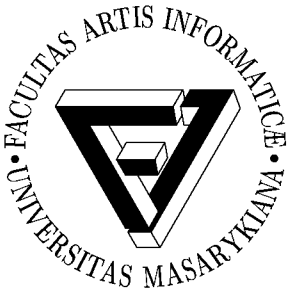
MASARYK UNIVERSITY
FACULTY OF INFORMATICS



Analysis and Testing of Distributed NoSQL Datastore Riak

MASTER THESIS

Bc. Zuzana Zatrochová

Brno, Spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Zuzana Zatrochová

Advisor: RNDr. Andriy Stetsko, Ph.D.

Acknowledgement

Most of all, I would like to thank my supervisor RNDr. Andriy Stetsko, Ph.D. for his numerous advices during our consultations within the past two years. Moreover, I would like to thank my colleagues in Y-Soft corporation, namely Marta Sedláková, Ján Slezák and Martin Hanus for their technical assistance during the development of the thesis. Nonetheless, I would like to thank my family for their neverending support.

Abstract

The goal on the thesis was the analysis of the consistency, latency and availability trade-offs in the NoSQL distributed database system Riak. The analysis included a theoretical study of the Riak mechanisms that provide the consistency and availability guarantees. We provided evaluation of existing metrics and proposal of new metrics for the measurement of the data consistency. Based on the results of the theoretical analysis, distributed testing application simulating interaction between database and clients was implemented. The application is able to model experiments in network partitions. The experimental results were produced by the designed application and evaluated with the defined metrics.

Keywords

Distributed databases, CAP theorem, consistency, Riak, NoSQL

Contents

1 **Introduction** 1
2 **Riak** 3
2.1 *Data Model* 3
2.2 *Data and Nodes Distribution* 5
2.2.1 Consistent Hashing 6
2.2.2 Gossip Protocol 11
2.3 *Data Storage and Replication* 12
2.4 *Consistency, Availability and Partition Tolerance* 18
2.4.1 Fault Tolerance 19
2.4.2 Eliminating Consistency Violations 20
2.4.3 Conflict Resolution 23
3 **Metrics** 25
3.1 *Consistency* 26
3.2 *Latency* 33
3.3 *Availability* 35
4 **Implementation** 37
4.1 *Riak Validator* 39
4.1.1 Planner 39
4.1.2 Test Worker 41
4.1.3 Validator 45
4.2 *Riak Interceptor* 46
4.2.1 Target Interceptor 47
4.2.2 RPC Worker 48
4.2.3 Riak Database 49
5 **Experiments and Evaluations** 51
5.1 *Probability of Request Types* 52
5.2 *W-R Quorum* 54
5.3 *Partitions* 59
6 **Conclusions** 63
A **Erlang** 65

1 Introduction

A distributed system is a collection of communicating components located at networked computers [4]. The communication and coordination among components is provided through message passing. Consequently, the most important characteristics of distributed systems are the concurrency and independent failures of components and the lack of global system clock.

A database system is an organized collection of related data. A distributed database system (DDBS) is a collection of multiple logically interrelated databases [26]. In contrast to centralized databases located on a single computer, databases of the DDBS are distributed over network. Hence data are spread among number of network locations as well.

DDBS replicate data to improve efficiency and fault-tolerance [26]. Consequently, the system must ensure propagation of updates to all data replicas and selection of an appropriate data copy for a user. If a failure occurs, the system must handle unfinished updates on the recovery. Moreover, if transactions are supported by the database system, a synchronization of requests becomes much harder problem then in centralized database systems. On the other hand, distributed systems provide improved performance, easier system expansion and reliability as a natural consequence of the data replication [26].

NoSQL is a term that represents non-relational databases. The motives behind NoSQL are relational database (RDBMS) limitations in processing a huge amount of data [29]. Although, RDBMS provide strict data consistency and ACID properties, the strong characteristics may be useless for particular applications. In contrast to RDBMS, NoSQL databases provide lower consistency guarantees in favour of a higher throughput and a lower request processing latency. Moreover, they use simple data structures with low complexity. As a result, they scale horizontally more easily seeing that sharding expenses of the RDBMS are avoided [29].

The main theoretical concept behind NoSQL databases is a CAP theorem [16]. The theorem states that in presence of partitions, there is a trade-off between availability and consistency. In addition, recent

research shows that in the networks without failures, the trade-off exists between the latency of requests and the data consistency.

Riak is an open-source distributed NoSQL database system [30] that runs on multiple nodes in the network. It uses a simple key-value model for storing data. Mechanisms used in Riak are based on Dynamo [10], set of techniques applicable to highly available key-value datastores. Dynamo exploits Consistent Hashing [20] for data distribution, Vector clocks [25] for data versioning, Hinted Handoff [10] for temporary failures, Merkle Trees [24] for anti-entropy and Gossip protocol [10] for failure detection. In addition, Riak extends Dynamo standard key-value query model by the implementation of MapReduce [9], secondary indexes and full-text search [30].

In this thesis, we are concerned with the analysis of Riak database properties and design of experiments that provide the insight on the Riak behaviour. We begin with a study of the database concepts followed by the evaluation of key Riak properties such as consistency, latency and availability. Subsequently, we propose metrics for the future analysis of the properties. Following the theoretical study, we design a testing environment for the needs of experiments. We conclude the thesis with the series of experiments proposed in the theoretical evaluation. Consequently, the methodology exploited in the thesis may be used in the analysis of other distributed NoSQL datastores.

Mechanisms of Riak datastore system are described in Section 2, supplemented by the main theoretical concepts behind NoSQL datastores. We studied latency, consistency and availability in the following Section 3 and proposed the metrics for their evaluation. An application simulating client-database communication is described in Section 4. Finally results of the experiments are provided in Section 5. The concluding thought are presented in Section 6.

2 Riak

Riak is a distributed datastore implemented in Erlang functional programming language. Erlang is usually exploited in distributed and fault-tolerant applications [12]. An Erlang application consists of lightweight processes that do not share memory and communicate only through a message passing. There is no access to a common critical data, hence processes are fast without any synchronization. Common process behaviours are grouped and implemented in Open Telecom Platform (OTP) framework. It is a set of modules implementing generic behaviours of processes. A brief demonstration is given in Appendix A

2.1 Data Model

Riak is a collection of databases, running on a physical servers called *nodes*. The group of nodes is called a *cluster*. *Client* is a user applications running on a server and communicating with the Riak. *Data model* is a logical organization of data within a single node. It specifies an identification mechanism of a data within the node.

Figure 2.1 shows a data identification structure of a Riak node. A *key-space* is a set of strings that characterize distinct groups of data on a node. A *bucket* is a key-space member with the specified configuration. The configuration, referenced as *bucket properties*, specifies a behaviour of requests made on bucket related data. The behaviour includes a consistency level, replication factor, conflict resolution or backend used as low-level storage. Each data belongs to a single bucket. The data can be managed by any node in the cluster. Therefore, changing the bucket properties increases costs of network communication since new configuration has to be distributed to all nodes in the system. In addition, there may be an arbitrary number of buckets stored on a node.

A *key* is an identifier of the unique data within a bucket. The <bucket, key> pair uniquely determines a single data-object within the entire database system. However, two objects with equal keys but different buckets constitute different entities. To address the data, a

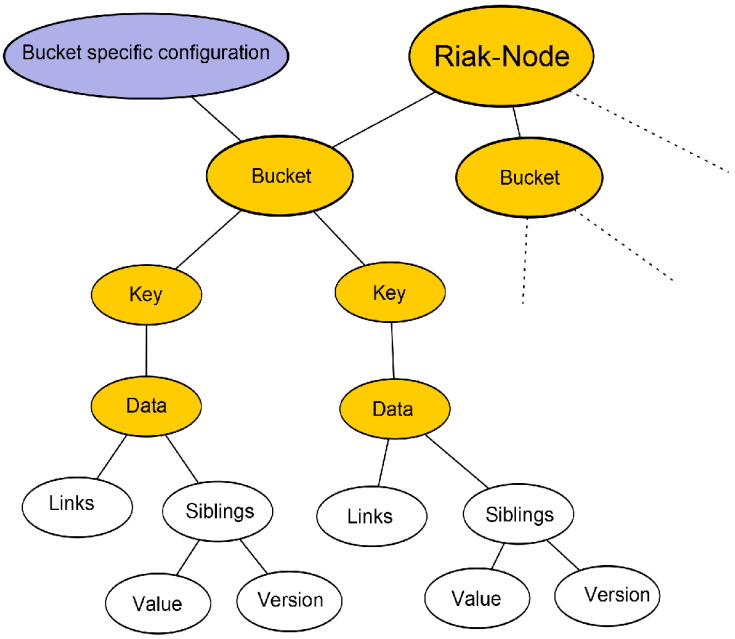


Figure 2.1: Riak Data Model.

client needs information about the corresponding key and the bucket. Riak uses keys in binary or string representation.

A data is associated with a *value*. The value (string, list, file, etc.) is stored to the data by a client. The client either creates data with the value or updates the value of the existing data. Each value has a *version* that identifies a time the data was updated with the value. The time may be represented in logical [21] or real-time fashion. Riak exploits the logical-time based concept called vector clocks [25]. However, vector clocks specify only partial ordering on events in distributed systems. If two data versions cannot be ordered, clocks conflict and multiple versions of the data are created. Different versions, called *siblings*, are stored to the data. If a client reads the object with multiple versions, the returned content is determined by the configuration of bucket properties. Either siblings are compared by real-time clock update times and latest value is chosen or the decision is left to the client. The client then chooses a preferred value of the data from all siblings.

In addition, Riak supports multiple one-way relationships between

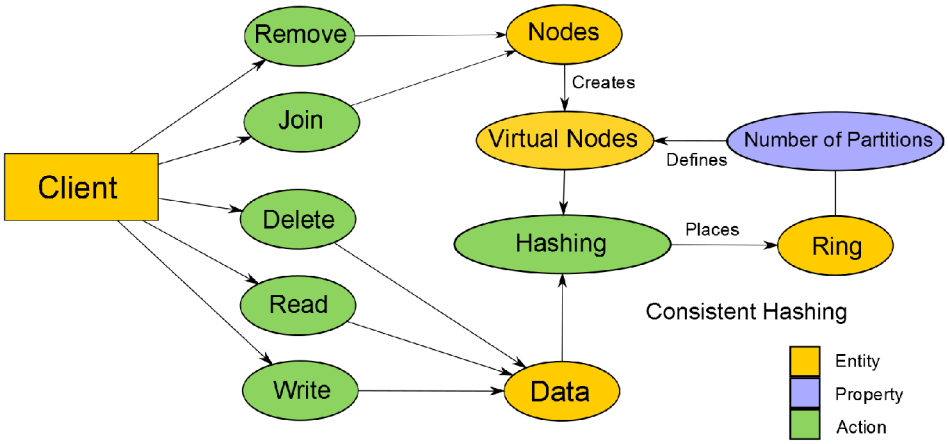


Figure 2.2: Riak Architecture.

different data. The relationships are expressed through links between related data. They are used in a link-walking search mechanism [30]

2.2 Data and Nodes Distribution

Data and nodes distribution represents relationships between data and nodes of Riak database. It specifies a storing mechanism of the data to a corresponding node in the cluster. Moreover, it determines the process of joining and removing nodes from the cluster.

Figure 2.2 represents a process of the Riak data and nodes distribution. The Riak distributed system consists of nodes, clients and data. Clients invoke join and remove operations on nodes. Each node creates a set of *virtual nodes*, each with inherent storage space. The virtual nodes are logically organized into a *partitioned ring*. The organization is based on *consistent hashing mechanism* [20]. The number of virtual nodes of a node is determined by a number of ring partitions. In addition, clients send read, write and delete requests to the cluster. Requests are executed on the particular data. Exploiting the consistent hashing, data are assigned a place on the ring. The place of the ring uniquely determines the node in the cluster.

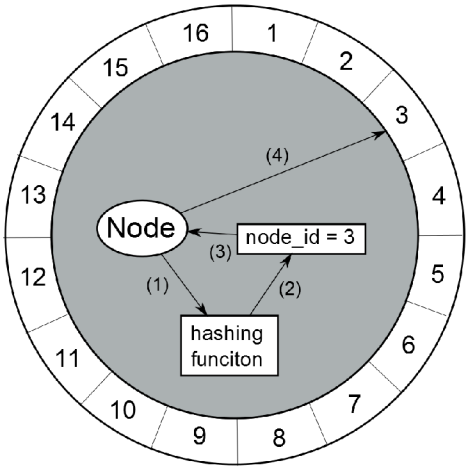


Figure 2.3: Consistent Hashing.

2.2.1 Consistent Hashing

Generally, the consistent hashing is a mechanism that determines mapping between data and nodes in distributed systems. A ring is a list of consecutive integer index values. In Figure 2.3 there is a ring with index values from [1,16]. Exploiting an uniform hashing function, nodes are assigned identifiers (IDs). Identifiers are mapped to indexes in the ring. In Figure 2.3 the node uses hash function (1) to get the ID (2,3). The ID is mapped to the corresponding ring index (4). The node with the highest ID is the predecessor of the first node on the ring. Similarly, data keys are hashed using the identical hashing function and mapped to the indexes on the ring. Each node is responsible for the keys in the interval between its ID and the successor ID. In Figure 2.3 the node is the first in the cluster hence it is successor to itself. It is responsible for all keys in whole ring interval. If a node is removed from the cluster, the predecessor node takes over the interval. If a node joins the cluster, the interval of its predecessor is split at ring index equal to new ID value.

Riak uses modified version of the consistent hashing [30]. At the beginning, the hashed key-space interval is equally partitioned. The

number of partitions is power of two and equal to the value in the Riak configuration. The list with pointers to the partitions is created. Each ring partition is linked to a single virtual node according to a claiming algorithm [31]. Accordingly, we use the term partition and virtual node interchangeably.

Anytime a node joins a cluster, ring partitions are reassigned. If the node is a singleton, it creates new cluster and takes all the partitions. If a node joins the existing cluster, it contacts a *seed node*. The seed node is a manually specified, fully-operating and non-faulty cluster node. The joining node retrieves a cluster state from the seed and recomputes new number of virtual nodes for each node in the cluster. Subsequently, it invokes partition claiming process.

The process of partition claiming is coordinated by the node that changed the cluster membership. The coordinator computes claiming phase for each node in the cluster. The claiming phase is encountered by every node owning less than precomputed number of virtual nodes. During the phase, the node claims existing partitions from the cluster until its amount of virtual nodes is satisfactory. Riak aims to assign partitions evenly. Each node should be responsible for Partitions/Nodes virtual nodes whereas maintaining 1/Nodes part of the key-space. However, minor variations are allowed in a claiming strategy implementation.

There are two basic claiming strategies, shown in Figure 2.4. Both strategies are partially-based on a parameter *n_val*. The parameter specifies the number of consecutive partitions on the ring that should be maintained by different physical nodes. The default value is equal to the replication factor [30]. If two partitions of the same physical node are distant less than *n_val* positions on the ring, then the partition with higher ID is called conflicting. After the claiming process is over and the conflicting partition exists, all virtual nodes are redistributed using the round-robin algorithm. Redistribution is executed irrespective of the strategy type. Even after the redistribution, there may exist some conflicting partitions in the tail of the ring. The anomaly happens if the number of nodes in the cluster is lower than the *n_val* value.

