedItems(keys)} \\
8: \quad \text{reply} (\text{NewItemResp}: self,v,kt, data) \text{ to } s \\

5.5.1 Correctness

The initiation property follows from the fact that the old group members send the download notification once they have installed the reconfiguration along with downloads locally and downloads are started only after receiving notification from majority.

Termination is guaranteed by the following facts:

1. Message retrials are used to cope with message losses and temporary failures.

2. Failure detector allows to re-assign downloads in the event of permanent node failure. It also allow to detect the recovery of nodes and thus allowing the downloads to restart again once a source recovers after failure.

3. State variables are stored on a permanent storage to recover from node crash and allowing any ongoing download before crash to restart again.

Linearizability is guaranteed by downloading the key-timestamp of the data items from majority and installing them in the local storage with the one having highest timestamp, which is enough to ensure linearizability.
Chapter 6 : Experiments and Results

6.1 Experiments with Recovery
CATS is built on top of Kompics framework [33]. Kompics also provide a deterministic simulation environment. We have used the deterministic simulation mode to generate several experimental scenarios with node joins and failures. Particularly we have created several experiments where majority of nodes fail and then enough nodes are recovered to establish the majority. In each experiment the success of the experiment was based on satisfying the consistent quorum invariants [22] for a long enough time, after waiting for a sufficient amount of time for the ongoing configuration changes to terminate. We have also tested for the correctness of data items stored in each node, for this in each experiment after the initial configuration of nodes is completed, we have loaded the system with 1000-10000 arbitrary key/value records, and in the end after satisfying the invariants the same records are read again to ensure the data items are intact in the system. The verification of data items roughly ensures the correctness of our bulk transfer component.

An example experiment scenario is given below

\{
\}

The first string is the initial configuration, as in the given example a system of 10 CATS process will be created with the given process ids. SXXX inserts a simulated delay for XXX milliseconds. LXXX-YYY loads the system with YYY random records within a key range of (0,XXX], FXXX is for the node XXX to fail/stop, JXXX is for the node XXX to join/recover and VXXX-YYY for verifying the records and has the same semantics as LXXX-YYY.

6.2 Performance Experiments
The rest of the experiments that we will discuss are performed using YCSB (Yahoo Cloud Serving Benchmark) [42]. YCSB is a framework and common set of workloads for evaluating the performance of different “key-value” datastore or “cloud” servicing stores. It consists of two parts.

- YCSB Client. An extensible workload generator.
- The core workloads, a set of workload scenarios to be executed by the generator.

We have implemented a YCSB client for CATS and we have created several workloads to measure different aspects of the system, with different database engine for CATS. We have
compared LevelDB, BerkleyDB and Memory based Java maps storage for CATS for different workloads.

**Test Environment**

All the performance tests were done on the SICS local cluster, which consisted of seven nodes, each equipped with a 2.6 GHz 12 core machine with 32 GB RAM and 1000 Mbps ethernet connection.

**Test Setup**

We have used a single machine, dedicated for generating load from YCSB Client, the number of client threads were changed to vary the load, while CATS processes were running on other nodes. In all the tests except for the scalability and elasticity tests we have used 3 CATS instances running on different machines. Also the number of server threads will be 16 except where explicitly specified otherwise. Each CATS process was given an initial heap size of 1GB and a maximum upto 12GB.

6.2.1 Load/Insert Performance

In this test the performance of random writes are compared. 10M records 256 bytes each were inserted from a 32 threads YCSB client. The comparison shows a smooth throughput for LevelDB as compared to BerkleyDB. BerkleyDB has higher throughput in the beginning but drops quickly once the number of records grow.

![Figure 6.1: Insert performance of CATS with DB engines.](image)

6.2.2 Effect of Record Size on Disk Size

The following graph compares the disk storage size of BerkleyDB and LevelDB. In each test 1M records were stored from YCSB in CATS running on 3 nodes. Also the replication degree was 3, which means that each node was a replica of others. Therefore a total of 1M records were stored on the local storage of each CATS instance. The total disk size was changed by increasing the size of the individual record while keeping the number of records constant to 1M. The bar charts show that there is a slight overhead on disk usage in BerkleyDB. Since Kryo serialization was used to serialize the records into byte arrays for both the DBs, therefore the overhead is not because of different serialization mechanism. It
also shows that the disk usage in LevelDB is very efficient as the \((\text{number of records} \times \text{record size})\) matches with the actual disk size.

6.2.3 Effect of Record Size on Latency

This graph compares the latency of the CATS different storage options and also shows the effect of record size on latency. The test was conducted on a CATS system running on 3 different nodes, while load was sent from a different machine running YCSB client with 32 non-throttled threads. In each test case 1M operations, with 70% read, 15% insert and 15% update were sent from the client.

