Scalable Persistent Storage for Erlang Theory and Practice

Amir Ghaffari, Natalia Chechina, Phil Trinder
Department of Computer Science,
School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh, EH14 4AS, UK

June 2013

Abstract

Distributed computing platforms, like clusters, grids or clouds, have become popular due to their scalability and cost-effectiveness. For taking the advantage of the scalable platforms, many programming languages have been designed to support distributed programming. Erlang has become one of the most popular programming language for developing distributed application. As any other programming language, Erlang needs distributed databases for saving and retrieving data . In the RELEASE project, we are working to scale up Erlang/OTP for massively parallel and distributed platforms. In this paper we evaluate the scalability of some of the mostly used DBMSs among Erlang developers. Our study shows that in theory Riak is a scalable, available and failure-tolerant distributed database. We investigate the scalability limits of Riak in practice, and find that Riak doesn't scale beyond ≈60 nodes. Moreover, our benchmark reveal that Riak provides a highly available and fault tolerant service.

1 Introduction

Scalability is one of the most desirable attributes of parallel and distributed systems. Scalability is the ability of a system to take full advantage of the growth in a number of available resources[1]. Erlang has become one of the most popular programming languages for development parallel and distributed applications because of features like data immutability, share-nothing concurrency, asynchronous message passing, and process location transparency[2][3]. Like any other application, an Erlang application needs databases to store data persistently, but scalability limits of database system can affect on the scalability of Erlang application.

The RELEASE project[4] aims to improve the scalability of Erlang onto commodity architectures with the following structure:

- 1-5 cluster(s)
- 100 hosts per cluster
- 32-64 cores per host

In RELEASE, we are working to scale up Erlang in three levels:

1. Computation, i.e. *global names update* and *transitive connection among nodes*
2. In-memory data structure, i.e. *Erlang Term Storage* (ETS)
3. Persistent data structure, i.e. *distributed DBMSs for Erlang.*

Here we address the last of these goals by studying the scalability of some of the most popular distributed databases among Erlang developers, i.e. Mnesia, CouchDB, Riak, and Cassandra. We evaluate their suitability and limitations for large-scale distributed platforms.

We start by listing the main principles of scalable and available persistent storage (Section 2). Then, we assess a number of database management systems (DBMSs) against these principles and we evaluate their technologies and their suitability for large-scale architectures (Section 3). Next, we benchmark the scalability and availability of the Riak NoSQL DBMS in practice (Section 4). Finally, we discuss the conclusion together with the future work (Section 5).

Contributions

- Identified a theoretical analysis of Erlang persistent storage technologies against requirements of scalable persistent storage, considering Mnesia, CouchDB, Riak, and Cassandra NoSQL DBMSs (Section 3).
- Investigated the scalability limits of the Riak NoSQL DBMS by using Basho Bench on 348-node cluster with 2784 CPU cores (Section 4).

2 Scalable Persistent Storage

Large-scale distributed architectures such as clusters, clouds, and grids have emerged to provide scalable high performance computing resources from low-cost unreliable components, i.e. commodity nodes and unreliable networks[**?**]. Parallel and distributed database systems have been developed to make distributed processing such as replication, fragmentation, and query processing, easier and more efficient in unreliable distributed environments[5].

In the following paragraphs, we outline the principles of highly scalable and available persistent storage:

- *Data Fragmentation*: A distributed database consists of a collection of nodes, each of which maintain subsets of the data (Figure 1). Data fragmentation improves performance by spreading the loads across multiple nodes and therefore increases the level of concurrency[5]. A scalable fragmentation approach should have the following features:

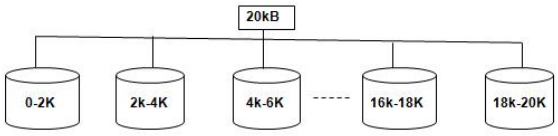

Figure 1: 20k data is fragmented among 10 nodes evenly

1. A decentralized model: Data is fragmented among nodes without any central coordination[6]. Decentralized approaches show a better throughput by spreading the loads over a larger number of servers and increases availability by eliminating single point of failure.
2. Load balancing: Fragment the data evenly among the nodes. A desirable load balancing mechanism should take the burden of load balancing of the shoulders of developers. Decentralized techniques like consistent hashing[7] may be employed to evenly spread the data across the nodes.
3. Location transparency: In large scale systems placing the fragments in the proper location is very difficult to manage. Thus, a preferable method for placement of fragments should be carried out systematically and automatically, without requiring developer intervention. Moreover, reorganizing database nodes does not impact on the programs that access them[**?**].
4. Scalability: The departure or arrival of a node should only affect its immediate neighbours and other nodes remain unaffected. A new node to the system accepts a roughly equivalent amount of data from other available nodes and when a node goes down, its load is evenly spread among the remaining available nodes[6].

- *Data Replication*: Maintaining multiple copies of data improves availability and performance[5] (Figure 2). Replication improves the performance of read-only queries by providing data from the nearest replica (*data localization*). Replication may also increases the system availability by removing a single points of failure. A scalable mechanism for replication should have the following features:

1. Decentralized model: Data is replicated among nodes without using concept of master, so each node has a full DBMS functionality[6].

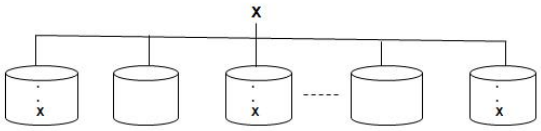

Figure 2: Record X is replicated on 3 nodes

A P2P model is desirable because each node is able to coordinate the replication and this can improve performance by spreading the loads, i.e. computational and network traffic loads, over the available nodes.

2. Location transparency: The placement of replicas is handled systematically and automatically[8]. Managing the location of replicas can be a difficult task in large-scale systems.
3. Asynchronous replication: A write command is considered complete as soon as the local storage acknowledges it, without having to wait for the remote acknowledgment[9]. In large-scale systems on slow communication networks, e.g. wide area networks, it can be very time consuming to wait for all replicas confirmation. Many Internet-scale applications such as email services, social networking services, may be able to tolerate some inconsistency among the replicas in the sake of performance. This approach would violate the consistency of data over time, so *eventual consistency* can be used to address this problem.