- **First strategy:** A node computes new number of partitions per

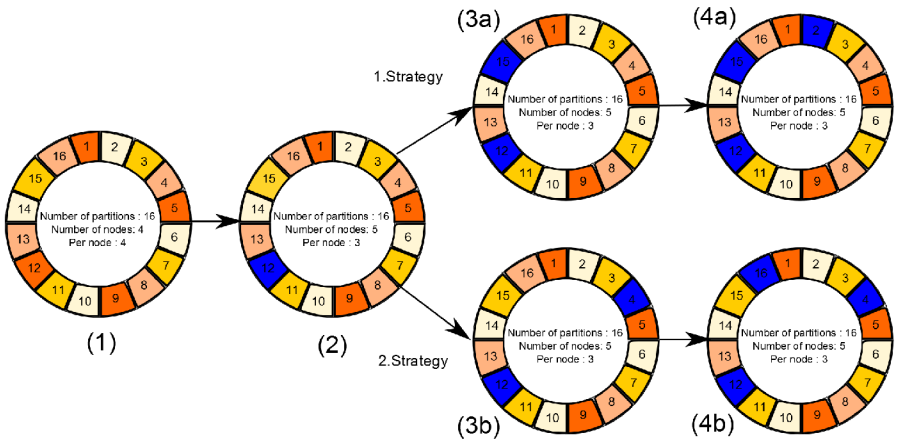


Figure 2.4: **Claimant Strategies:** Different colours of partitions represent distinct Riak nodes.

a node. The number is equal to Partitions/Nodes. If a conflicting partition exists, the node claims the partition as its first partition. Otherwise the node randomly picks its first partition from the ring. In each iteration, the node tries to claim another partition from a ring. Ring is iterated consecutively, starting at the partition *n_val* positions away from the last claimed partition. The iterated partition is claimed if its owner maintains redundant partitions in respect to the newly computed number. Otherwise, following partition is investigated. The process repeats until the joining node has claimed enough partitions. For example, in Figure 2.4 the claiming node with zero virtual nodes (1) randomly picks partition number 12 (2). The n_val = 3 and iteration starts at position 15. The partition 15 belongs to the node with four virtual nodes. Since the average number of virtual nodes is three, the position is available and claimed (3a). The next iteration starts three position further. Similarly, the investigated partition belongs to a node with redundant number of virtual nodes, thus it is claimed (4). The claiming phase ends because the node acquired precomputed average amount of partitions.

- **Second strategy:** A node computes new number of partitions per a node. The number is equal to Partitions/Nodes. The node recursively computes the longest interval between its already claimed partitions. The first partition is claimed randomly. Otherwise, the node claims the partition in the middle of the longest interval. Note that the claimed partition may belong to a node that maintains exactly the computed number partitions. As a result the amount of partitions maintained by different nodes may not be equal after claiming process. After the coordinator computed the claiming phase for all nodes in the cluster, some of them might own less or more than average number of partitions. If the variation is greater than two, all partitions are redistributed using Round Robin algorithm. For example, in Figure 2.4 the claiming node with zero partitions also picks random position number 12 (2). The longest interval between the position 12 and itself is the rest of the ring. The partition 4 is in the middle of the interval, thus it is claimed (3b). Thereafter, the longest interval between position 4 and 12 is either <5,11> and <13,3>. The interval <13,3> is randomly chosen and the middle position 16 is claimed (4b). Note that position belongs to the node that has three virtual nodes assigned and it has no redundant virtual nodes to provide. The claiming phase is over because the node acquired demanded amount of partitions. Although, average number of virtual nodes is not balanced, the variation is only one virtual node and partitions distribution is valid.

The advantage of the first strategy is more uniform load-balancing. A node always detaches only a redundant partition. As a result nodes maintain uniform number of partitions. A drawback is less uniform partition distribution. The node claims partitions separated by *n_val* virtual nodes. If value of *n_val* is small enough, claiming process produces an interval of considerable length where the node does not own any partition. On the other hand, the second strategy distributes the partitions evenly, claiming the middle of a greatest interval. However, average amount of virtual nodes may be disrupted when unavailable partition is claimed. Moreover, both strategies suf-

fer from inefficient redistribution of partitions if the number of physical nodes is low in respect to the *n_val* parameter [31].

A node removal is another type of cluster membership change operation. The departure of the node is related to a permanent failure or a node redundancy. The permanent failure is usually followed by a replacement of the crashed node with a new one. The partition ownership remains unaltered. Partition links of the crashed node are passed to the new node. Nonetheless when node becomes redundant it is removed by an administrator. If the node is removed, it coordinates the claimant phase of remaining nodes in the cluster with recomputed average number of virtual nodes. The node is destroyed only after all its partitions with data were distributed.

Data are assigned to the nodes according to the <bucket,key> identification. The pair is hashed to an index equal to a value from the 160 bit interval. Riak uses SHA function from Erlang crypto module that computes message digest with length of 160 bits. The interval represents the ring, and partitions correspond to the equally sized subintervals of the interval. The number of subintervals is determined from the number of partitions in the Riak configuration. Data are distributed to the node responsible for a partition with subinterval equal to the data index.

The consistent hashing mechanism used in Riak has better load-balancing efficiency compared to the standard mechanism [10]. The load-balancing efficiency is defined as ratio of average number of requests handled by each node to the maximum number of requests handled by the most overloaded node. Moreover, since ranges of partitions are fixed, data maintained by a virtual node are fixed too. As a result, recovery and transition of a node is faster. On the other side, in contrast to the standard node ID hashing, changing the node membership requires the coordination of partition reassignment. Furthermore, it is recommended to maintain around ten partitions per a node [30]. For a fixed number of partitions specified in Riak configuration, only several cluster sizes satisfy the recommendation. The cluster size is the number of physical nodes in the cluster. If the amount of nodes in the cluster rapidly increases or decreases, the number of partitions must be reconfigured. Otherwise, Riak does not exploit advantages of consistent hashing mechanism. The reconfiguration may cause all data to be redistributed. In conclusion, Riak

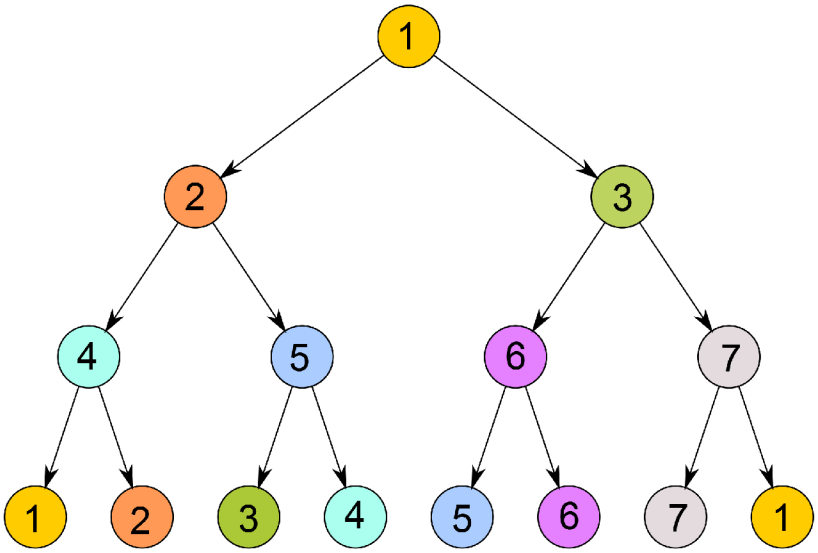


Figure 2.5: Gossip Protocol.

partitioning mechanism is recommended for stable clusters, without dynamic membership changes.

2.2.2 Gossip Protocol

After the claiming process, some partitions changed their owner but data are still located on the obsolete node. Consequently, data need to be moved to their new owners. The process of the moving consists of gossiping and handoff phases. The gossip is a process of spreading new cluster state, called a *ring state*, to the rest of the cluster. The handoff phase transfers data to the new node. It is started only after all nodes received new ring state through the gossip protocol.

The gossip protocol is used to communicate a *ring state* around a cluster. The ring state is the mapping table between indexes and partitions and contains also bucket properties and other meta-information. The communication is invoked whenever the ring state is changed. Moreover, each node uses the protocol to periodically forward the state in case any node missed previous updates. The period is speci-

fied through *gossip_interval* configuration option.

The gossip protocol specifies a *gossip group* for each node in the cluster. Consequently, new ring state is forwarded to the nodes in gossip group. Two strategies are implemented in Riak.

- The simple gossip protocol randomly assigns a node from members of the cluster to the gossip group each time the protocol is invoked.
- The recursive gossip protocol creates a binary tree from the cluster members. The process is shown in Figure 2.5. The cluster consists from seven members 1 to 7. Nodes are ordered into a list [1,..,7]. A member with list index *i* is mapped to children with indexes *2i* and *2i+1*. If 2i or 2i+1 is higher than the number of cluster members, the indexes are mapped from beginning of the list repeatedly. The example demonstrates that inadequate amount of nodes may result in a child equal to the parent node (node 6 and 7). However, there is always at least one distinct node receiving the gossiped information. Although, the structure of binary tree depends on members of the cluster, the tree is not recomputed on cluster membership change. It is rather computed each time the gossip is invoked. As a result it is not held in the state of the node process.

After all nodes observed partition changes, data transfer process begins from obsolete nodes. During the data hand-off, new owner handles requests for a data. Due to the transfer delay, data might not be stored on new node yet.

2.3 Data Storage and Replication

Data storage represents the communication between a client and Riak database. The communication between Riak nodes and clients is provided directly or through a load balancer. The load-balancer may exploit several approaches. In the first approach, the balancer is situated in the network layer and monitors the system performance.

According to the performance information, it directs requests to the least engaged node. In another approach, the balancer distributes requests to nodes using the round-robin algorithm.

An official Erlang client does not implement any load-balancer [30]. Therefore, the client has to specify a seed node for the communication. In the experiments in Section 5, a proprietary implementation of round-robin algorithm is used to assign the nodes to the clients. If a faulty node is contacted, the client receives an error report.

Data replication influences the data availability and consistency. The replication in Riak is defined by a process from a request creation to an acknowledgement of its realization. The process is distinct for different request types. In Riak, there are four basic types of request: write, read, update and delete. For the sake of simplicity we describe the process only for write and read requests[1].

Generally, the data replication is used to increase the data availability. A *primary replica* is a node responsible for data according to the data distribution function. If the primary node fails, its data become unavailable for future requests. Replication solves the problem since multiple nodes maintain data replicas. Subsequently, a client is able to access the data from *secondary replicas*. Secondary replicas are nodes determined from a replication mechanism. Data are available if at least one replica is non-faulty.

There are two basic types of replication [35].

- **Active replication** is a decentralised technique where all replica nodes are treated as primaries. Therefore, the nodes receive the same sequence of client requests. The data consistency is violated if nodes receive requests in different order. High per-replica processing and increased amount of consistency violations are main disadvantages of the active replication. In order to implement the strong consistency, a synchronization mechanism among replicas has to be provided. The mechanism increases a latency of request processing. On the other hand, active replication is simple and transparent to failures since requests are simultaneously processed on other replicas.

1. Update and delete requests are special types of the write request. A delete request may inflict additional problems, however, it is omitted in both thesis theory and experiments.

- **Passive replication** is a centralised technique where a single primary-replica handles all requests. It periodically forwards updates to secondary replicas. If primary fails, some secondary node takes over its responsibilities. The high reconfiguration cost is a drawback of the passive replication. However, data consistency is easily implemented since all requests are processed on the primary node at first. Hence the order of requests is equal on all replica nodes and it is determined by the primary. Difficulties arise when primary does not manage to forward some updates before crashing. When the node is recovered, the system must provide mechanisms to consistently order requests executed by the old and new primaries.

Riak exploits modified version of the active replication. Primary nodes are computed from an *ordered list*. The ordered list is a list of partitions, lexically ordered by their indexes, where the first partition of the ordered list is different for each request. Data associated with the request are mapped to a partition with interval containing the hash(<bucket,key>). The partition is places on the first position in the ordered list. Successive partitions are determined by the lexical order.

Primary nodes are first N partitions in the ordered list where number of replicas (N) is a *replication factor*. A request is always forwarded to N nodes. However, primary nodes may be temporarily unavailable. Therefore, a *preference list* is estimated for each request. The preference list represents the first N non-faulty virtual nodes taken from the ordered list. The request is sent to all nodes contained in the preference list. If some primary nodes are faulty, preference list contains also secondary replica nodes. Secondary replicas are temporal backup for the failed nodes.

A coordinator is a mediator of a communication between clients and data-replica nodes. It is a process implemented as an Erlang gen_fsm_state_machine that belongs to a physical node with single Riak database. The node is an owner of the coordinator. The machine consists of four main states: prepare, validate, execute and waiting. In the beginning of the communication, a client forwards a request

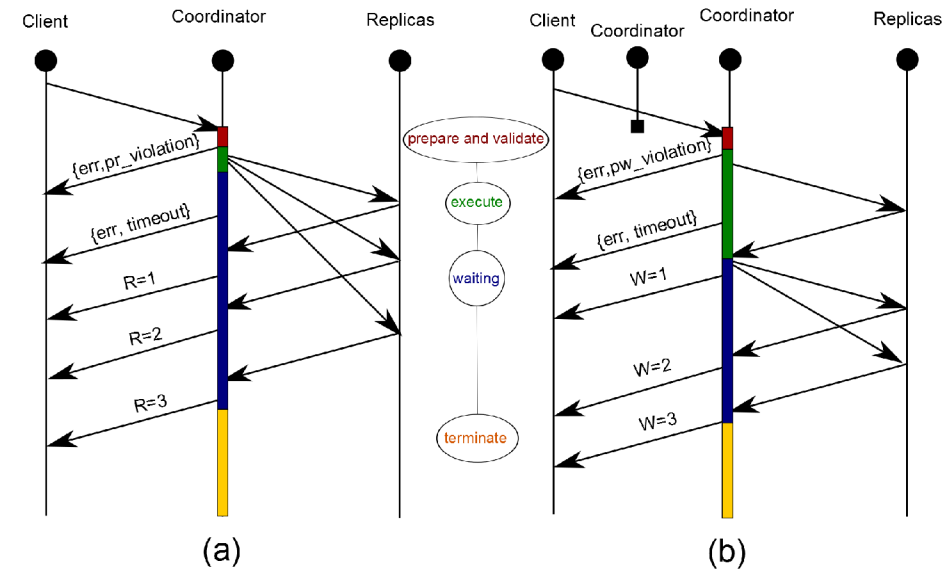


Figure 2.6: **Request Coordination:** a) represents the read request. b) represents write request. Replication factor = 3.

to a specified seed node. The seed node creates a coordinator process that enters the prepare state. The coordinator execution depends on the type of the processed request. Differences are captured in Figure 2.6.

- **Prepare State:** The preference list for requests is estimated. If the write request is processed, the coordinator checks if some of the partitions maintained by its owner node match the partition in the preference list. If the partition matches, coordinator enters validate state. Otherwise, new coordinator is created on the randomly chosen node corresponding to the owner of partition in the preference list and current coordinator terminates. The situation may be observed in Figure 2.6. The read request coordinator always enters validate state.

- **Validate State:** Client request properties are validated. PW and PR represent the number of primary nodes required to reply to the coordinator write and read requests respectively. If the

preference list contains less than PW(PR) primary nodes {error, pw(pr)_violation} message is immediately returned to the client. Moreover, the request configuration is checked for consistent values of its parameters. If the request configuration is valid, the process enters execute state.