6.2.4 Latency Variance with Time

The below graph shows the latency variance with time. In these tests 10M operations of 256 byte record size with 70% read, and 30% update were sent from the client with 32
threads. The DB was also pre-loaded with 1M records of 1KB record size. The graph shows that the latency decreases over time, the main reason for the decrease in latency is because of the OS disk cache, as the reads in later time are fetched from the OS disk cache and due to a quite large OS disk cache all the database disk blocks were cached after some time. For LevelDB case the block size was set to default 4KB, whereas for the BerkleyDB there is no read block size for individual read operation and we believe that OS default read block size will be used which is usually 1KB.

6.2.5 Effect of Server Threads on Latency

The graph shows the effect of varying server threads on the latency of read, write and update operations. In these tests 1M operations of 1KB record size with 70% read, 15% inserts and 15% update were sent from a single client with 32 non-throttled threads. The DB was also preloaded with 1M records of 1KB record size. The graph shows that increasing the server threads are more effective in BerkleyDB than LevelDB. Also for BerkleyDB, insert operations have lower latencies than the update operation, whereas for LevelDB insert and update operations behave similarly. The reason is that the update operation generated from YCSB have keys which were already inserted previously and for insert operations the keys are unique keys which were not inserted previously. As the write operation in CATS is a read modify operation, therefore for BerkleyDB insert operations fail to find the key from the in-memory B+ tree and returns without issuing any disk seeks, but
for update operations a disk seek is required to fetch the item. On the other hand for LevelDB since, key and values are stored together so searching for either already existing key or a non-existing key will require disk seeks.

6.2.6 Effect of Varying Load (Client Threads) on Latency

The graph shows the effect of varying load (client threads) on the latency of read, write and update operations. In these tests 1M operations of 1KB record size with 70% read, 15% inserts and 15% update were sent from a single client with varying non-throttled threads. The DB was also preloaded with 1M records of 1KB record size. The graph shows that for BerkleyDB and LevelDB the latency of the operations doubles by increasing the load after 8 client threads. This is because each server has 12 cores, therefore increasing the number of clients over 12 will consume all the available cores making the latency to increase in directly proportional with load (number of client threads).

6.2.7 Scalability with Write Intensive and Read Intensive Load

The following graphs shows the scalability of the system. In these tests 1M operations of 1KB record size were performed in a preloaded system of 1M records of 1KB record size. For a write intensive load the percentage distribution of operations were 40% read, 30% inserts and 30% update and for a read intensive load the distribution was 90% read, 5% insert and 5% updates. In these tests the load is also increased proportional to the number of nodes increased i.e. if the number of node doubles then the load also doubles. For the write intensive load the scaling is linear for LevelDB, BerkleyDB and memory based Java maps, whereas for a read intensive workload only LevelDB showed linear scaling.
6.2.8 Bulk Transfer

6.2.8.1 Effect of Chunk Size on Download Time.

The following graph show the effect of chunk size on download time. In these tests download were done on a preloaded system with 2M records of 1KB. Each test was run in two scenario, in one scenario no operations were going on during download, and the other scenario was with an ongoing operations with a workload of 70% read, 15% insert and 15% update. The graph shows that for BerkleyDB increasing the chunk size over 132 KB increases the download time significantly, this is because of the poor scan performance in BerkleyDB and with bigger chunk size the scans will run longer.

6.2.8.2 Download Time with Data Size.

The following graph show the effect of database size on download time. In these tests download were done on a preloaded system with (1,2,4 and 8)M records of 1KB. Each test
was run in two scenario, in one scenario no operations were going on during download, where as the other scenario was with an ongoing operations with a workload of 70% read, 15% insert and 15% update. The graph shows very large download time for BerkleyDB because of slow scans due to log structured storage, where as LevelDB is as good as memory or even better, this is because in memory based downloads, a snapshot of the downloaded range from Java maps are created, which results in some overhead.

6.2.8.3 Effect of Chunk Size on Latency and Throughput

The following graph show the effect of chunk size on the latency of ongoing operations. In these tests download were done on a preloaded system with 2M records of 1KB, while a load for 2M records of 1KB size with 70% read, 15% insert and 15% update were going on while download. The graph shows that for BerkleyDB and LevelDB varying chunk size does not have significant effect on latency during downloads, while for java maps latency increases with downloads going on with larger chunk sizes.

Figure 6.10: Effect of download size on download time with different DB engines.