   Eventual consistency[10] is a specific form of weak consistency that updates propagate throughout the system and eventually all accesses will return the last updated value. Domain Name System (DNS) is the most popular system which employs eventual consistency.

- *Partition Tolerance*: The loosely coupled nature of large-scale distributed systems requires an approach to cope with failures like network partition or node failure. A highly available system should continue to operate despite loss of connection between some nodes. The CAP theorem[9] states a database cannot simultaneously guarantee the consistency, availability, and partition-tolerance. Thus we must sacrifice the strong consistency to achieve partition-tolerance and availability (Figure 3). An eventual-consistency mechanism can improve availability by providing a weakened consistency. In this approach, if some nodes fail or become unreachable while an update is being executed, the effects will be reflected on the data residing on the failed nodes as soon as they recover from the failure.

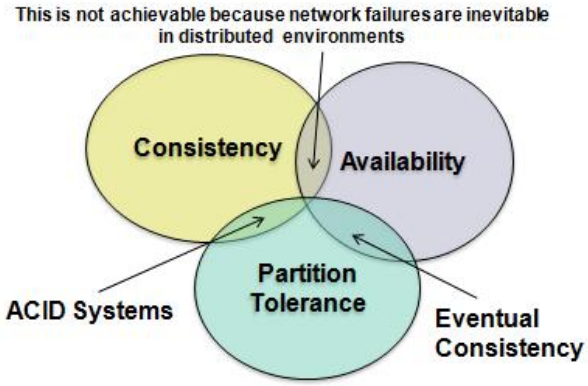

Figure 3: CAP Theorem

- *Query Execution Strategies*: Queries provide a mechanism for retrieving specific information from a database. A favorable query processing approach, in addition to provide a good performance, hides the low-level details of replicas and fragments[5]. A scalable query strategy should have the following features:

    1. Local Execution: On large-scale geographically distributed systems which data is fragmented over nodes, query response time may become very high due to communication overhead. Scalable techniques like MapReduce[11] reduce the amount of exchanging data between the participating nodes in a query execution by *local processing*. In local processing approach, a query is passed to where the data lives, rather than transferring a large amount of data over the network to be processed on a client. Then, the results of local executions are combined into a single output.
    2. Parallelism: In addition to improving the performance, local processing increases parallel execution of large computations by dividing the query to sub-queries and spread them among multiple nodes. So a good approach should exploit the local resources as much as possible to achieve the maximum efficiency.
    3. Fault Tolerance: Since large-scale databases are distributed on a large cluster of nodes, node failure is very common and thus a query approach must tolerate communication or node failures. In the case of failure, the query coordinator should mask that failure by running the query on the available replicas.

3 Scalability of Erlang DBMSs

Recently, there has been a lot of interest in NoSQL databases to store and process large-scale data sets, e.g. Amazon Dynamo[6], Google BigTable[12], Facebook Cassandra[13], Yahoo! PNUTS[14]. These companies have rejected traditional Relational Database Management Systems (RDBMSs) because they could not provide the required features such as high scalability, high availability and low latency, for large scale unreliable distributed environments.

In this section we analyse a number of popular NoSQL DBMSs for Erlang systems in terms of scalability and availability. We evaluate them against the principles outlined in Section 2.

3.1 Mnesia

Mnesia is a distributed DataBase Management System (DBMS) written in Erlang for industrial telecommunications applications[15]. The Mnesia data model consists of tables of records and attributes of each record can store arbitrary Erlang terms. Mnesia provides ACID (Atomicity, Consistency, Isolation, Durability) transaction which means either all operations in a transaction are applied on all nodes successfully, or, in case of a failure, it doesn't have any effect on the nodes. In addition, Mnesia guarantees that transactions which manipulate the same data records do not interfere with each other. To read and write from/to a table through transaction Mnesia sets and releases locks automatically. Fault-tolerance is provided in Mnesia by replicating tables on different Erlang nodes. In Mnesia the placement of replicas should be mentioned explicitly, e.g. Code1 shows a *student* table is replicated on three Erlang VMs.

Code 1: Mnesia Explicit Placement of Replicas

```
mnesia:create_table(student, [{disc_copies,
 [node1@sample_domain, node2@sample_domain,
node3@sample_domain]},{type, set},{attributes,
[id,fname,lname,age]},{index,[fname]}]).
```

Generally, to read a record only one replica of that record is locked (usually the local one), but to update a record, all replicas of that record are locked and must be updated. This can become a bottleneck for write operations, when one of the replica is not reachable due to node or network failures. To address this problem Mnesia offers *dirty operations* that manipulate tables without locking all replicas. But in case of using dirty operations, there is no mechanism to eventually complete the update on the all replicas, which leads to data inconsistency.

Mnesia also has a limitation in size of tables. Mnesia inherits limitations of DETS tables, and since DETS tables use 32 bit file offsets, the largest possible Mnesia table per an Erlang VM is 2GB. To cope with large tables, Mnesia introduces a concept of *table fragmentation*. A large table can be split into several smaller fragments on different Erlang nodes. Mnesia employs a hash function to compute the hash value of a record key, and then that value is used to determine to which fragment the record belongs. The downside of the fragmentation mechanism is that the placement of fragments should be specified explicitly by the user. Code 2 shows how fragments are placed explicitly in Mnesia:

Code 2: Mnesia Explicit Placement of Fragments

```
mnesia:change_table_frag(SampleTable,
 {add_frag, List_of_Nodes}).
```

Query language in Mnesia is Query List Comprehensions (QLCs). QLCs are similar to ordinary list comprehensions in Erlang programming language, e.g. the following code (Code 3) returns *lname* for student with id=1.