- **Execute State:** The execute state models a communication between the coordinator and replicas. The execute state operations of read and write requests differ. Before the data is stored on a replica, version of the write request must be decided. The version of the request is determined by the coordinator. The coordinator stores the data to the virtual node of its owner corresponding to a partition in the preference list. The prepare state ensures that such virtual node exists. The virtual node decides the data version and replies to the coordinator. If the local request timeouts, the {error,timeout} message is replied to the client. Otherwise, the request is sent to other replicas in the preference list and the coordinator enters the waiting state. In contrast, the read request does not modify the data, hence it does not create new data version. The read request is sent to all replicas simultaneously and the coordinator enters the waiting state.

- **Waiting State:** the coordinator waits for replies from the rest of replicas. Parameters W and R represent the number of replicas required to reply to the coordinator for write and read requests respectively. If a reply is received from a replica, the number of already received responses is matched against R and W parameters. If the number of responses is equal to the parameter value, a reply to the client is generated. Otherwise, the coordinator stays in the waiting state. If the request timeout is encountered, {error,timeout} message is returned to the client. Note that if the {error, timeout} message had been generated to the client, some replicas could have stored the value despite the fact that the acknowledgement message was lost. Correspondingly, future requests could read inconsistent values.

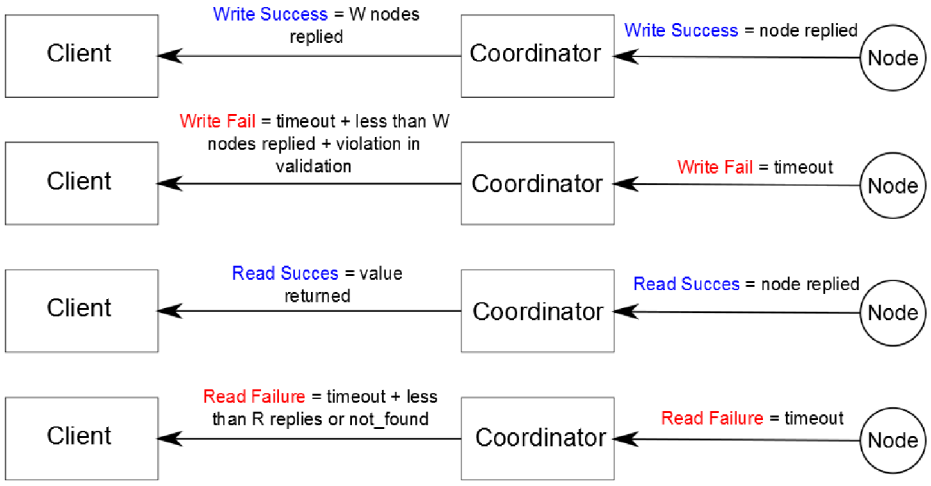


Figure 2.7: Availability of Requests.

- **Terminate State** Eventually, coordinators enter a terminate state. The terminate state is encounter either if timeout occurs or a coordinator received enough responses.

The availability of requests also depends on the type of the request processed. Figure 2.7 shows the availability of requests in all situations that can be observed by a client. The coordinator evaluates successful write if a node replies to the request. The write is failed when timeout was reached and a reply for request was not observed. The client observes successful write if coordinator received at least W responses from replicas. Otherwise the write is failed. Moreover, the write request is failed if coordinator observed a violation during the validation state.

Similarly, the coordinator evaluates successful read request if a node replies to the request. The request is failed if the node did not respond before the timeout. The client observes successful request if coordinator received at least R messages and a data was returned. If the *no_value* response is obtained, request is evaluated as failed.

2.4 Consistency, Availability and Partition Tolerance

Availability, consistency and partition-tolerance are key properties of distributed NoSQL datastores. Riak is a system built on CAP theorem [16]. The CAP theorem defines relationships between the properties. It states that in the presence of partitions, the choice between availability and consistency of data must be made.
We present informal definition for each property:

- The system is **available** if every request received by a non-failing node results in a response [16].
- The system is **consistent** if each read request sent after a write request completes, returns the value of the write request or a newer version [16].
- The system is **fault-tolerant** if it works in a presence of failures of some components.
- **Network partition** is a temporal state of the system where messages between two network components are lost. A single message loss can simulate an instantaneous network partition [16].
- The system is **partition-tolerant** if it works in a presence of partitions.

The proof of the theorem may be summarised in a single example. Assume a two partitioned components of a distributed system. A write request is processed in the first partition and terminates. A second partition does not receive the write request due to unspecified network error. The read request is performed in the second partition after the write request was acknowledged in the first. The request returns the stale value if no other writes were processed after the write request in the first partition. As a result, a system that prefers availability, returns stale value to the client. On the other hand, a system that prefers the consistency, blocks the requests if it is partitioned. Only a single partition may be available in order to preserve data consistency.

Riak default configuration is to prefer availability to consistency in the event of partitions [30]. A data availability is enhanced through

the replication process. Replica nodes are reason for inconsistent data states. Riak provides many mechanisms that increase the consistency among replicas such as hinted-handoff, read-repair and anti-entropy. The trade-off between the consistency and the availability is a tunable by the request configuration. However, in the presence of inconsistent responses, the system has to provide a conflict resolution. The conflict resolution is a mechanism that computes a correct value from a set of conflicting data values. Riak conflict resolution mechanisms include vector clocks and last write wins. We describe all mechanisms in a greater detail.

2.4.1 Fault Tolerance

A failure is an inevitable feature of the distributed environment. System developers wish to hide the failures from service users. Fault-tolerance is an ability of the system to fully operate in a presence of failed nodes or network partitions. In Riak, it consists of partition detection and reaction mechanisms [30].

A fault-detection is a process ensuring the partition detection and results in the state reconfiguration. As a result, faulty-nodes are ignored in request processing. Riak exploits the concepts of distributed Erlang [12, 3] in order to detect partitions. In distributed Erlang, processes run on Erlang nodes. A node represents a single Erlang virtual machine with own address space and a set of processes. Multiple Erlang nodes may be created on a single physical server. Nodes are registered using the EPMD name server application [13], running on each physical node. As a result, any Erlang node can communicate with any other Erlang node in the network.

When a node is created, it may contact other nodes in the network. The access to other nodes is provided by a cookie password. The password is held in the node state information. Only nodes with the same cookie are allowed to communicate.

The communication is started between an initiator node and a contacted node. The contacted node is either a singleton or a part of a group. The connection to the singleton creates bidirectional monitoring links between both nodes. The connection to the node in the group creates bidirectional monitoring links between initiator node

and each node in the group. The result is the cluster where each node monitors the rest of the nodes. If a node crashes, all nodes in the cluster are informed but not affected. Each node receives the crash report with the state information and removes the crashed node from its links.

There are two types of failures that can occur in the Riak. The crash failure, referenced as closed connection failure, is triggered at any event causing disruption of the connection. It is not important if the node crashed, the Riak application stopped or cable was manually damaged, the error is reported on all active nodes in the cluster within a few milliseconds [11].

The other type of failure is a network partition. The partition detection is based on Erlang monitoring mechanism. The mechanism is based on periodic ping messages exchanged between Erlang nodes. The parameter *net_tick_time* is configured on each Erlang node. Every quarter of the *net_tick_time* interval a ping message is sent to all monitored nodes. If the response is not received to four consecutive messages, the node is proclaimed crashed and the link is destroyed. A default value of the parameter is 60 seconds. The crashed node is discovered in the interval <45,75> [12].

When partition is detected, appropriate reaction must be provided by the system. When a failure of a node is detected, Riak temporarily removes the node from the preference list. Each failed node partition is handled by consecutive virtual nodes in the ring. As a result, Riak is able to process write requests when at least one node is alive. Riak guarantees availability of read requests when at most N-1 consequent nodes have failed. N is the replication factor.

2.4.2 Eliminating Consistency Violations

In distributed systems, a consistency model is defined by a set of specific rules. If the execution of requests follow these rules, it is guaranteed that database state is consistent within specified model. There are several consistency models used widely in distributed systems. Riak is adjusted to the eventual consistency. The model favours high data availability but Riak supports the strong consistency as well.

There are two approaches to consistency evaluation: a consistency

in partitioned networks and a consistency when a cluster is fully-operational. In both approaches, the strong consistency is implemented using a *quorum* on replicas. The quorum specifies a minimum number of replicas necessary to deliver data to the coordinator. Assuming the replication factor N, [R+W]>N quorum implies that R read acknowledgements and W write acknowledgements must to be returned to the coordinator. The sum of R and W must be greater then replication factor in order to implement the strong consistency. As a result, for any two different types of requests, executed on the same data, there is always at least one node that receives them both. Therefore, the coordinator of a read request has information on complete history of requests executed on the data before it responds to the client.

It is obvious that reordering of write requests on different replicas is still possible. If W = 1 then two write requests can be ordered differently on two distinct nodes. In [6], it is stated that Dynamo-style systems with write quorum W> [N/2] ensure that a majority of replicas will receive a write in the presence of multiple concurrent write requests. However, in Riak, the quorum does not prevent the reordering of concurrent write requests. The reordering is caused by the Riak request storage process. The first phase of each write request is the local store operation on the coordinator. There is no synchronization with other replicas, hence operation can be done concurrently on all nodes in the cluster irrespective of the W value. If write request are made on the same data in parallel, requests are reordered on two replicas.

In the partitioned system, processing two write requests in distinct partitions on same data results in different values stored on replicas. Subsequent read requests return inconsistent data. To achieve consistent executions, stronger consistency quorum [PW+PR] >N is exploited. PR is a parameter defining the number of primary replicas that must acknowledge the coordinator. PW has the identical meaning for write requests. The quorum allows execution of both types of requests only in one partition. If PW = 1 and PR = 3, data are written to any partition that has at least one primary node. The read requests are executed only in partitions with all available primary nodes for data in the request.

Hinted-handoff is a mechanism that increases consistency among

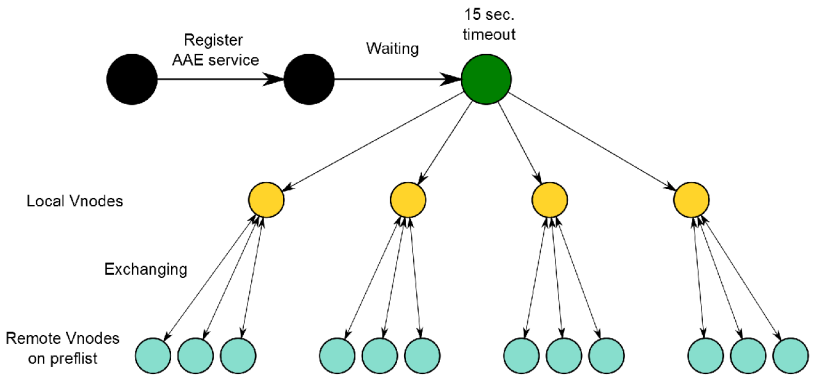


Figure 2.8: Anti-Entropy.

replicas when partition is healed. The partition recovery is detected and secondary nodes temporarily processing requests over primary partition data forward the updates to the corresponding virtual nodes. The primary node resolves conflicts and merges data into consistent state.

Read repair is a mechanism that passively increases data consistency. It is implemented as a state of read request coordinator process. The state of is called *read repair* and follows the waiting state of the read request. After the reply to the client, the coordinator enters read repair state and waits for additional replies from replicas. If enough requests are received or timeout is encountered, data value is generated from all replica responses. Inconsistencies are merged to a single value exploiting the conflict resolution. The consistent value is than forwarded to all replicas with different values. Read-repair increases consistency among replicas and availability of read requests in partitioned networks.

Anti-entropy (AAE) is a periodically executed mechanism that actively increases consistency between primary replicas based on Merle trees [24]. In Figure 2.8, the process of AAE manager is described. When Riak node is started, AAE service process manager is created. Every 15 seconds the manager sends tick messages to the local virtual nodes. Each virtual node checks the ownership of the partition. If the virtual node is secondary owner of the partition the tick message is ignored. Otherwise, the virtual node begins AAE process. Each node

has three types of data according to the data hash(<key, bucket>) pair: data directly mapped to its partition, data directly mapped to its predecessor and data directly mapped to its pre-predecessor. The number of data types is dependent on the replication factor. The node compares and exchanges Merkle tree created for data set with the predecessor and pre-predecessor. If the conflict is detected, conflict resolution is invoked.

2.4.3 Conflict Resolution

A conflict resolution is a process of merging two distinct data values into single consistent one. The resulting value is determined from data versions. The version is the timestamp created by the the clock mechanism when the value was written to the data.

In distributed systems, event ordering must satisfy a *clock condition*: for any two events e_i, e_j, where $e_i \rightarrow e_j \Rightarrow RC(e_i) < RC(_j)$ [25]. RC(e) is the value of the global clock when event e is executed and the relation $\rightarrow$ represents the relationship between events that is called causal precedence relation.

The causal precedence relation is an irreflexive, asymmetric, transitive binary relation [25]. It is based on the idea that two events are related if the first event could affect the outcome of the second event. In asynchronous systems, two events affect each other if they belong to the same process. If events belong to different processes and message exchange between the processes is the part of the event, then the process sending the message may affect the process that receives the message. The formal definition of the precedence relation $\rightarrow$ for events e_i, e_j, e_k and process P is:

- $e_i, e_j \epsilon P \wedge i < j \Rightarrow e_i \rightarrow e_j$,
- $e_i = send(mess_a), e_j = receive(mess_a) \Rightarrow e_i \rightarrow e_j$,
- Transitivity: $e_i \rightarrow e_j \wedge e_j \rightarrow e_k \Rightarrow e_i \rightarrow e_k$.

Two events e_i and e_j where neither $e_i \rightarrow e_j$ or $e_j \rightarrow e_i$ are concurrent: $e_i \parallel e_j$.

Causal precedence relation induces a clock condition. If two events

are causally related $e_i \rightarrow e_j$ then global clock of cause event e_i is lower then global clock of effect event e_j ($RC(e_i) < RC(e_j)$).

A distributed real-time clocks cannot implement a global clock that decides the correct event ordering due to the time synchronization problems [25]. Therefore, the concept of logical clock was introduced [21]. Logical clocks are an example of a global clocks in distributed systems that satisfy clock condition.

Each process has a logical clock value, denoted LC. Initial value of a logical clock is zero. Every time an internal or send event occurs on the process, logical clock is increased. The message of the sent event carries the logical clock of the sending process. Every time a process receives a message it modifies its logical clock to the maximum from the local and received clock value. Consequently, if $e_j \rightarrow e_k \Rightarrow LC(e_i) < LC(e_j)$.

The problem of logical clocks is that if $LC(e_i) < LC(e_j)$ then event may or may not be related. Vector clocks are extension of logical clocks that determine the causual precedence relation between any two events. Each process maintains a vector of logical clock values for each process in the distributed system. The initial value is the zero vector. If internal or send event happens on the process, vector value with the local process index is increased. The message of the send event carries vector clock of the sending process. If receive event is executed on the process vector clocks of sending and receiving processes are merged. The merged value is the vector that contains the the maximum value of received and local clocks for each index in the vector. Additionally, local index value is increased. It follows, if $e_j \rightarrow e_k \Leftrightarrow VC(e_i) < VC(e_j$. Therefore, for any two events in the system, the vector clocks decide whether they are related or concurrent.