Figure 6.11: Effect of chunk size on latency with different DB engines.
6.2.9 Elasticity

The following graph shows the elasticity of the system. The test was started with 3 CATS nodes and after every 600 sec a node was added till 6 nodes and after that the nodes were taken out after every 600 sec till there were 3 nodes. In the test 600 sec delay was enough for the reconfiguration and downloads to complete. During the whole test a read intensive workload with 90% read, 5% inserts and 5% update was being performed with a client of 64 threads. The graph shows major spikes when the nodes leave the system for all the storage options. This is due to the reason that when a node leaves the system the client threads have ongoing operations on those nodes and for a long timeout value set for clients, they will stuck until timeout and thus increasing the latency.

Figure 6.12: Elasticity of CATS with different DB engines.
7.1 Future Work

A number of improvements and new features are planned for the future:

7.1.1 Evaluation of bLSM with CATS

Recently research conducted at Yahoo has produced bLSM [37], a Log Structured Merge (LSM) tree with the advantages of B-Trees and log structured approaches: (1) Unlike existing log structured trees, bLSM has near-optimal read and scan performance, and (2) its new “spring and gear” merge scheduler bounds write latency without impacting throughput or allowing merges to block writes for extended periods of time. As a first step a JNI wrapper of the C++ API has been developed and work is in progress for evaluation with CATS.

7.1.2 System Wide Recovery

The system wide recovery will allow the whole system to gracefully shutdown and restart again. In theory the node recovery algorithm can recover the nodes, but in addition to that some form of coordination among the nodes needs to be implemented to inform the nodes about system shutdown and thereby telling the nodes not to do any further reconfigurations. Some sort of global view of the system also needs to be stored, which will be used to recreate the same system view during startup.

7.1.3 Merkle Trees

In the current implementation all the data items are downloaded by the new member from old members during a reconfiguration. Merkle tree will allow to detect the inconsistencies in the local replica and only the updated data items will needed to be pulled from the old members. Merkle trees will greatly reduce the time to recover and also will help to handle churn to some extent.

7.1.4 Thread Pool in Bulk Transfer Component for Sources

By introducing threads in the provider role of bulk transfer component, it will be able to entertain multiple download range items and selected item requests simultaneously. As only read operations are performed therefore parallelizing several read operation will help in alleviating the problem of high latency read operations.

7.1.5 Load Balancing

Load balancing allows adjusting of load on the nodes in the system. Load balancing can be
implemented by gathering information about the load on the nodes and adding or removing
virtual nodes as in (Dynamo, Riak) [15], [17]. Each physical node will be capable of joining
the system with several different nodes using different identifiers.

7.1.6 Distributed Range Queries

Work is going on in a separate project to implement consistent range queries across the
nodes. The local range queries implemented for bulk transfer component might be utilized
in implementing distributed range query on top of that.

7.1.7 Object Storage Model

Work is going on in a separate project to implement a storage model for storing complex
objects. The storage model will allow arbitrary objects to be stored and retrieved. The
model will provide Object relational mapping (ORM) like facility to persist Java objects.

7.1.8 Distributed Transactions in CATS

Distributed transactions can be built on top of consistent quorums as consistent quorums
provide a consistent view of the replication group and replicated data items.

7.1.9 Using CATS for HDFS Name node

Another experimental project that is being considered, is to use CATS for providing a
scalable, highly available and distributed solution for the HDFS name nodes. HDFS name
nodes are single point of failure and lacks high scalability and high availability. CATS can
be used to store the meta-data stored by the name nodes providing high scalability and
high availability.

7.2 Conclusion

The thesis presented extensions and feature added in the existing research project CATS,
a distributed key value data store that uses consistent quorums to provide linearizability.
First accomplishment of the thesis is a persistent layer which has been implemented with
LevelDB and BerkleyDB as a storage engine for the CATS node. Due to different design of
the engines each one has dominance over the other in different use cases. LevelDB is
suitable where scans and range queries are required, where as BerkleyDB is more suitable
for a use case requiring random point reads. The design of the persistence layer also
allows to incorporate other engines, and it will be very interesting to try other engines like
bLSM as a lot of research is still going on in this area.

Second accomplishment is the implementation of node recovery algorithm, which allows
the system to recover even after majority of the nodes were crashed and then later
recovered in some replication group. The node recovery is done by storing the state
variables and Paxos group membership on the persistence storage.

Third accomplishment of the thesis is implementation of a bulk transfer component that
provides efficient replication of data items in the replication group. An extension in the
CATS API has been made by adding local range queries that is essential for the bulk
transfer component.
Finally, several optimizations and improvements were discovered during the course of the thesis, while some of those have been implemented; introducing concurrency by threads to improve throughput of the read and write operations, reducing the amount of download data during reconfiguration and recovering from CAS failure when reconfiguration coordinator is unavailable.
References