Code 3: Mnesia Example Query

```
Query = query [S.lname ||
S <- table(student), S.id == 1] end,
  Result =
        mnesia:transaction(
             fun() ->
                      mnemosyne:eval(Query)
             end).
```

Here we summarize the Mnesia limitations for large-scale systems:

- Explicit placement of replicas
- Explicit placement of fragments
- Limitation in size of tables
- Lack of support for eventual consistency

3.2 CouchDB

CouchDB ("Cluster Of Unreliable Commodity Hardware") is a schema-free document-oriented database written in Erlang[16]. Data in CouchDB is organised in a form of a document. Schema-less means that each document can

be made up of an arbitrary number of fields. A single CouchDB node employs a B-tree storage engine that allows searches, insertions, and deletions be handled in logarithmic time. Instead of traditional locking mechanisms for concurrent updates, CouchDB uses Multi-Version Concurrency Control (MVCC) to manage concurrent access to the database. MVCC makes it possible to run at full speed all the time, even when a large number of clients use the system concurrently. In CouchDB creating views and aggregation reports is possible by joining documents through Map/Reduce technique. A view is made up of a map JavaScript function and optionally a reduce function.

In CouchDB, fragmenting data over nodes is handled by Lounge[17]. The Lounge is a proxy-based partitioning/clustering framework for CouchDB. Lounge applies a hash function on the document's ID to find on which shard save and retrieve the documents. Hash function is a consistent hash which balances the storage loads evenly across the partitions. Lounge doesn't exist on all CouchDB nodes. In fact, Lounge is a web proxy that distributes HTTP requests among CouchDB nodes. Thus, to remove single points of failure we need to run multiple instances of Lounge, i.e a multi-server model.

CouchDBs replication system synchronizes all the copies of the same database by sending the last changes to all the other replicas. Replication is an unidirectional process, which means the changed documents are copied from one replica to the others and not automatically vice versa. The replicas placement should be handled explicitly.

Code 4: CouchDB Explicit Placement of Replicas

```
POST /_replicate HTTP/1.1
{"source":"http://localhost/database",
"target":"http://example.org/database",
 "continuous":true}
```

In Code4, "continuous":true means that CouchDB will not stop the replication and automatically send any new changes of the source to the target by listening to the CouchDBs changes API. Maintaining consistency between replicas in CouchDB is an eventual consistency which means the document changes are periodically copied between replicas. A conflict occurs when a document has different information on different replicas. In the face of conflict between replicas, CouchDB employs an automatic conflict detection mechanism. CouchDB does not attempt to merge the conflicted visions automatically and it just tries to find the latest version by using version of documents, and the result will be represented as the winning version. But the losing versions are kept in the document's history and client applications can use them to resolve the conflicts in a way that makes sense for them.

In summary, CouchDB has the following limitations that can be a bot-

tleneck for large-scale systems:

- Explicit placement of replicas
- Explicit placement of fragments
- Multi-server model for coordinating fragmentation and replication
- Lounge which handles data partitioning is not part of each CouchDB node

3.3 Riak

Riak is a NoSQL, schemaless, open source, distributed key/value data store primarily written in Erlang[18]. Riak is scalable, available, and fault tolerant database suitable for large-scale distributed environments, such as Clouds and Grids. Riak is highly fault-tolerant due to its master-less structure by offering a no single point of failure design. Riak is commercially proven by being used by large and well known companies and organizations such as Mozilla, Ask.com, AOL, DotCloud , GitHub.

In Riak data is organized into buckets, keys, and values. A key/value pair is stored in a bucket, and values can be retrieved by their unique key. For non-key-based query, MapReduce is a scalable approach in which query can be submitted through client API, i.e. HTTP, protobufs.

Data fragmentation in Riak is handled through consistent hashing technique[18]. Consistent hash distributes data over the nodes dynamically and adapts as nodes join and leave the system. The placement of fragments is done implicitly and programmer does not need to specify any location for the fragments explicitly.

Availability in Riak is provided by using replication and hand-off technique. By default each data bucket in Riak is replicated on 3 different nodes. But the number of replica (N) is a tunable parameter and can be set per each bucket. Other tunable parameters are read quorum (R) and write quorum (W). Quorum is the number of replicas that must respond to a read or write request before it is considered successful. If $R + W > N$, Riak provides strong consistency which guarantees subsequent accesses will return the previously updated value. When $W + R <= N$, Riak provides a weak consistency because R and W will not overlap each other and there may be some nodes which keep stale data.

In hand-off technique, when a node fails temporarily, due to node failure or network partitioning, neighbouring nodes take over the failed node's duties. When the failed node comes back up, Merkle tree is used to determine which records need to be updated. Each node has its own Merkle tree for the keys which it hosts. Merkle trees reduce the amount of data that is needed to be transferred for checking the inconsistencies among replicas.

Riak provides eventual consistency, i.e. an update is propagated to all replicas asynchronously. However, under certain conditions such as node failure or network partitions, updates may not reach to the all replicas. Riak employs vector clocks (or vclock) to handle such inconsistencies by reconciling the older version and the divergent version.

The default Riak backend storage is Bitcask. Although Bitcask provides low latency, easy backup, restore, and it is robust in the face of crashes but it has one notable limitation. Bitcast keeps all keys in RAM and therefore can store a limited number of key per node. For this reason, Riak users apply other storage engines to store billions of records per node.

LevelDB is a fast key-value storage library written at Google[19]. It does not have Bitcask RAM limitation.Moreover, LevelDB provides an ordered mapping from keys to values whereas Bitcask is a hash table. LevelDB supports atomic batch of updates. Batch of update may also be used to speed up large updates by placing them into the same batch. A database may only be opened by one process at a time. There is one file system directory per each LevelDB database where all database content is stored. To improve the performance, adjacent keys are located in the same block. A block is a unit of transfer data to and from persistent storage and default block size is approximately 8192 bytes. Each block is individually compressed before being written to persistent storage and compression can be disable. It is possible to force a checksum verification of all data that is read from the file system. Eleveldb is an Erlang wrapper for LevelDB included in Riak, so there is no need to separate installation. LevelDB read access can be slower in comparison with Bitcask because LevelDB tables are organized into a sequence of levels. Each level stores approximately ten times as much data as the level before it. For example if 10% of the database fits in memory, one seek is needed to reach the last level. But if 1% fits in memory, LevelDB will need two seeks. So using Riak with LevelDB as storage engine can provide a suitable data store for a large amount of data.