Riak uses vector clocks to decide the consistent value of a data. If $VC(data_i) < VC(data_j)$, the consistent value is the value of $data_j$. If events are concurrent, siblings are created. Siblings are usually created if two write requests execute concurrently or in different partitions [32]. If the conflict resolution decision is left to a client, the set of siblings for the data is returned. Otherwise, Riak returns latest value from the siblings set, based on physical clock in data meta informatio.

3 Metrics

Consistency, availability and latency are the most important measured features of key-value stores. In this section, we analyse them individually providing the formal definitions. Furthermore, we derive metrics used to evaluate the behaviour of the Riak in Section 5.

Consistency trade-offs are hot topic among researchers. The proof of the CAP theorem established the trade-off between consistency and availability in a presence of partitions. The Brewer's conjecture proves the impossibility of both available and consistent systems in partitioned networks [16]. A system implementation must choose only one guarantee. Therefore, in systems preferring the availability, the model of eventual consistency was introduced. The system is eventually consistent if it converges to a consistent state in the absence of failures [7, 5, 34]. It is rather a weak consistency model.

Nevertheless, the CAP view of trade-offs in distributed systems is somewhat limited. A PACELC model was introduced to add latency and consistency trade-off to distributed database analysis [1]. PACELC stands for partition (P) leads to choice between availability (A) and consistency (C) else (E) in the absence of partitions the choice is made between latency (L) and consistency (C). Default version of the Riak database is PA/EL [1].

The trade-off between latency and consistency is caused by replication mechanisms used in distributed systems [1]. A future possibility of a failure forces systems to replicate their data. The consistency guarantees depend on the replica agreement or synchronization. The additional communication and processing increases the request latency. If the synchronization is omitted, requests are served faster but the risk of inconsistent responses increases.

In this section, we analyse each property in greater detail. We provide the terminology used in the section.

A system execution produces a *history*. The history H is a final sequence of operation invocations and responses that represent a system execution on single data. An operation $op_i = [op_i(start), op_i(end)]$, or the request, is a time interval on the history H, where i is a unique request identifier. The $op_i[start]$ is the sending time of the request op_i executed by a *process P*. The $op_i[end]$ is the time of the request op_i re-

sponse obtained by the process. Depending on the type of analysis, the process P may be a client or the database replica.

We distinguish two types of requests. The *write request* is denoted $w_i[x]$=[$w_i(start)$,$w_i(end)$] where x is a value written by a client. The *read request* is denoted $r_j[y]$=[$r_j(start)$,$r_j(end)$] where y is a value returned to the client. A *dictating write* of the read request $r_j[y]$ is the write $w_i[x]$ where $x = y$. In our analysis, we design experiments where for every two write requests w_i[a], w_j[b] holds that $a \neq b$. As a result, each $r_j[y]$ has a singular dictating write w_i denoted $dw(r_j[y]) = w_i[y]$, however, a write can dictate multiple reads.

3.1 Consistency

Consistency model is defined by safety and liveness guarantees [5]. The *safety property* defines the ordering of requests. It restricts the past behaviour of requests in the system execution. "It guarantees that nothing bad happens" [34]. The *liveness property* defines future behaviour of requests. It restricts the time when read requests start to reflect the current state of the datastore. "It guarantees that something good will eventually happen" [34]. An inconsistent state is the state of the system that violates liveness or safety properties of the analysed consistency model.

Consistency models can be analysed from different perspectives. A *data-centric perspective* is a view of a system internal state [7, 18]. The history of execution is analysed on database replica processes. The system is in consistent state if safety properties defined by consistency model are not violated and all replicas store identical data. The defined consistent state is hard to obtain in distributed systems due to a communication delay and clock synchronization.

The data-centric perspective is desirable for system developers. Usually, they are interested in behaviour of the internal mechanisms like Hinted Handoff, Read-repair or Anti-entropy. Their goal is to analyse convergence to consistent state on all replicas in respect to exploited supporting mechanisms.

A *client-centric* perspective is related to the process, usually a client, interacting with a database system [7]. The system is analysed from

external view. The state on replicas is insignificant but request replies observed on each client are evaluated. The history of execution is analysed on client processes. A request is consistent if the safety and liveness properties defined by the consistency model are not violated.

The client-centric analysis is beneficial to the application developers. It investigates the effects of synchronization and agreement protocols seen by client. Based on the analysis, the application is modified to satisfy the needs of its end-users. In the thesis, we focus on the client-centric consistency perspectives.

Linerizability [19] is the strongest consistency model in distributed systems. The history is linearizable if requests can be ordered in respect to their real time execution and each request is processed in a single instant. More formally, a history H is linearizable iff:

- H is equivalent to some sequential history S. The history H is equivalent to S if all operations executed by every process in S respect the ordering of operations executed by the corresponding process in H. The history is sequential if each operation is atomic: $op_i[start]$ is immediately followed by $op_i[end]$. We write the history S as a sequence of atomic operations $r_i[x]$ and $w_j[y]$.
- The ordering of operations in S respects a real-time ordering of requests in H. If $op_i[end] < op_j[start]$ in H then $op_i < op_j$ in S.

A system implementing the strongest consistency model suffers from increased latency and decreased availability in partitioned systems. Moreover, current database end-users demand always available, fast accesses. Therefore, the current trend is to sacrifice the consistency in the benefit of availability and latency. There are many weaker consistency models explored by the research community. They can be divided into data-related and client-related models[1].

- **Data-related consistency models:**
  - **Linearizability** is the strongest consistency model

1. In [7] models are referenced as client-centric and data-centric, similar name as perspectives to consistency. However, both models types may be verified from data-centric and client-centric perspective thus the term could be confusing.

  - In **Sequential consistency**, requests on replica must be ordered the same as received from the client [22]. It differs from linearizability in respect to the liveness property. In the linearizable systems, requests cannot return stale values, however, sequential consistency has no bounds on staleness of the request.
  - **Causal consistency** exploits the concept of causal precedence explained in Section 2.4.3 [23]. Two requests that are causally related must be ordered equally on all replicas. Similarly, there are no bounds on the request staleness. The causal consistency is the strongest consistency model that can be implemented in the event of partitions [23].
  - **Eventual consistency** provides no safety guarantees on request ordering. The only requirement is the convergence of replica to a consistent state.
  - **Weak consistency** model has no guarantees. It states that replicas might by chance become consistent in the future [7, 34].

- **Client-related consistency models:**
  - In **Read Your Writes Consistency**, the request is inconsistent if the read observed a value that was older than the version of data previously written by the client. The list of other models can be found in [7, 34]. The focus of the thesis is on data-related consistency models.

Riak is an eventual distributed key-value store. The only guarantee that eventual datastores provide is the convergence of replica to consistent state. Therefore, the goal is to verify how much eventual the system is. To verify the convergence of replica states, metrics used to capture the idea of the convergence must be presented.

There are several metrics used to evaluate datastores from client-centric perspective. The system is *k-atomic* if a read operation returns the version of one of the k latests writes [2]. The system is Δ-*atomic* if

a read operation returns value that was considered consistent maximum Δ time units before [17]. Moreover, probabilistic metric may be used to evaluate consistency of the system [6].

We analyse the consistency of the Riak database using the Γ-metric defined in [18]. Moreover, we propose new ζ metric representing number of violated requests in history. The Γ-metric is based on Δ-consistency which captures the deviation of the system execution from the linearizability [17].

We denote by *conflicting request*, the request that is not linearizable in history H. The algorithm to compute all conflicting requests is based on algorithms from [15, 18]. We use a similar notation.

- Requests are divided into clusters labelled C(k,v), where k represents a key[2] and v represents a value. The cluster contains a write request w_i and set of read requests r_j where $dw(r_j) = w_i$ for all j. Clearly, each cluster consists of requests made on the common key and value. A cluster contains exactly one write request and arbitrary number of read requests. The problem of identifying conflicting requests when write requests share a value is NP-complete [15]. We ensure that a cluster has a single write request by assigning each write request unique data value.

- A zone Z(k,v) is computed for each cluster. The zone Z(k,v) is an time interval [Zones$_{start}$, Zone$_{end}$] corresponding to the cluster C(k,v) requests times. We define a zone minimum Z_{min}(k,v) and a zone maximum Z_{max}(k,v).

  - $Z_{min}(k,v) = op_i[end] : \forall\, j \neq i,\ op_i[end] < op_j[end] \wedge op_i, op_j \in C(k,v)$.
  - $Z_{max}(k,v) = op_i[start] : \forall\, j \neq i,\ op_i[start] > op_j[start] \wedge op_i, op_j \in C(k,v)$.

- Each zone is assigned a type.

  - If the $Z_{min}(k,v) < Z_{max}(k,v)$, the zone Z(k,v) is a forward zone. The first request end was executed before the last re-

2. In Riak, the referenced key is the <bucket, key> pair.

quest started. The interval [$Zone_{start}$, $Zone_{end}$] = [Z_{min}(k,v), Z_{max}(k,v)].

- If the Z_{min}(k,v) $\geq$ Z_{max}(k,v) the zone Z(k,v) is a backward zone. The backward zone represents a cluster where the last request started before the first end of the requests in the cluster. The interval [$Zone_{start}$, $Zone_{end}$] = [Z_{max}(k,v), Z_{min}(k,v)]. The intervals of all requests in the backward zone overlap on the common subinterval. As a result, all requests could be scheduled sequentially at the subinterval.

- Figure 3.1 depicts requests of a partial history with corresponding zones.

- To identify conflicts, pairs of clusters can be evaluated individually [19]. Two zones corresponding to the clusters have a potential conflict, if their intervals intersect. There are three combinations of overlapping zones:

  - **[forward,forward]:** The equivalent sequential history S of the forward zone in history H must contain the whole interval of the zone. Therefore, if two forward zones intervals intersect, the requests cannot be scheduled on the history S with respect to the timing constrains of the linearizability [15].
  - **[backward,forward]:** All requests in the backward zone can be sequentially scheduled in any part of the zone interval. The equivalent sequential history S of the backward zone in history H must contain some part of the interval of the zone. Therefore, if backward zone lies within a forward zone interval, there is a conflict. Otherwise, operations of the backward zone can be scheduled before or after the forward zone interval in any subinterval that does not intersect the forward zone.
  - **[backward,backward]:** The zones do not conflict because operations of both clusters can be scheduled at any point of the corresponding backward interval. Operations of clus-

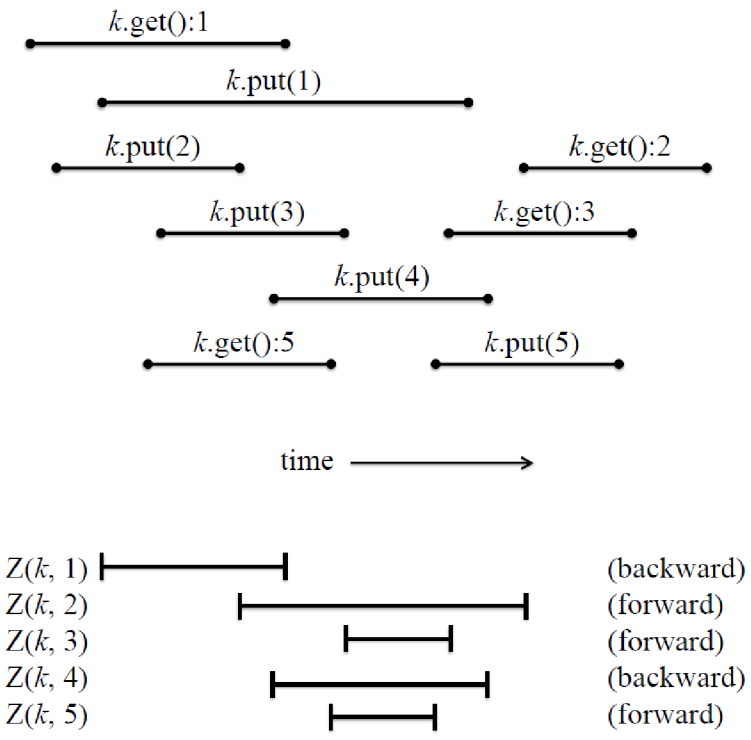


Figure 3.1: **Requests and their corresponding zones.** Taken from [18]

ters are scheduled on different disjointed subintervals, thus the history is linearizable.

The algorithm to compute the metric Γ is taken from [18]. We modified the algorithm so the Γ is computed only for requests zones that are conflicting. Additionally, we compute the metric Γ′ that is used in definition of ζ metric.

- Z(k,v) and Z(k,v′) are distinct conflicting zones with specified max and min values: Z_{min}(k,v), Z_{max}(k,v) and Z_{min}(k,v′), Z_{max}(k,v′).

- if $(Z(k,v)_{max} - Z(k,v')_{min}) < (Z(k,v')_{max} - Z(k,v)_{min}) \Rightarrow$
  $\Gamma(k,v,v') = Z(k,v)_{max} - Z(k,v')_{min}$
  $\Gamma'(k,v,v') = \{Z(k,v)_{max}, Z(k,v')_{min}\}$

- if (Z(k,v′)$_{max}$ - Z(k,v)$_{min}$) < (Z(k,v)$_{max}$ - Z(k,v′)$_{min}$) ⇒
  Γ(k,v,v′) = Z(k,v′)$_{max}$ - Z(k,v)$_{min}$
  Γ′(k,v,v′) = {Z(k,v′)$_{max}$, Z(k,v′)$_{min}$)}
- If the Z(k,v) or Z(k,v′) is contracted by the Γ(k,v,v′) interval the conflict is eliminated [18].

We define the metric ζ that represents the number of violated requests. It is based on computation of the metric Γ′.

- $\zeta(k, v, v')$ represents the number of conflicting operation of two conflicting clusters C(k,v), C(k,v′).
  - ζ(k,v) is the set of conflicting operations of the zone Z(k,v).
  - if Z(k,v)$_{max}$ ϵ Γ′(k,v,v′) ⇒ $\boldsymbol{\zeta(k, v)}$ = {op_i|
    $op_i(start)$ ϵ [Z(k,v′)$_{min}$, Z(k,v)$_{max}$] ∧ op$_i$ ϵ C(k,v)}.
  - Z(k,v)$_{min}$ ϵ Γ′(k,v,v′) ⇒ $\boldsymbol{\zeta(k, v)} = \{op_i|$
    $op_i(end)$ ϵ [Z(k,v)$_{min}$, Z(k,v′)$_{max}$] ∧ op$_i$ ϵ C(k,v)}.
  - Similarly, ζ(k,v′) is computed.
  - $\boldsymbol{\zeta(k, v, v')} = min(|\zeta(k, v)|, |\zeta(k, v')|)$
- ζ(k) is the number of conflicting requests on history K which represents the execution of requests on key k. We present the algorithm to compute the ζ(k):
  - Compute $\zeta(k, v, v')$, the set of conflicting operations, for all zone conflicts Z(k,v) and Z(k,v′) on key k.
  - ζ'(k)= $\Sigma_{Z(k,v),Z(k,v')\epsilon Conflicts}$ $\zeta(k, v, v')$
  - ζ''(k) = ζ'(k) without duplicate requests.
  - ζ(k) = $|\zeta''(k)|$.

In summary we present all metrics used to evaluate Riak distributed system in Table 3.1.