The evaluation shows in theory Riak meets scalability requirements of a large-scale distributed system . Here we summarized the requirements:

- Implicit placement of replicas
- Implicit placement of fragments
- Bitcask has limitation in size of tables but LevelDB has no such limitation
- Eventual consistency that consequently brings a good level of availability
- No single point of failure by using a peer to peer model
- Scalable query execution approach by supporting MapReduce queries

3.4 Other Distributed DBMSs

There are other distributed DBMSs which are not written in Erlang but Erlang application can access to them by using a client library. The Apache Cassandra is a highly scalable and available database written in Java recommended for commodity hardware or cloud infrastructure [20]. Cassandra is in use at Twitter, Cisco, OpenX, Digg, CloudKick, and other companies that have large data sets. Cassandra offers an automatic, master-less and asynchronous mechanism for replication. Cassandra has a decentralized structure where all nodes in a cluster are identical. Thus, there are no single points of failure and no network bottlenecks. Cassandra provides a ColumnFamily-based data model richer than typical key/value systems. Large scale queries can be run on a Cassandra cluster by using Hadoop MapReduce. Hadoop runs MapReduce jobs to retrieve data from Cassandra by installing a TaskTracker on each Cassandra node. This is an efficient way for retrieving data because each TaskTracker only receives queries for data that the local node is the primary replica. This avoids the overhead of the Gossip protocol.

Since both Riak and Cassandra are inspired by Amazon's description of Dynamo[6], their methods for load-balancing, replication, and fragmentation are roughly the same. So we don't repeat all the details here. Erlang applications employ the Thrift API to use Cassandra. There are also some client libraries for common programming languages such as Erlang, Python, Java, which are recommended to use instead of raw Thrift. Cassandra meets the general principles of scalable persistent storages:

- Implicit placement of replicas
- Implicit placement of fragments
- ColumnFamily-based data model
- Eventual consistency which leads to high availability
- No point of failure by using a peer to peer model
- Scalable query execution approach by integrating Hadoop MapReduce

3.5 Discussion

- Our theoretical evaluation shows that Mnesia and CouchDB have some scalability limitations, i.e implicit placement of replicas and fragments, single point of failure due to lack of P2P model. (Sections 3.1, 3.2)
- Dynamo-style NoSQL DBMS like Riak and Cassandra do have the potential to provide scalable storage for largely distributed architecture as required by RELEASE project. (Sections 3.3, 3.4)

- In the next section we investigate the scalability limitation of Riak to see how it scales in practice.

4 Scalability and Availability of the Riak NoSQL DBMS in Practice

This section investigates the *scalability* limitations of the Riak DBMS. By scalability we mean how system throughput increases by adding resources, i.e. Riak nodes. In addition to scalability, we measure the *availability* and *elasticity* of Riak. In the availability benchmark, we examine the effect of node-failure on a Riak cluster. Elasticity is Riak's ability to cope with loads dynamically on a running system when number of nodes in the cluster changes.

For performing the benchmarks, we have employed Basho Bench, a benchmarking tool for NoSQL DBMS[21]. Basho Bench is an Erlang application that exposes a pluggable driver interface and can be extended to serve as a benchmarking tool against a variety of data stores.

4.1 Experiment Setup

Platform

Benchmarks are carried out on the Kalkyl cluster [22]. The Kalkyl cluster consists of 348 nodes with 2784 64-bit processor cores which are connected via 4:1 oversubscribed DDR Infiniband fabric. Each node has 24GB RAM memory and 250 GB hard disk. The Kalkyl cluster is running Scientific Linux 6.0 , a Red Hat Enterprise Linux. Each node comprises Intel quad-core Xeon 5520 2.26 GHz processors with 8MB cache. Riak data is stored on the local hard drive of each node.

Parameters

In the benchmarks, Riak version 1.1.1 has been used. The number of partitions, sometimes referred to as virtual nodes or vnodes, is 2048. Generally, each Riak node hosts $\frac{number\ of\ vnodes\ in\ the\ cluster}{number\ of\ nodes\ in\ the\ cluster}$ vnodes. Thus, based on this formula, each Riak node will host $\frac{2048}{100} \approx 20$ vnodes. Riak documentation recommends 16-64 vnodes per node, e.g. 64-256 vnodes for a 4-node cluster. We keep the default setting of replication during the benchmark, so data is replicated to three nodes on the cluster. Also, the number of replicas that must respond to a read or write request is two, which is the default value.

4.2 How Does the Benchmark Work?

We have done the benchmarks on a cluster nodes wherein each node can be one of two types: *traffic generators* or *Riak nodes*. A traffic generator node runs one copy of Basho Bench which generates and sends database commands to Riak nodes. A Riak node contains a complete, independent copy of the Riak package which is identified by an IP address and a port number. Figure 4 shows how traffic generators and Riak nodes are organized inside a cluster. There is one traffic generator per each three Riak nodes, e.g. for 20 Riak nodes there are 7 generators ($\frac{20}{3} \approx 7$).

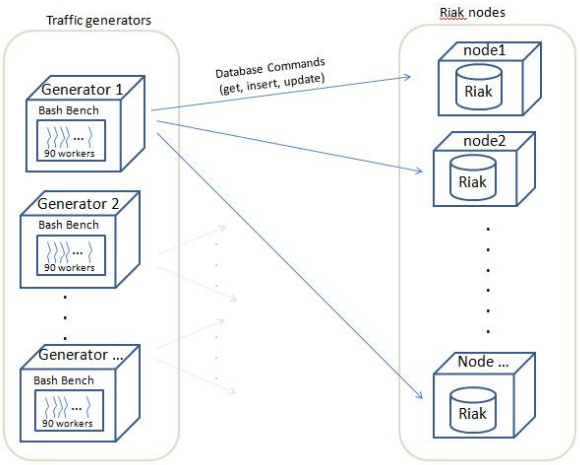

Figure 4: How generators and Riak nodes are organized

Each Basho Bench application creates and runs 90 workers, i.e. Erlang process, in parallel (Figure 5). Then, every worker process selects randomly an IP address from a predefined list of Riak nodes. A worker process also selects a database operation, i.e. *get*, *insert*, or *update*, randomly and submits corresponding HTTP or Protocol Buffer command to the selected IP address and port number. The default port numbers for HTTP communication is [*8098*] and for protocol buffer is [*8087*]. A list of database operations and corresponding HTTP commands is shown below:

1. *Get* corresponds to the HTTP GET command
2. *Insert* corresponds to the HTTP POST command
3. *Update* corresponds to the HTTP PUT command

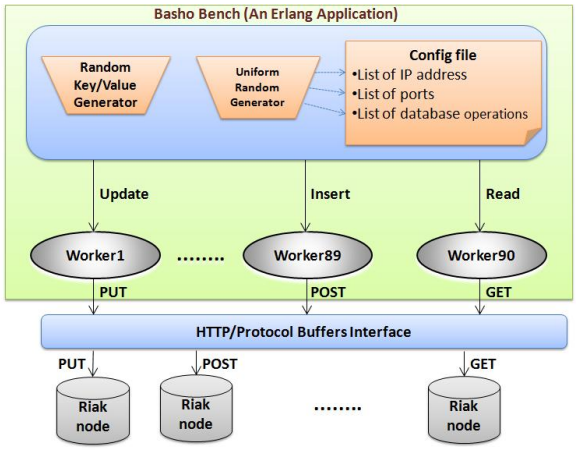

Figure 5: Basho Bench Application

4.3 Scalability Benchmark

In the scalability benchmark, we measure how adding more Riak nodes to the cluster can increase the throughput. We run the scalability benchmark on 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100-node clusters. In each stage, traffic generator nodes issue database commands, i.e. *get*, *insert*, *update*, as mush as the cluster can serve. Then, we delete the data and after adding more Riak nodes, the benchmark is run on a larger cluster. We run the scalability benchmark three times and Figure 6 represents the median of three executions. The results reveal that Riak scales up linearly to 60 nodes, but it can not scale up beyond 60 nodes. We see from Figure 6 that variation (the green lines) is very small for the clusters with size less or equal than 60 nodes. But clusters with more than 60 nodes behave erratically and we see a significant variation in the throughput.

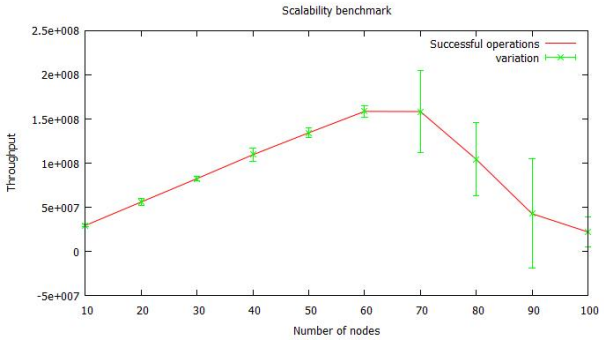

Figure 6: Riak Scalability

Figure 7 shows the number of failed operations in the scalability benchmark. We see the number of failures is 0 for the clusters with size less or

equal than 60 nodes. But when the number of nodes exceeded 60, errors emerge. For example there are 2 million failures on 70-node cluster (1.3% failures). The log files show that the reason of failures is timeout.

Figure 8 presents the average latency of successful operations. We see that the average latency for 60-node cluster is ≈21 millisecond, which is much less than the timeout (60 seconds).

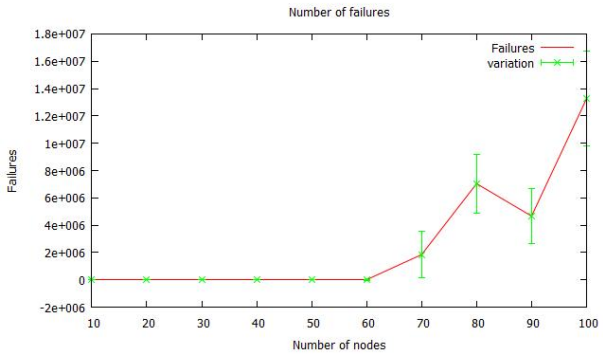

Figure 7: Failures in the scalability benchmark

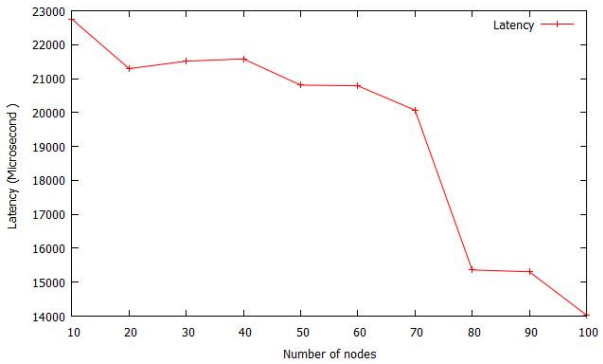

Figure 8: Commands latency in the scalability benchmark

4.4 Identifying the Scalability Bottleneck(s)

Profiling Resources Usage

To identify any possible bottlenecks in the Riak scalability, we measure the usage of processor (*CPU*), random access memory (*RAM*), *disk*, and *network*.

Figures 9 and 10 compare the usage of CPU and RAM for Riak and generator nodes respectively. The maximum usage of CPU is ≈550% which means 5.5 cores of 8 available cores are used. Moreover, the maximum usage

of memory is ≈3% which means 720MB (total RAM memory is 24GB). The result reveals that CPU and RAM can not be bottleneck for Riak scalability because processor and RAM usage are actually quite low.

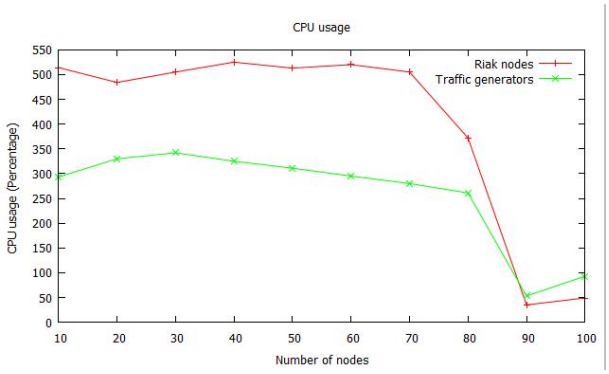

Figure 9: CPU Usage During the Scalability Benchmark

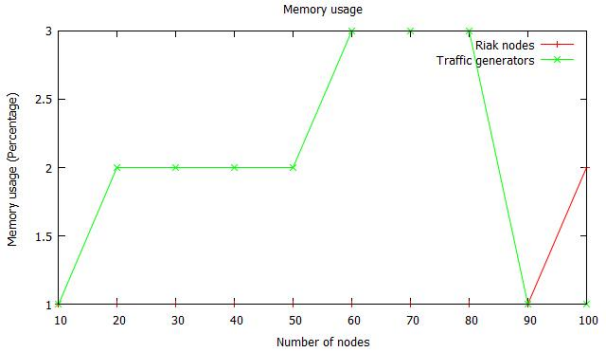

Figure 10: RAM Memory Usage During the Scalability Benchmark

Figure 11 depicts the percentage of time that *disk* spends in servicing requests. The maximum usage of disk belongs to 10-node cluster which is ≈10%. Disk is almost not busy and so it can not be a bottleneck for Riak nodes. Disk usage for traffic generators is ≈0.5% which is negligible and cannot be considered as a bottleneck.