$\Gamma(k, v, v')$	Gamma metric
ζ(k)	The number of violated key-requests / total number of key-requests * 100
ζ(S)	The sum of ζ(k) for all keys / total number of requests * 100
ζ(S,write)	ζ(S) where only write operations are considered in ζ evaluation
ζ(S,read)	ζ(S) where only read operations are considered in ζ evaluation

Table 3.1: Consistency Metrics

3.2 Latency

In our analysis, we define the request latency as the time between request the submission and its acknowledgement on the client process. In other words, it is the time needed for the request processing. In Riak, there are two main causes of increased request latencies: a delay in request processing on individual replicas and amount of node-to-node communication during a lifetime of the request.

The latency of a request depends on a duration of the network communication and the duration of the communication depends on the network properties. Naturally, communication between geographically distributed Riak nodes is more demanding in contrast to the communication in a local database cluster. Our analysis of the Riak is done on the local network cluster.

The type of a request influences the amount of node-to-node communication. In Riak, distinct types of requests are processed in different manner. A write request is processed by two coordinators and N replicas at most, where N is the replication factor. However, one coordinator is the same physical node as the data replica. Thus, at most, N + 1 different physical nodes communicate during the write request. Similarly, a read request is processed by one coordinator and N replicas. The coordinator and replica may differ, hence, at most N + 1 nodes are contacted during the read request lifetime.

Moreover, each request, write or read, has specified number of replica acknowledgements A explained in Section 2.3. The coordi-

nator communicates with N replicas concurrently but receives acknowledgements only from specified amount of replicas. The latency of the requests directly depends on the amount of required acknowledgements. The higher A imposes higher latency of a request. If N - A + 1 replicas are unavailable the maximum delay of the request may be at most up to the defined request timeout. A default value of the Riak request timeout is one minute [30].

Nevertheless, read and write requests differ in terms of the storage processing. The write request requires store operation to the underling low-level storage system while read request requires retrieve operation on the storage. Hence, the delay of the request processing depends on the used type of the low-level storage. Riak supports several types of low-level storages: Bitcask [27] and LevelDB based on Google Bigtable [8].

The Bitcask is the default storage system of the Riak. Bitcask is the Erlang application providing key/value data storage and retrieval based on hash tables. It is designed to provide low latency for both reading and storing data. The retrieving operation is handled by the in-memory hash table that directly points to the disk locations. The advantage of the Bitcask storage is its key-value designed implementation. In contrast to Bitcask, LevelDB may impose slower read accesses and more discs seeks per single write access [30].

Metrics used to analyse the latency are provided in Table 3.2. The latency of each request in the history is computed. Moreover, minimum, maximum and percentile values are computed on all requests in history.

Name	**Description**	**Granularity**
Maximum latency	Request with maximum duration	Per history
Minimum latency	Request with minimum duration	Per history
75-latency	75-percentile of requests	Per history
25-latency	25-percentile of requests	Per history

Table 3.2: Latency Metrics

3.3 Availability

In synchronous distributed systems, an each request received by a non-failing node must result in a response [16].

In Riak, each request has specified maximum time allocated for the execution. The default time of the Riak request is one minute. If the coordinator does not receive enough replica responses by the specified time, timeout error is generated to the client. The timeout may be caused due to system partitions, increased network latency or lost request in the communication.

In Riak, we define a request availability for each type of the request separately.

A write request is available if a client received an acknowledgement from the contacted coordinator. In contrast, the request that resulted in the timeout error is unavailable. Note that if the request was considered unavailable by a client, the data could be stored on some replica. For example, if W = 3 and a single replica did not respond and timeout error was generated, data were written to the remaining set of replicas. As a result, future read request may return inconsistent data.

A read request is available if a client receives some data in the request response. The content of the data is inessential as long as any value is returned. The content is evaluated through consistency metrics. A request is unavailable if the timeout error message or not-found value were returned. The Figure 2.7 represents the unavailable responses and reasons for their return. The not-found value indicates either the fact that value was never written to the system or the inability of the partition system to retrieve data from the partition [30, 32]. We are not concerned with the first situation as it is prevented in our data analysis implementation where an initial value is store to each data in the beginning of the experiment. Therefore, not found value indicates an unavailable request.

The not found value is returned when replica replies processed by the coordinator contain only not found values. Data are present on the system, but network is partitioned. Not-found value indicates either that data are not present in the partition or data are present on some replicas but R<N and a replica containing the data did not respond to the coordinator.

In Table 3.3 metrics to evaluate the availability of the system are presented. The availability is measured for each type of the request separately. Overall availability is measured for all requests in the history.

Name	**Description**	**Granularity**
Request-availability	Number of available requests /total number of requests * 100%	Per history
Read-availability	Number of available read requests /total number of read requests * 100%	Per history
Write-availability	Number of available write requests /total number of write requests * 100%	Per history

Table 3.3: Availability Metrics

4 Implementation

A program used in our evaluation of Riak datastore is based on *Cdbvalidator application*, developed by Y-Soft engineers. The *Cdbvalidator* is a set of interfaces specifying behaviour of testing applications. Our contribution is the distributed application designed for the Riak evaluation. The application implements the behaviours of the *Cdbvalidator*. During the development, *Cdbvalidator* behaviours were modified in order to enhance flexibility of the experimental model. The architecture of the application is shown in Figure 4.1.

The application consists of two remote components, *Riak Validator* and *Riak Interceptor*, that communicate in distributed cluster exploiting the RabbitMQ middleware [28]. The RabbitMQ is an open source message broker implementing Advanced Message Queuing Protocol. It is written in Erlang programming language and provides transparent communication between applications built on different platforms. The details of the communication model are provided in Section 4.1.2.

Moreover, the communication is a message exchange between the two components. The *Riak Validator* forwards messages to the *Riak Interceptor*. The *Riak Interceptor* processes the messages and replies back to the *Riak Validator*. Subsequently, the *Validator* is able to process the results. The output of the result processing is analysed by the *Analyser*.

We present the architecture of the distributed application, shown in Figure 4.1.

- **Riak Validator** is an Java application running on a single physical node. It consists of several components.
  - **Planner** creates a plan of requests execution according to the experiment configuration.
  - **Test Worker** simulates a client application submitting requests to the database system.
  - **Validator** analyses the history of execution gathered from the responses obtained by the clients.

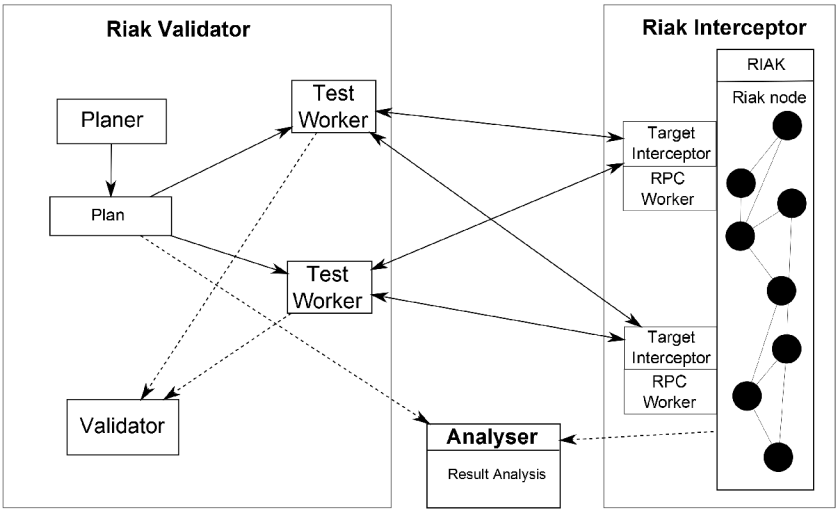


Figure 4.1: Components of the distributed application used for Riak analysis. Arrows indicate a communication between particular components and sub-components.

- **Riak Interceptor** is an Erlang application running on a physical node in the distributed cluster. The node is distinct from the server with the *Riak Validator*. Typically, several Riak Interceptors run in the cluster each corresponding to a single Riak node. The interceptor consists of several components.
  - **Target Interceptor** maintains channels used for the communication with the *Test Workers*.
  - **Rpc Worker** is responsible for communication between *Test Workers* and the Riak datastore.
  - **Riak Database** [30].
- **Analyser** is an independent Erlang application used to process logs obtained from the *Riak Validator* and Riak datastore application. The analyser implements the metrics provided in Section 3.

In this section, details of each application are provided.

4.1 Riak Validator

Riak Validator is a Java application implementing *Cdbvalidator* interface. The main component of the application is the *Test Manager*. It is a process handling a plan generation, assignment of the plan to the *Test Workers* and execution of the workers by the created thread pool. Additionally, the manager creates *Test Validators* that analyse the plan execution.

The *Test Manager* generates the plan of requests and distributes the plan to the *Test Workers*. In the original implementation of the test manager, test workers were initialized and an individual plan was generated for each worker. Each plan of the worker was independent of the other. However, the implementation did not allow the modelling of relationships between plans of the workers. Therefore, the modified test manager creates a single plan of requests executions. The plan is partitioned and distributed to all initialized workers using the round robin algorithm. A single plan offers more flexibility in a modification of global experiment properties, e.g. designing the global plan for network partitions thorough experiment.

4.1.1 Planner

A *Planner* is a module that creates a plan of the requests executions. A configuration of the experiment is provided on input and *Planner* computes the list of actions. The actions represent communication messages. The *Cdbvalidator* defines the structure of the messages. All messages are encapsulated in a single *Test Action* behaviour. The *Test Action* is defined by the type of the message, its validator and communication properties (timeout, maximum number repeats, etc.). *Test Action* can be default action of the validator or an application specific request. The *Time Action* is an equivalent of the *Test Action* in the *Planner* execution. The *Planner* generates a list of time actions that are translated to the corresponding test actions at the end of the plan generation.

The *Planner* works in several phases.

- **Generate timeline phase** defines the time instances of all ac-

tions executed in the experiment. The experiment configuration specifies the number of actions per a minute along with the total execution time of the experiment. The planner takes the parameters and assigns corresponding number of actions to time instances within each minute of the experiment. The assignment is done using a random uniform function.

- **Request valuation phase** assigns each *Time Action* (with already defined time of execution) a type, a key, a value and a node in the distributed cluster. A key-set of all keys used in the experiment is created in the beginning. The size of the key-set is taken from the configuration. First X requests in the timeline are converted to the write action, each with a distinct key, where X is the number of keys in the key-set. Therefore, each distinct data-object is assigned initial value. Consequently, we may assume that each read request should return some value. For other actions, keys are assigned from the key-set according to the random uniform function. The size of the key-set determines the rate of requests processing on identical data. For each request, nodes are assigned randomly from the set of specified nodes in the cluster. The value is also generated randomly and type is determined according to the probabilities of request types specified in the configuration.

- **Generate partitions phase** creates additional halt requests that represent the network partitioning. Partitions are generated according to the probabilities specified in the configuration parameters.A duration of the partition in the network is defined by the interval [Min_parition_time,Max_partition_time].

- **Generate clients phase** assigns requests in the timeline to the particular clients. The assignment is realized using round robin algorithm.

- **Generate wait actions phase** creates wait actions between all

subsequent requests for each client. The duration of the wait action is equal to the difference of times of the two requests. Without wait actions, client requests would be executed one after another. As a result, all client requests would be executed in the beginning of the experiment. The client would be idle in the rest of the time. Exploiting the wait actions, the corresponding rate of requests per a minute is modelled. The wait action is the default *Cdbvalidator* action. Moreover, it is the only action not used as a message in the communication.

Each planner phase execution depends on the output of the previous phase. Although, generate partitions, request valuation and generate timeline phases could be merged together into singular phase, the independent design provides more flexibility. The algorithm complexity trade-off is insignificant and the flexibility may be exploited if some alternation to model relations among requests were intended in the future.

4.1.2 Test Worker

The *Test Manager* receives a plan from the *Planner* and creates a *Test Worker* for each client in the plan. Subsequently, the plan is passed to the *Test Worker*. When all workers are initialized and mapped to their plans, *Test Manager* runs them in a thread pool.

Test Worker executes communication exchanges with remote servers with the Riak datastore. The communication is implemented by RabbitMQ (RBMQ) message broker [28]. The *Test Worker* creates an independent worker-client for each remote node in the database cluster. The worker-client stores network properties for the communication with the assigned remote node. The execution of a single worker is sequential. Consequently, actions of the worker-clients belonging to the common test worker are executed sequentially. Moreover, the initialization time of worker-clients may vary for distinct test workers. As a consequence, the individual plans are slightly shifted, however, the properties of the experiment are not affected by the time drift between workers.

In Figure 4.2, the worker-clients mapping to the remote server

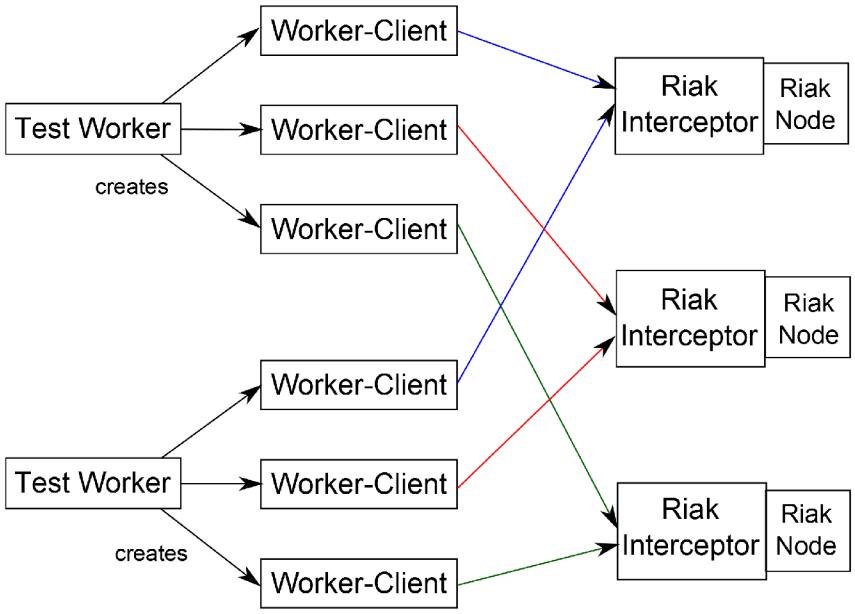


Figure 4.2: Workers Clients Architecture.

is depicted. In the example, there are two distinct *Test Workers* and three nodes in the database cluster. The worker creates three worker-clients, each for a single remote server. Consequently, one worker-client of each worker is linked to a single remote server.

Before, the plan is executed each worker-client establishes communication channels to the remote servers. The communication in RBMQ framework is done using message queues. A queue is an infinite buffer that represents a mailbox inside RBMQ. The queue belongs to the communication channel. Messages between applications can be stored only in the queue. A producer, is a process sending a message to the queue and a consumer waits for messages in the queue. If the queue is non-empty, the consumer processes the first message.

Each queue is maintained by the RabbitMQ server. The server is located on each node with the *Riak Interceptor* application. The RBMQ client, the *Riak Validator*, must have direct connection to the RBMQ

server. The connection is established when the *Test Worker* is initialized. All messages from producers and consumers are directed to the RBMQ server. The server maps messages from producer queues to the corresponding consumer queues.

In the application, there are two types of communication. A default communication is used to map each worker-client to the single *RPC Worker* of the corresponding remote node. The mapping is accomplished by means of a shared queue name exchanged in a registration message. The shared queue is used in communication between worker-client and the *RPC Worker*. The registration messages are sent to a default queue known to the application. On the other hand, application-specific communication models request exchange between clients and Riak datastore. Application-specific requests are communicated using the shared queues from the default communication.

The default communication is executed on the channel defined by the default queue called *TARGET_NODE_QEUEUE_NAME_ID*. There is a single default queue for each *Riak Interceptor* application. Each remote *Riak Interceptor* has its own unique ID. The value of ID determines the name of the default queue of the interceptor. The queue name is known to the *Riak Validator* and the *Riak Interceptor* as well. As a result, each worker-client can determine the default queue of its corresponding remote interceptor node.

A worker-client is uniquely defined by the <nodeID,planID> pair, where planID is the identification of the parent *Test Worker* and nodeID is the identification of the corresponding remote *Riak Interceptor* application. The worker-client determines a unique name of the shared queue (*TC_PLAN_QUEUE_NODEID_PLANID*) using its identification pair <node,planID>. The shared queue name is serialized into the *REGISTER_QUEUE* message sent to the default queue. The registration messages are communicated in many-to-one fashion. All worker-clients mapped to the common remote *Riak Interceptor* send the registration message to the same default queue. The *Riak Interceptor* application handles messages from the queue and assigns shared queesu to the corresponding *RPC Workers*. Subsequently, the application-specific communication uses a shared queue in the one-to-one fashion. As a result, each client-worker is mapped to the unique *RPC Worker* of some *Riak Interceptor*. Therefore, the execution of re-

quests from the distinct workers is done in parallel. In Figure 4.3, the difference in communication of registration messages and requests execution is shown.

After, the communication channels are established, *Test Worker*

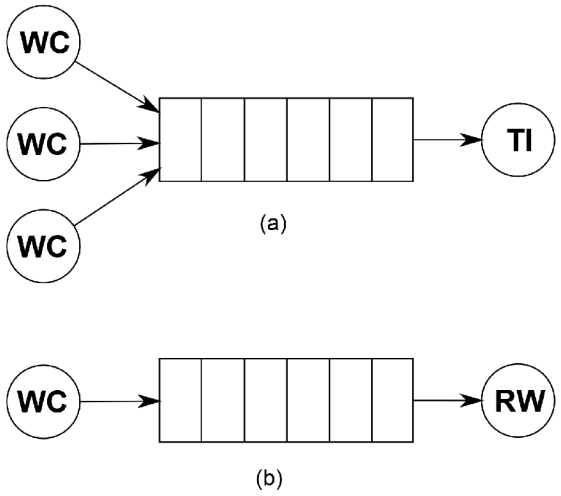


Figure 4.3: **RabbitMQ communication:** a) Communication between worker-clients and Target Interceptor. b) Communication between worker-clients and RPC Workers

sequentially executes actions from the plan. The worker-client executed in the iteration is defined by the action nodeID property. The nodeID is equal to the ID of the *RPC Worker* node mapped to the worker-client. The executed worker-client serializes the test action and produces serialized message to the shared queue.

In order to model partitions, each *RPC worker* in the cluster must know the Riak database communication ports of the other nodes. Therefore, in the beginning of the plan execution, each worker-client retrieves the Riak database port of the corresponding *RPC Worker* using a *GET_COMMUNICATION_PORT* message. The *Test Worker* gathers all ports obtained by its worker-clients. Consequently, the list of obtained ports is spread to all *RPC Workers* paired to the worker-clients.

There are two types of requests executed by a worker-client in the plan execution. The wait action is the only action that does not model

any communication with the remote server. It represents a time interval between two consecutive application-specific requests. Therefore, each request is followed by the wait action in the plan. The wait action signals to the *Test Worker* the amount of the time it should wait before executing the next action. The duration of the wait action is the time difference between the two request invocations. However, the first request has some duration of the execution. If the *Test Worker* waited for the original duration of the wait action the time of the experiment would be increased by the sum of all request execution durations.

Therefore, before the wait action is executed, the wait action duration is recomputed. New duration is the time difference between the previous request acknowledgement and the current request invocation. If the duration is negative, the request is dropped and next request in the plan is executed. Request is dropped because the deadline for its invocation passed. The reason is high overhead in previous request execution. If the request were executed, the time of the experiment would be increased and the minute rate of requests would be disrupted.

The second type of requests are application specific requests: read, write and delete. The RabbitMQ provides means for communication between applications built on distinct platforms. The *Riak Validator* is implemented in the Java programming language and the *Riak Interceptor* is implemented in Erlang programming language. Thus messages must be serialized to the form that both application understand. The serialization is implemented in the *Riak Validator* application. Each request is serialized to an Erlang tuple exploiting the Java OtpErlangObject class representing an arbitrary Erlang term. Serialized messages are sent to the corresponding shared message queues.

After all requests in the plan were executed, the *Test Worker* unreqisters shared communication queues using the default queue. The communication is similar to register messages in Figure 4.3. After all workers terminate, the *Validator* starts to execute.

4.1.3 Validator

The *Validator* analyses all request processed in the system execution. The analysis is application-specific and dependent on the require-

ments of the experiments. The validator is structured into several levels.

- **Request Validator** processes results from the messages received by the *Test Workers*. When the worker obtains response to the request, it forwards the result to the request validator. The validator stores the result. When the validation phase is executed, the request result is processed. During the processing, the properties of the request result are encapsulated into a single result for the validation. The usual properties of the result include the time of the execution, availability of the request and content received in the reply.

- **Test Plan Validator** processes requests from a single plan corresponding to the *Test Worker* execution. It receives results from the request validator for each requests in the plan. The validator produces a single result that represents the plan validation. A result contains the amount of requests that were successful, failed, timeout or dropped during the plan execution.

- **Test Validator** validates all plans executed by the *Test Workers*. It merges the results from the all plan validators into a single output. The outcome of the experiment is logged externally for the future analysis.

4.2 Riak Interceptor

Riak Interceptor is an Erlang application, implementing communication behaviour specified in the *Cdbvalidator*. It consists of two main modules, *Target Interceptor* and *RPC Server*, monitored by the structure of the supervisors shown in Figure 4.4. The main supervisor is started on each physical node in the Riak cluster and initializes the *Target Interceptor*. Subsequently, the *Target Interceptor* creates the worker supervisor that will monitor future *RPC Workers*. The *Riak*

Validator uses the default queue to contact the *Target Interceptor* and announce shared communication queues. The communication queues are sent to the supervisor that creates the *RPC Worker* for each shared queue. Each *RPC Worker* handles requests from a single *Test Worker*. Consequently, the RPC Worker propagates requests to the Riak database.

4.2.1 Target Interceptor

When the *Target Interceptor* is initialized, it creates AMQP connection to the local RBMQ server. A channel is specified for the connection and default queue is created for the communication with remote application. The name of the default queue is defined by the node ID and hence it is known to the application. Furthermore, the interceptor starts the worker supervisor and continues executing as the Erlang generic server.

During its lifetime, the *Target Interceptor* consumes the registration messages from the default queue. Each message in the queue contains a reply queue of the remote application and payload of the message. The payload is deserialized and the content is processed. The content represents the request from a client. The request is either register or unregister queue message. After the request processing, the acknowledgement must be sent to the application. Accordingly, the *Target Interceptor* produces an acknowledgement to the reply queue.

The register message represents a requirement of the client to create a channel for the application-specific requests. Requests from the different clients must be executed concurrently. Therefore, each client should be processed by a separate process. The process is called *RPC Worker*. It is created when the register message is processed. The *Target Interceptor* sends the register request to the worker supervisor. The supervisor spawns a new process with new communication channel and specified queue name for the communication. Moreover, the unregister message represents the requirement of the client to destroy its communication link in the end of the experiment. The *Target Interceptor* terminates the *RPC Worker* corresponding to the queue specified in the request.

Furthermore, the channel of each *RPC Worker* is handled by the same connection as channels of the *Target Interceptor* and all other *RPC Workers*. As a result, if the connection is dropped unexpectedly, communication channels of all *RPC Workers* must be reestablished by the *Target Interceptor*. The *RPC Workers* are reinitialized with the states equal to the crashed processes.

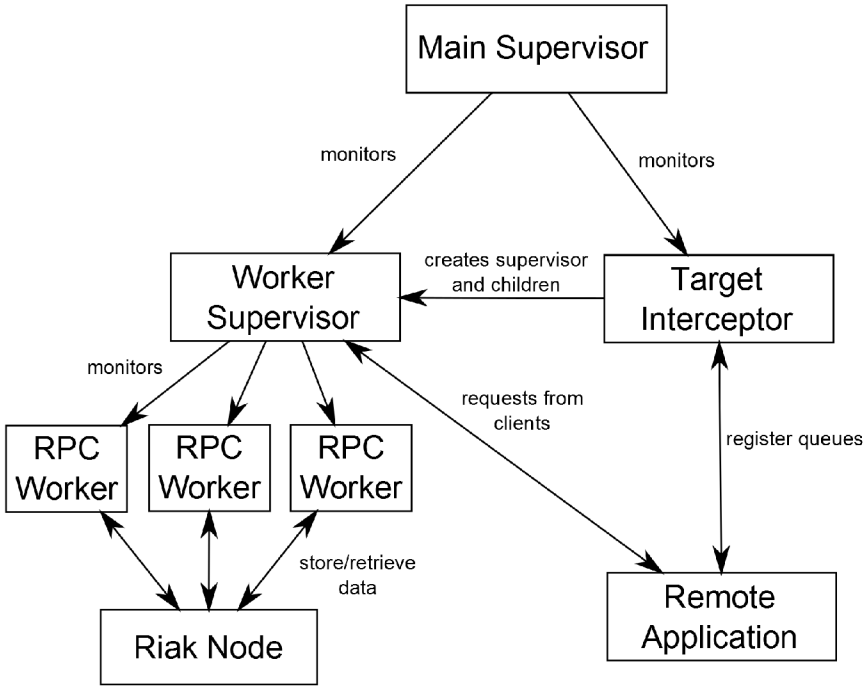


Figure 4.4: Riak Validator Architecture.

4.2.2 RPC Worker

The *RPC worker* is a process that handles application specific request from a single *Test Worker*. When the *RPC Worker* is initialized by the supervisor, it creates a channel with defined queue and behaves as generic server thereafter. It consumes messages from the shared

queue. Each *Test Worker* request is processed synchronously. As a result, at most one message is present on the queue in every moment of the execution. The message that arrives to the queue contains a return queue and a message payload. The content is deserialized from the payload and corresponding request is executed. Afterwards, the acknowledgement message is generated to the reply queue.

Requests too the Riak database are generated from the message content received from the *Test Worker* using the standard operations of the Riak_KV module [32]. Before and after each request, the time and content of the request or its outcome is logged to the file. Each request is properly handled and response is returned to the *Test Worker*.

The halt request is the special type of application request used to model partitions. Partitions are modelled reconfiguring the IP tables to ignore the requests received or forwarded to Riak databases on the other nodes in the cluster. To block the communication from the remote node, Riak application port must be blocked. Hence, in the beginning of the experiment, requests to exchange the remote ports are processed by each *RPC Worker*. Consequently, when the halt request arrives, the *RPC Worker* blocks all ports different from its own Riak database port. The acknowledgement of the halt request is produced immediately after its arrival. Henceforth, the ports are blocked and the process executing the halt request is put to sleep for defined amount of miliseconds. When the process awakes, ports are unblocked and the *RPC Worker* continues the execution. Due to the blocking time of the *RPC Worker*, special the *Test Worker* is created for the execution of the halt requests. As a result, the sleeping interval of the *RPC Worker* during the time of the partition does not affect the execution of the other requests.

4.2.3 Riak Database

We use the 1.4.8 version of Riak database in our experiments. Additionally, we implemented interceptor modules that are loaded by the *Target Interceptor* to the Riak database application. The interceptors are used to log the internal communication in the Riak database. The coordinator logs the acknowledgements of remote Riak replicas for each request. The replica datastore logs received from the coordi-

nator. Exploiting the logs from internal communication, the internal Riak behaviour may be analysed.

5 Experiments and Evaluations

In this section, we present the results of experiments realized by the testing application described in Section 4. We run the application in the Y-Soft local cluster. The cluster consists of several physical computers managed by the VMware vSphere cloud computing virtualization platform. The clock synchronization between the nodes in the cluster is managed by a single NTS server that communicates with each node in the cluster and periodically synchronizes clocks of the nodes.

All experiments presented in the thesis were done on six virtual servers created by the vSphere client application. Each of the six virtual machines had the same working environment. In Table 5.1, configurations that were identical to all experiments are presented. The most of the parameter values were influenced by the information provided by Y-Soft engineers, in order to simulate the behaviour of the Y-Soft distributed system. The only parameter set differently is the key-space size. The value was chosen in order to observe the intensified inconsistent behaviour.

We realized series of tests each focused on modification of a single test configuration parameter. *Probability of Request Types Experiments* analyse the system under different ratio of read and write requests. *W-R Quorum Experiments* analyse the system execution under different values of W and R parameters. *Partition Experiments* are concerned with simulation of partitions and their influence on datastore behaviour. We provide the default values of parameters in all experiments. The modification of parameter values will be stated in each experiment individually:

- **Write:Read ratio = 1:1** - there is roughly an equal amount of write and read requests executed during experiment.
- **R = 1, W = 1** - a coordinator of the request receives 1 acknowledgement from a replica for each type of request.
- **Halt probability = 0** - there are no partitions in the network

In this section we present the most interesting results obtained from the realized experiments. The results are provided in the form of the

Duration of experiment	120 minutes
Number of clients	500
Data value size	2 KB
Key-space size	20
Requests per minute	20000
Size of Riak cluster	5
Replication factor	3
Number of ring partitions	64

Table 5.1: **Experiment configurations:** Note that there are 20000 requests processed in every minute. Each client processes 40 requests per a minute. The number of requests executed on data with identical key is 1000 per a minute.

metrics defined in Section 3

5.1 Probability of Request Types

The goal of the *Probability of Request Types* experiments was to analyse the influence of different write and read requests ratios to the overall consistency and latency of the system. Seeing that the experiments were without partition, all requests executed in the experiment were naturally available. The modified configuration of the experiment is:

- **Write:Read ratio = X:Y** - Analysed ratios of requests types: {95:05, 05:95, 75:25, 25:75, 50:50}

Figure 5.1 depicts the latency of requests executed in five experiments, each with different ratio of request types. We can see, that there is a direct relation between the proportion of write requests and values of requests latencies. Increasing the rate of the write requests increases the overall system latency. When the amount of write requests is low (5%), 75% of requests are processed within 20 milliseconds. On the other hand, when amount of write requests is high (95%) the latency is increased five times and 75% of requests are processed within 100 ms. The maximum values of requests latencies did

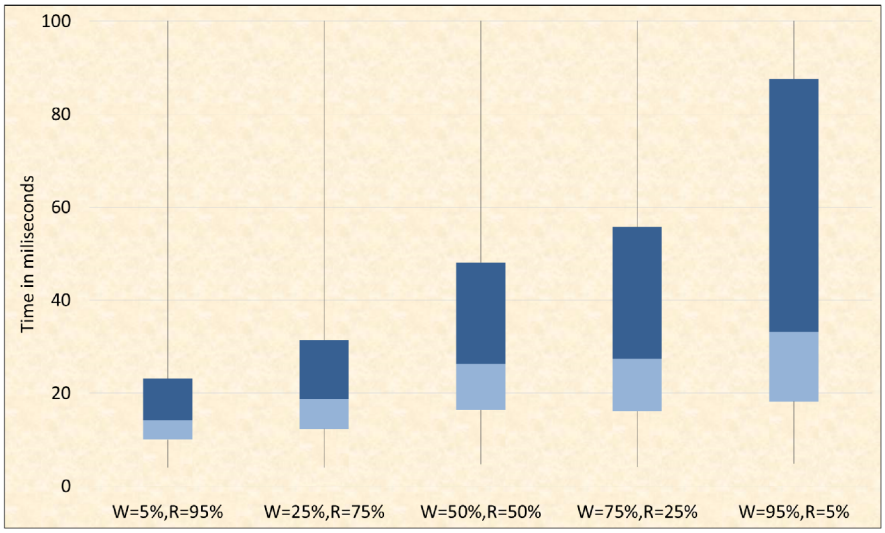


Figure 5.1: **Requests latencies for different ratios of request types:** The values of request latencies are expressed in the form of box-plot graph with emphasized minimum, maximum, 1st, 2nd and 3rd quartile values. The light coloured part corresponds to values between 1st and 2nd quartile and dark coloured part corresponds to values between 2nd and 3rd quartile. The lines represent values between minimum and 1st quartile and 3rd quartile and maximum.

not fit into the figure and scale between [3,4] seconds range.