In network profiling, we measure *sent*, *received*, and *retransmitted* packets for both Riak and generator nodes (Figures 12 and 13). Increase in sent and received packets is consistent with the increase of throughput. The number of sent and received packets increases linearly up to 60 nodes and beyond that there is a significant decrease in the number of sent and received packets. We measure retransmitted packets to identify whether there is TCP incast [23], [24].

In general, TCP Incast occurs when number of storage servers send a huge amount of data to a client that Ethernet switch is not able to buffer the

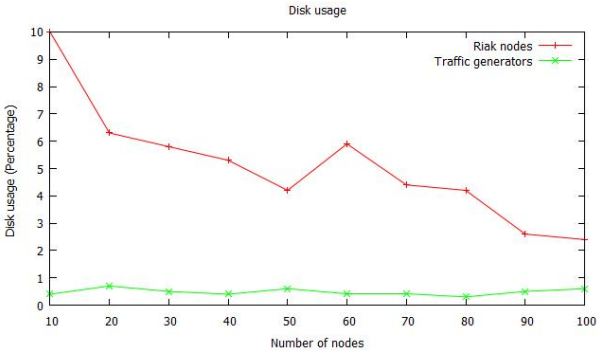

Figure 11: Disk usage during the benchmark

packets. When TCP incast strikes, the number of lost packets increases and consequently causes a growth in the number of *retransmitted* packets. But measuring the number of retransmission packets shows that TCP incast has not occurred during the benchmark. We find that the maximum number of retransmitted packets is 200 packets, which is negligible.

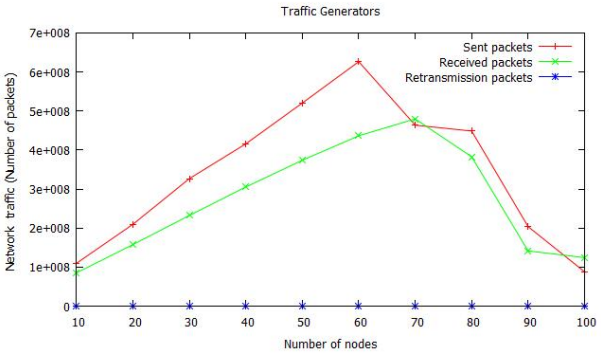

Figure 12: Network traffic for generator nodes

Network traffic comparison between generators and Riak nodes shows that Riak nodes produce 5 times more traffic than generators. For example on 10-node cluster, 100 million packets are sent from generators nodes but on the same size cluster, 500 million packets are sent from Riak nodes. Riak nodes have heavier traffic because in addition to communicating with generators, they also need to communicate with the other Riak nodes to replicate data.

Investigating the Riak Source Code

Profiling results for CPU, RAM, disk, and network show that they can not be bottleneck for the Riak scalability, thus we investigate the Riak source

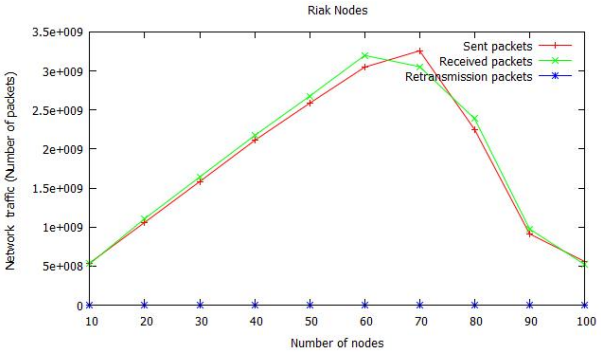

Figure 13: Network traffic for Riak nodes

code to find where is the bottleneck. First, we profiled all the *gen_server* calls which are done in Riak. Table1 shows the 15 most time-consuming *gen_server* calls in Riak.There are two rows for each function: *Mean_T* is the average time that each *gen_server:call* takes to be completed in microseconds and *Mean_N* shows the number of times that a function is called during the 5 minutes benchmarking. We see that the average time for function *call* from module *rpc* increases proportionally when the size of cluster grows (marked by yellow bar in Table1).

In next step, we just focus on RPC calls in Riak to see which RPC call is the bottleneck. Table2 presents all the RPC calls in Riak. There are two rows for each RPC function in Table2, the same we had for *gen_server* calls,: Mean_T shows the average time that each *rpc:call* takes to be completed in microseconds and *Mean_N* is the number of times that a function is called during the 5 minutes benchmarking. The results reveal that the latency of RPC call on function *start_put_fsm* grows when size of cluster increases (marked by yellow bar in Table2). For Example, on 24-node cluster, each RPC call takes 13,461 microseconds, and on 16, 8 and 4-node cluster, each RPC call takes 8,342, 2,284 and 2,019 microseconds respectively.

Module	Function		Runtime on 24-node cluster	Runtime on 16-node cluster	Runtime on 8-node cluster	Runtime on 4-node cluster
file_server_2	list_dir	Mean_T	10711	7265	3619	5210
		Mean_N	3473	5137	5086	5127
file_server_2	del_dir	Mean_T	12041	6208	2790	3395
		Mean_N	563	845	843	851
file_server_2	rename	Mean_T	9186	5450	2568	4261
		Mean_N	1689	2535	2529	2553
riak_core_vnode_manager	all_vnodes	Mean_T	74421	75983	85940	88814
		Mean_N	1832	2023	1926	1810
file_server_2	write_file	Mean_T	6213	3428	1831	4996
		Mean_N	1218	1754	1711	1720
riak_core_ring_manager	set_my_ring	Mean_T	58205	30895	28694	26874
		Mean_N	1	1	1	1
file_server_2	read_file_info	Mean_T	4357	2768	1408	2380
		Mean_N	4662	6890	6827	6883
file_server_2	open	Mean_T	3853	2559	1185	1973
		Mean_N	1130	1694	1690	1706
riak_core_stats_sup	start_child	Mean_T	16598	16933	16614	13816
		Mean_N	3	3	3	3
rpc	call	Average	15180	8547	2228	1953
		Mean_N	31104	49088	77468	106591
riak_core_capability	register	Mean_T	13587	12783	13311	14321
		Mean_N	7	7	7	7
lager_handler_watcher_sup	start_child	Mean_T	8736	5501	6919	6002
		Mean_N	4	4	4	4
net_kernel	connect	Mean_T	10109	14303	11854	24613
		Mean_N	15	6	6	3
riak_core_ring_manager	ring_trans	Mean_T	5532	1844	639	527
		Mean_N	1355	1667	1476	1476
riak_kv_js_sup	start_child	Mean_T	5388	4562	4745	4648
		Mean_N	16	16	16	16

Table 1: The 15 most time-consuming Riak gen_server calls

Module	Function		Runtime on 24-node cluster	Runtime on 16-node cluster	Runtime on 8-node cluster	Runtime on24-node cluster
riak_kv_put_fsm_sup	start_put_fsm	Mean_T	13461	8342	2284	2019
		Mean_N	33898	48689	82070	104973
riak_core_ring_manager	get_my_ring	Mean_T	1080	1037	813	1045
		Mean_N	1	1	1	1
riak_core_ring_manager	get_raw_ring	Mean_T	910	1032	983	889
		Mean_N	1	1	1	1
riak_core_gossip	legacy_gossip	Mean_T	676	739	650	794
		Mean_N	1	1	1	1
riak_core_capability	get (local)	Mean_T	4	4	3	3
		Mean_N	68692	96782	164178	210001

Table 2: All RPC calls in Riak

	24-node cluster	16-node cluster	8-node cluster	4-node cluster
start_put_fsm call-time	228	224	219	233
Latency of RPC	1896	2412	9951	14771

Table 3: *start_put_fsm* Runtime

Finally, we measure the call-time of function *start_put_fsm* to see whether it grows when the cluster size increases or not. We see from table3 that the call-time of function *start_put_fsm* doesn't increase when cluster size increase. Thus, we can conclude that the growth in the latency of RPC call for function *start_put_fsm* is not due to the function call-time locally. It seems that latency of RPC calls increase when the cluster size grows. To prove this, in the next section we will benchmark the scalability of *spawn* and *RPC* to see where *RPC* is a bottleneck for Erlang and hence Riak scalability.

Benchmarking the Scalability of RPC

To find the scalability limits of RPC, we employed *DEbench* [25]. *DEbench* is a benchmarking tool to investigate the scalability limits of distributed Erlang commands. We used *RPC* and *spawn/receive* commands. In the following section, we briefly explain each command [3]:

- *RPC*: a point-to-point command which executes a function on a remote node synchronously.
- *spawn*: a point-to-point command which executes a function on a remote node asynchronously. Since *spawn* is a asynchronous command, we use receive to get the result from the remote node.

Figure 14 shows the benchmark doesn't scale beyond ≈50 nodes. To find the bottleneck we measure the latency of commands individually and

we see from Figures 15 that the latency of RPC calls increases when cluster size grows. Since RPC is implemented with *rex* gen_server, this process can become a bottleneck for large scale systems, because all RPC requests that get to a node, should be handled by this single process.