Figure 5.2 shows Γ values of all inconsistent request clusters defined in Section 3. The interesting result is the high Γ value in experiment with configuration of 95% read and 5% write requests. Although, read requests conflict only with the write requests, which number is low in the experiment, the Γ value is high in comparison to experiments with higher amounts of writes. However, Figure 5.3 shows that the number of ζ conflicting requests is actually below 0.3 %. Hence the consistency of requests is better compared to other request ratios but the conflicting requests have increased Γ value.

Similarly, high percentage of write requests results in low number of ζ inconsistencies, see Figure 5.3. The reason is the low number of read requests as two request clusters consisting only from write requests cannot conflict, see Section 3. Hence, if the number of reads

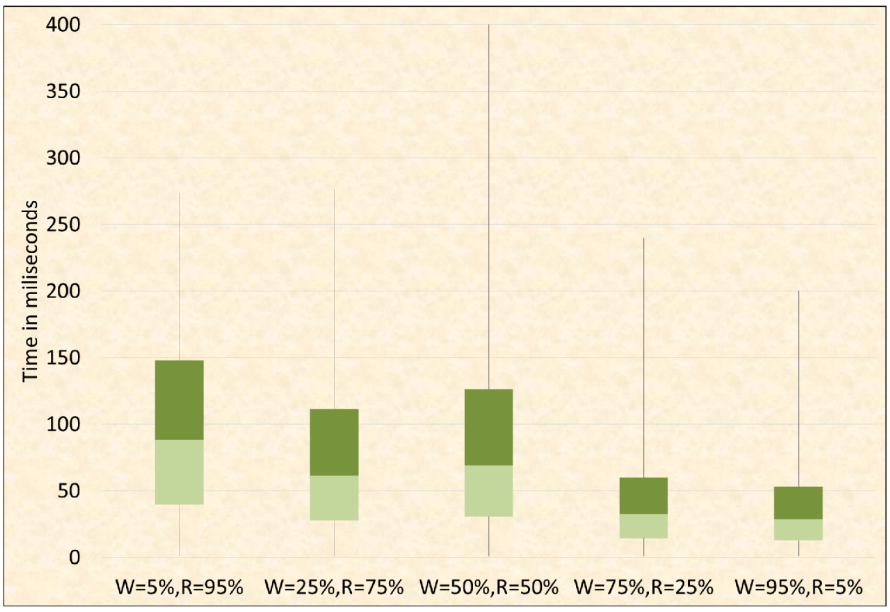


Figure 5.2: **Request clusters Γ-consistency for different ratios of request types:** The values of request clusters Γ-consistency are expressed in the form of box-plot graph with emphasized minimum, maximum, 1st, 2nd and 3rd quartile values. See Figure 5.1 for further details.

is low, the number of read clusters is low too and lesser clusters are conflicting. In contrast to the high read request rate, the Γ metric in the experiment with high amount of write requests indicates lower request staleness. The reason for the result may be high rate of write request that change the value of data in shorter intervals.

Furthermore, in Figure 5.3 we can see that the number of inconsistent requests is the highest when the rate of read requests is equal to the rate of write requests. The amount of inconsistent requests drops with decreasing the rate of any type of request.

5.2 W-R Quorum

The goal of the *W-R Quorum* experiments was to analyse the influence of different W-R quorum on the consistency-latency trade-off.

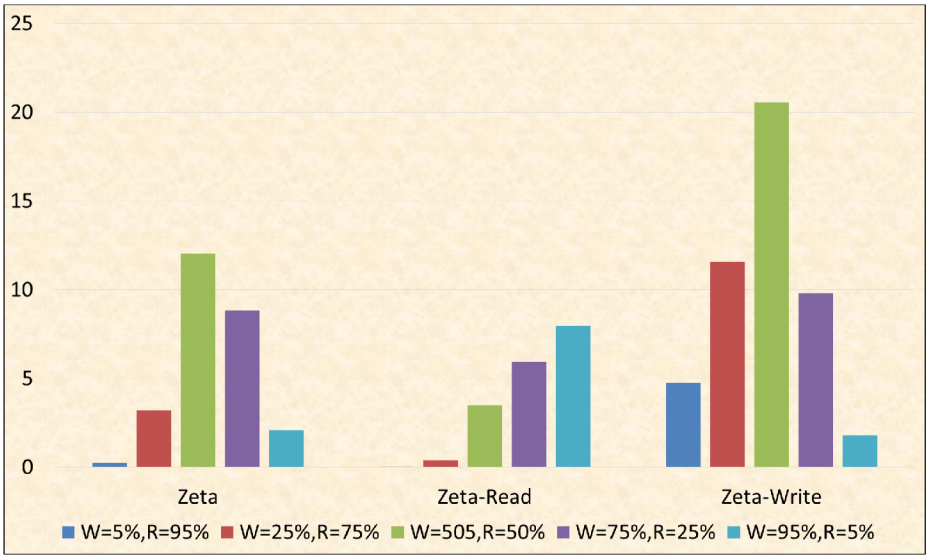


Figure 5.3: **The amount of ζ-inconsistent requests for different ratios of request types:** Different colours of graph columns represent experiments with different configurations. Zeta is the ratio of violates requests to all requests processed in experiment. Zeta-Read and Zeta-Write are ratios of violated read and write requests to all read and writes requests executed in the experiment.

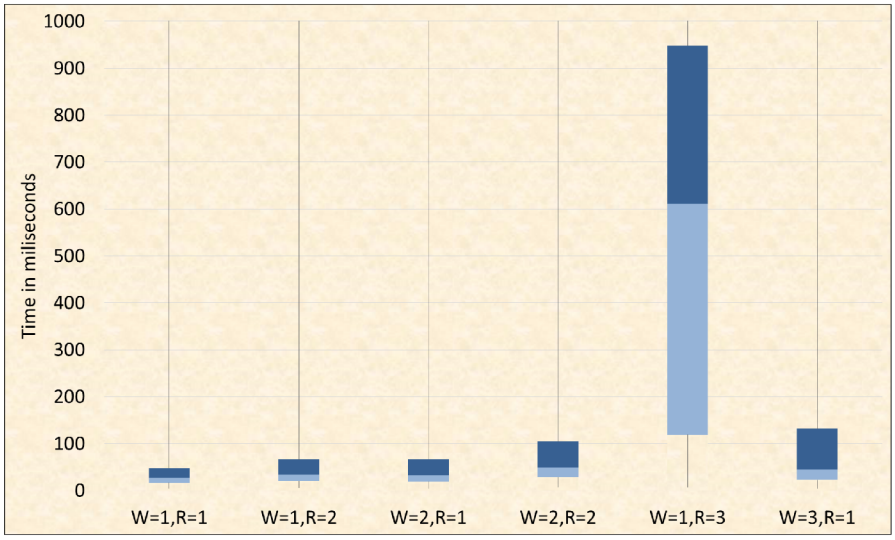


Figure 5.4: **Requests latencies for different W-R Quorum:** The values of request latencies are expressed in the form of box-plot graph with emphasized minimum, maximum, 1st, 2nd and 3rd quartile values. See Figure 5.1 for further details.

The modified configuration of the experiment is:

- **Different {R,W} quorum** - we evaluate all {R,W} quorum, where $R + W \leq N + 1$.

Figure 5.4 depicts latency of requests processed in six experiments with different R-W quorum. We can observe, that the latency of the requests increases when R, W parameters are higher. This is an expectable result, because the increased number of replica acknowledgements inflicts higher delay time of coordinator process execution. However, the interesting result is the latency of the W = 1, R = 3 quorum. The latency of 75% requests is up to one second which is ten times higher than latencies of requests for other quorum configurations. The result is surprising in respect to results obtained in Section 5.1 that implied that write requests impose higher latencies. It was expected that requests latencies in the quorum with higher W values will be higher than in quorum with higher R. On the other hand, the experiment results may be caused by an increased resource

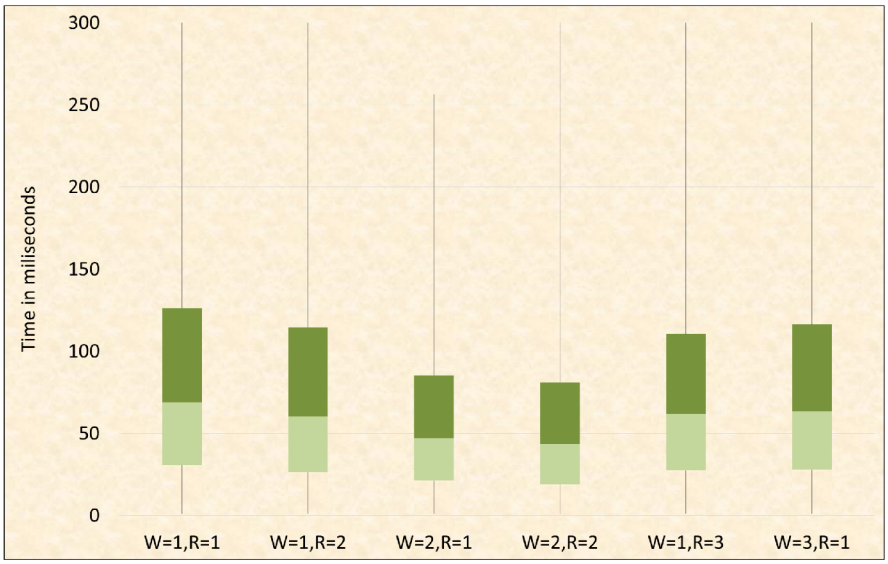


Figure 5.5: **Request clusters Γ-consistency for different R-W quorum:** The values of request clusters Γ-consistency are expressed in the form of box-plot graph with emphasized minimum, maximum, 1st, 2nd and 3rd quartile values. See Figure 5.1 for further details.

consumption on cluster nodes in the time of the experiment. Ignoring the result of the W = 1, R = 3 quorum, the latency of requests increases with higher W values.

Furthermore, the quorum $W + R \geq N + 1$ should provide strong consistency in the systems with no partitions. However, our results show a different behaviour. Although the amount of ζ-inconsistent requests is lower in quorum with higher R, W values, see Figure 5.6, the amount is not zero for quorum $W + R \geq N + 1$ as expected. The number of ζ-violated requests were almost 20 % for low quorum values and 10% for higher quorum values. The Γ value of violated requests is roughly similar for all experiments. The requests show lower Γ values for quorum $\{2, 1\}$ and $\{1, 2\}$, see Figure 5.5. In all experiments, 75% of results were stale less than 150 milliseconds.

The unexpected consistency violation of requests in strong W-R quorum configuration leads to an analysis of possible causes of the result. There are several possible causes of inconsistencies:

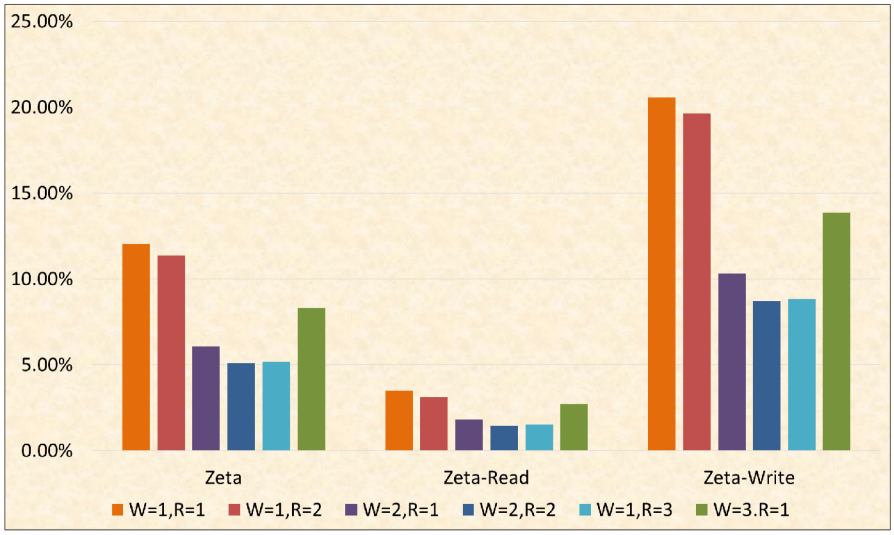


Figure 5.6: **The amount of ζ-inconsistent requests for different W-R quorum:** Different colours of graph columns represent experiments with different configurations. See Figure 5.3 for further details.

- The wrong implementation of consistency metrics - This is not the case, since we manually checked the logs of requests processing for some clusters that were marked inconsistent by the analyser. The manual checking confirmed the existence of conflict in the execution from the logs. For the proof of the inconsistent behaviour, it is sufficient to prove that at least one request was inconsistent.

- The internal error of the Riak application - This is not the case, since we checked the error logs produced by each Riak node participating in the experiment. The execution of experiment did not suffer from any errors. Moreover, we checked the logs of requests processing produced by the coordinator process. The number of acknowledgements from replica nodes was equal to the configuration of the experiment.

- The most promising cause of inconsistency is the value of the DW = 1 configuration that was set in the experiments. The DW represents the number of writes that are durably written to

the low-level storage before acknowledgement. Combining the low DW value with the default conflict resolution mechanisms could lead to the inconsistent behaviour. A detailed analysis is left for the future work.

- Another undiscovered causality.

5.3 Partitions

The goal of the *Partitions* experiments was to analyse the consistency and availability trade-off of in partitioned Riak datastore. The modified configuration of the experiment is:

- **Halt probability** ϵ {**1,5,10**} - The probability of the partition per hour. Each partition has a duration within [Min,Max] interval.
  - **Min** ϵ {**5,15,60,30**} - The minimum duration of partition
  - **Max** ϵ {**10,30,120,600**} - The maximum duration of partition
- **Configuration FX:Min-Max** - The partitions in the system occur with a probability X per hour. The duration of partition is in interval [Min,Max].

Figure 5.7 depicts various durations of partitions with probability ϵ {1,5}, and their influence on overall request latency. The latency of requests is increasing with the partition duration in experiments with one partition per hour. However, in experiments with five partitions per hour, the latency of 75% requests is not dependent on the partition duration. Moreover, latency of requests in experiments with higher number of partitions is lower. A clearer observations can be made from maximum latency values that did not fit into the figure. In experiments with lower partitions durations, the maximum request latency was almost 20 seconds. In contrast, the maximum latency of requests in experiments with higher partition durations was almost 60 second which corresponds to the time of partition detection mechanism. In conclusion, the maximal latency in partitioned system is bounded by the default request timeout, but a direct relation

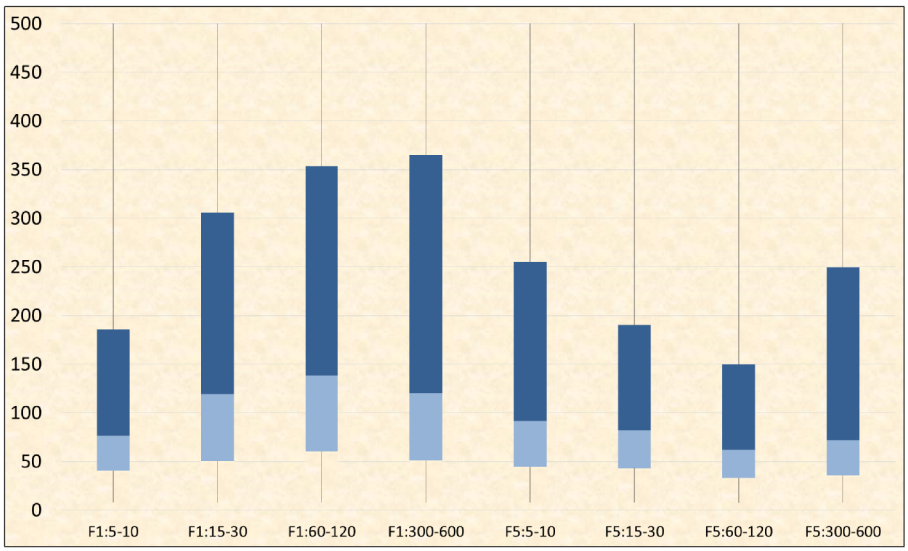


Figure 5.7: **Requests latencies for different parameters of partitions:** The values of request latencies are expressed in the form of box-plot graph with emphasized minimum, maximum, 1st, 2nd and 3rd quartile values. See Figure 5.1 for further details.

between partition duration and request latencies was not observed. The latency is slightly increased in contrast to experiments without partition.

Figure 5.8 shows the Γ values for various durations of partitions with probability ϵ {5,10}. We can observe that the Γ consistency of request clusters is similar to the experiments without partitions. However, the maximum Γ values that did not into the figure are increasing with higher probabilities of partitions and higher durations as well. The interesting result provides the experiment with configuration F10:60-120, in which increased occurrence rate of partitions rapidly influenced the Γ consistency the system. The interesting fact is that the Γ values are lower in the similar experiment with higher duration of partitions.

Finally, Figure 5.9 depicts the number of unavailable requests in the experiment execution. We can observe that the number of unavailable requests is less that 0.7% for each experiment. Hence exper-

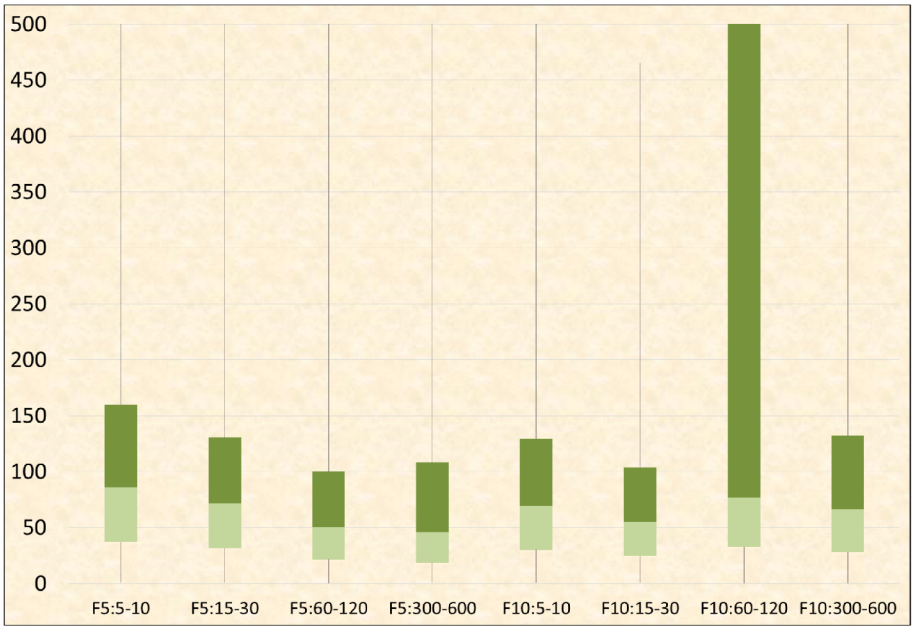


Figure 5.8: **Request clusters Γ-consistency for different parameters of partitions:** The values of request clusters Γ-consistency are expressed in the form of box-plot graph with emphasized minimum, maximum, 1st, 2nd and 3rd quartile values. See Figure 5.1 for further details.

iments show that Riak is highly available datastore even in the presence of partitions. The unavailability of requests slightly increases with amount of partitions per hour. Moreover, the amount of unavailable requests is constant for experiments with equal partition rate but different partition durations. the unavailability of requests is caused by the failure detection mechanism, see Section 2.4.1. The failure detection is provided within a minute interval hence requests that are made minute after the partition occurs may be served without complications.

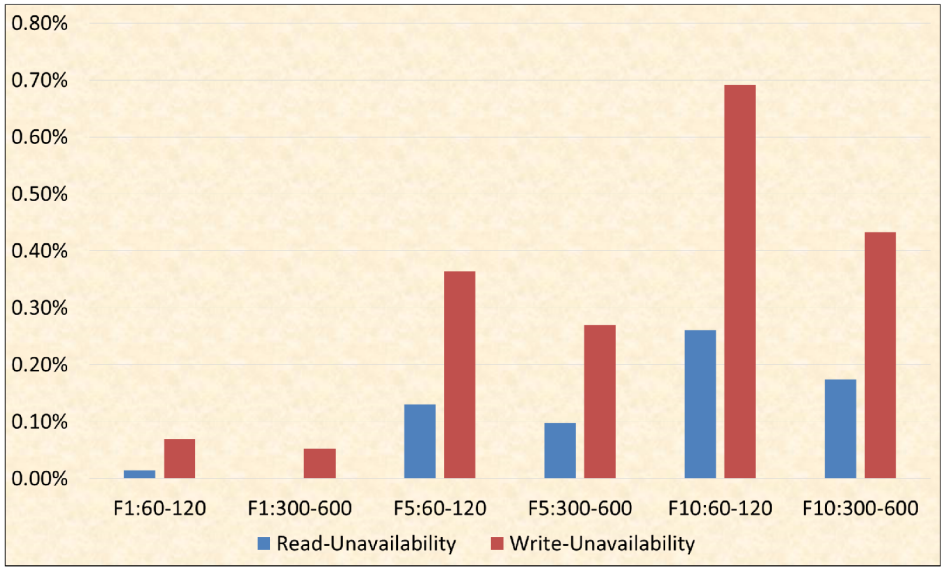


Figure 5.9: **The availability of requests for different parameters of partitions:** Different colours of graph columns represent different types of requests.

6 Conclusions

In the thesis, we prepared an overview of mechanisms used in NoSQL database system Riak. We analysed the problem of consistency, availability and latency trade-off that occurs in NoSQL databases and proposed metrics that evaluate the given properties. Moreover, we implemented distributed testing application that simulates a communication between the database and its users. Finally, exploiting the proposed metrics, we presented results of experiments realized by the testing application.

In the first part of the thesis we provided a deep overview of mechanisms used in Riak distributed databases. The contents of the overview are not present in the standard documentation. The information provided in the overview is based on the Riak official documentation [30] and source code analysis of Riak datastore modules [31, 32]. Moreover, some information presented in the overview was verified exploiting the Riak testing application [33].

Furthermore, we provided independent study of consistency availability and latency properties and listed popular methods used to analyse them in respect to distributed systems. As a result, we proposed a set of metrics for each property. The metrics were exploited in later Riak analysis. The proposed metrics may be reused in the future experiments focused on evaluation of different NoSQL databases and their comparison with Riak.

Another contribution of the thesis was the implementation of the robust distributed testing application based on the set of interfaces designed by the Y-Soft engineers. Our application was one of the first that implemented designed concepts. As a result, we provided a valuable feedback to the software designers. The most notable part of the application is the planner component that generates plan of an experiment based on the input configuration. The planner implementation is flexible, considering many types of input parameters. Consequently, the planner can model various client-database interactions. Moreover, we designed the Riak interceptor application that is able to simulate partitions among the set of nodes in Riak cluster. Taken together, our distributed testing application is able to model many types of client-database interactions and behaviours.

Finally, one of the most interesting contributions are the results obtained from experiments realized by the designed testing application. We summarize the most interesting observations.

The surprising result on Riak consistency was the discovery of the inconsistent state in experiments exploiting the $W + R \geq N + 1$ quorum. The obtained inconsistent results imply that the majority quorum is not sufficient for the implementation of the strong consistency in Riak. The manual analysis of logs from the coordinator showed that each request was acknowledged by the corresponding number of replicas defined by the quorum. We think that the result may be inflicted by a $DW = 1$ configuration combined with the default conflict resolution and the delayed time synchronization on replicas. We leave the detailed analysis of the problem for future works.

The consistency of requests in partitioned networks was comparable to results in $R = 1$, $W = 1$ quorum. We managed to simulate more inconsistent values only in experiments with partitions that occurred periodically within six minutes interval. Hence, the consistency in systems with the irregular and short partitions is not affected in respect to the $R = 1$, $W = 1$ quorum values.

Furthermore, the unavailability of requests were observable only in partitioned networks when a duration of partition was higher then one minute. The result corresponds to the 45 second duration of partition detection. All requests that are executed before the failure is detected are condemned to the error result. The concluding thought is that Riak works with high availability in the presence of partitions and the faults are observed only due to the delay of the partition detection mechanism.

The results from experiments created many directions for the future work. We want to further analyse the consistency trade-offs of the Riak datastore in order to find the source of the inconsistent results in majority quorum configurations. Furthermore, we plan to extend experiments on partitioned networks and analyse the strong consistency PR-PW quorums. Finally, we would like to extend the testing application so that it can be used for analysis of other distributed NoSQL databases.

A Erlang

Erlang is a functional programming language used to implement highly reliable, concurrent and large-scale applications [3, 12]. It is functional, hence recursive functions are used to model cycles in the programs. The concurrency model is based on Erlang processes.

In Erlang, a process belongs to the programming language not an operating system. Processes are light-weight, created and terminated swiftly, sharing no memory with other processes of a virtual machine. The state of a process is held in parameters of the executed function. Two processes can communicate and exchange their state only through message passing. Therefore, they cannot acquire simultaneous access to a critical data [14]. The result is a reliable application without a need for locks and synchronization mechanisms among processes. Consequently, a single Erlang virtual machine is able to create millions of processes. However, the additional process communication for the state exchange causes increased latency in program execution.

Moreover, Erlang is a platform with robust fault-tolerance. If an error occurs during an execution, the process is immediately terminated. The memory states of other processes are unaffected. In addition, Erlang provides high control over crashed processes and their subsequent recovery. Processes can monitor each other if they are linked together. Only linked processes exchange the error and state information after a crash. The process that received the error information decides an appropriate reaction. The reaction is often application specific.

Common patterns of concurrent applications are grouped in Open Telecom Platform (OTP) framework. It is a set of modules and functions dealing with generic behaviour of concurrent applications. Consequently, programmers focus on application specific code. OTP framework is established on common behaviour of all processes: Usually a process is spawned by a function, executes some initialization code and enters a loop. During the loop, it receives or sends messages to other processes and executes a code accordingly. When the process receives an exit signal or an error occurs, it enters a termination phase. Afterwards, it executes the final code and finishes. There are

several common behaviours implemented in the OTP framework.

A generic server is a behaviour of a process handling remote procedure calls from other processes [12]. Calls may be synchronous or asynchronous. They are executed only from processes that have a reference to a generic server id (Pid). After a call, the server executes a handle function (handle_call or handle_cast) and replies back to the client, whenever it is required. Besides, server has specified initialization (init) and termination (terminate) function that execute a code in the beginning and end of the process lifetime. Between the phases, process is in loop, waiting for calls from other processes.

A finite state machine behaviour implements processes with finite number of states [12]. A state represents a function. The function is triggered if the process is currently in the state and event occurs. Each state defines a next state the process enters after the function execution. In contrast to generic server, processes communicate through events. The event is usually synchronous, asynchronous or timeout. Similar to the generic server, a state machine starts in an initialization state and finishes with a termination code. Between the phases, the execution is defined by states combined with events.

A generic event behaviour is used for event management [12]. An event manager is a process that maintains (event handler, state) pairs. The event handler is a process with defined reactions to the events. The event manager can register event handlers or receive events. When manager receives an event, it calls all registered event handlers and updates their state. If the event handler is dispensable, it is deleted by the manager.

Described generic behaviours are the most common process types of Riak modules.

Bibliography

[1] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, Feb. 2012.

[2] A. Aiyer, L. Alvisi, and R. A. Bazzi. On the availability of non-strict quorum systems. In *Distributed Computing*, pages 48–62. Springer, 2005.

[3] J. Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2007.

[4] H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[5] P. Bailis and A. Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[6] P. Bailis, S. Venkataraman, J. M. Hellerstein, M. Franklin, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. Technical Report UCB/EECS-2012-4, EECS Department, University of California, Berkeley, Jan 2012.

[7] D. Bermbach and J. Kuhlenkamp. Consistency in distributed storage systems. In V. Gramoli and R. Guerraoui, editors, *Networked Systems*, volume 7853 of *Lecture Notes in Computer Science*, pages 175–189. Springer Berlin Heidelberg, 2013.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[11] Ericsson. Distributed erlang. `http://erlang.org/doc/reference_manual/distributed.html`.

[12] Ericsson. Erlang. `http://www.erlang.org/`.

[13] Ericsson. Erlang empd. `http://www.erlang.org/doc/man/epmd.html`.

[14] Ericsson. Erlang processes. `http://www.erlang.org/doc/reference_manual/processes.html`.

[15] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[16] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[17] W. Golab, X. Li, and M. A. Shah. Analyzing consistency properties for fun and profit. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 197–206. ACM, 2011.

[18] W. Golab, M. R. Rahman, A. A. Young, K. Keeton, J. J. Wylie, and I. Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 28:1–28:2, New York, NY, USA, 2013. ACM.

[19] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[20] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed

caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[22] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[24] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology*, pages 369–378. Springer, 1988.

[25] S. Mullender, editor. *Distributed Systems (2Nd Ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[26] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[27] D. J. Sheehy and D. Smith. Bitcask. a log-structured hash table for fast key/value data. *White paper, April*, 2010.

[28] P. Software. Rabbitmq. `https://www.rabbitmq.com/`.

[29] C. Strauch, U.-L. S. Sites, and W. Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 2011.

[30] B. Technologies. Riak. `http://basho.com/riak/`.

[31] B. Technologies. Riak core application. `https://github.com/basho/riak_core`.

[32] B. Technologies. Riak kv application. `https://github.com/basho/riak_kv`.

[33] B. Technologies. Riak test application. `https://github.com/basho/riak_test`.

[34] D. Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, Dec. 2013.

[35] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 464–474. IEEE, 2000.