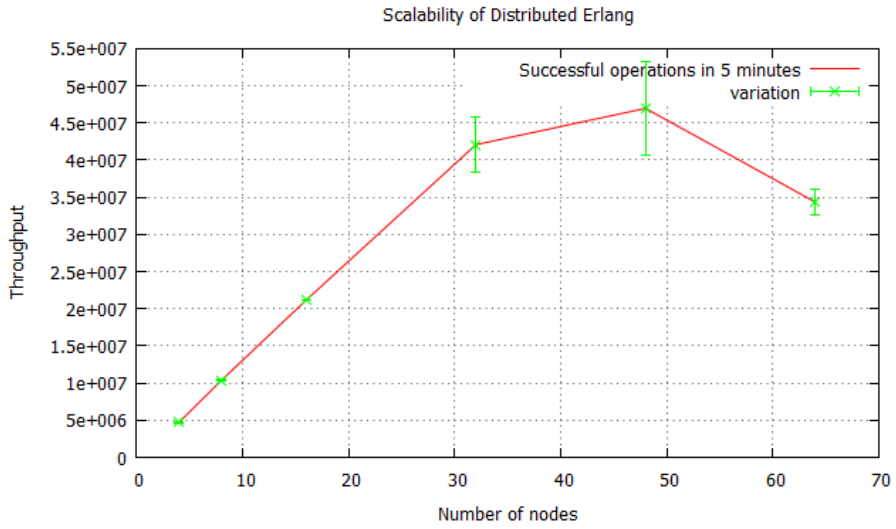

Figure 14: Scalability of Distributed Erlang

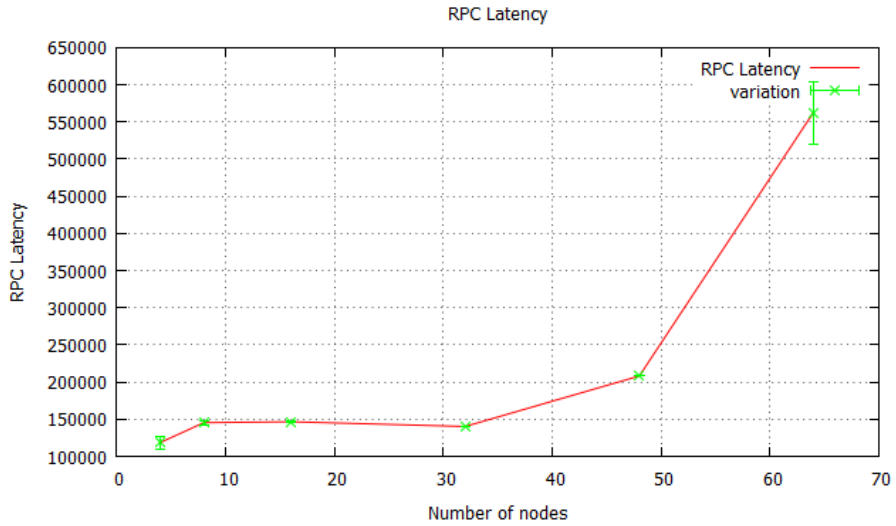

Figure 15: Latency of RPC calls for different different cluster size

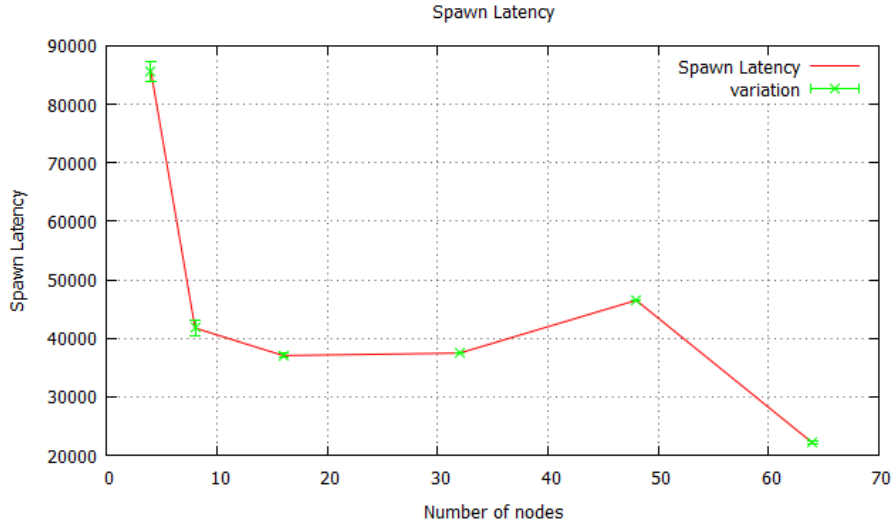

Figure 16: Latency of spawn calls on different cluster size

Discussion

Profiling the Riak source code reveals that the latency of RPC call on function *start_put_fsm* increases when size of cluster grows (marked by yellow bar in Table2). The results from benchmarking the scalability of *RPC* or *spawn* shows that *RPC* can a bottleneck for the Riak scalability because the latency of *RPC* calls increase dramatically when cluster size grows (Figures 15).

4.5 Availability and Elasticity

Distributed database systems must maintain availability despite network or node failures. Another important feature of distributed systems is Elasticity. Elasticity means shifting the system's load to the new nodes evenly when new resource (i.e. Riak node) are added to a running system [26]. So, a good elasticity means Riak cluster should show a better performance when new nodes are added. For benchmarking the availability and elasticity of Riak, we run 7 generators on a 20-node Riak cluster. During the benchmark, the number of generator remains constant (7 nodes) but the number of Riak nodes changes. As Figure 17 demonstrates, during the first 30 minutes of the benchmark, the Riak cluster has 20 nodes. We choose 30 minutes because we want to be sure that the cluster is in a stable condition. After the 30th minute, every two minutes, one node goes down, and in total 9 nodes go down until the 48th minute. Fails of 9 nodes is roughly 50% of all the nodes which is a considerable amount of failures. At the 48th minute, the cluster has 11 nodes and from the 48th to the 78th minute, the number of nodes

remains constant (11 nodes). After the 78th minute, in each two minutes one node comes back to the cluster. So, at the 96th minute the cluster obtains all 20 nodes again. From the 96th minute, the benchmark is run on a 20-node cluster for another 30 minutes (from the 96th to the 126th minute).

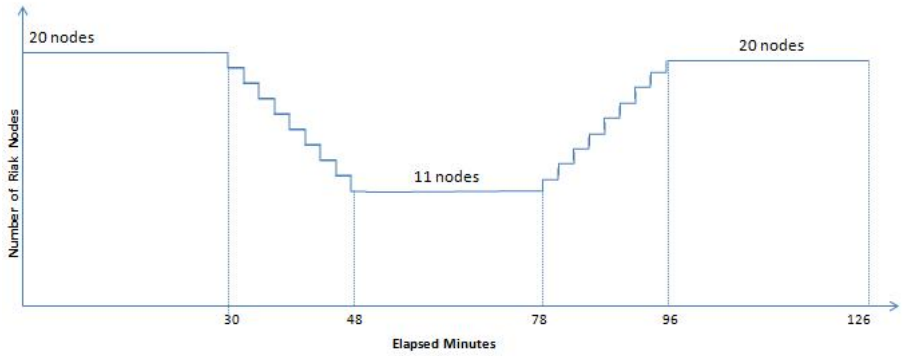

Figure 17: Availability and Elasticity Time-line

Figure 18 shows that when the cluster has lost some nodes (between 30th to 48th minutes), Riak throughput has degraded and the number of failures grows. The number of failures (in the worst case 37 failures occur in 180 seconds) in comparison with the number of successful operations (3.41 million operations in the same time) shows a good degree of fault tolerance. Between 48th and 78th minutes, the throughput hasn't changed hugely. During and after adding the new nodes (78th minute and onwards), the throughput has grown that shows an acceptable degree of elasticity.

4.6 Summary

The benchmark results are as follow:

1. The most important observation from Section 4.3 is that Riak scales up to approximately 60 nodes linearly on the Kalkyl cluster (Figure 6). But throughput does not scale beyond 60 nodes and with more nodes timeout errors emerge (Figure 7).

   We investigate the reason of scalability limits by profiling:

   - The results of profiling *CPU*, *RAM*, and *Disk* reveal that these can't be bottleneck for Riak scalability because they are mostly free and available.
   - Network profiling shows that the number of retransmitted packets (200 packets) is negligible in comparison to the total number of successfully transmitted packets ($\approx 5 * 10^8$ packets)

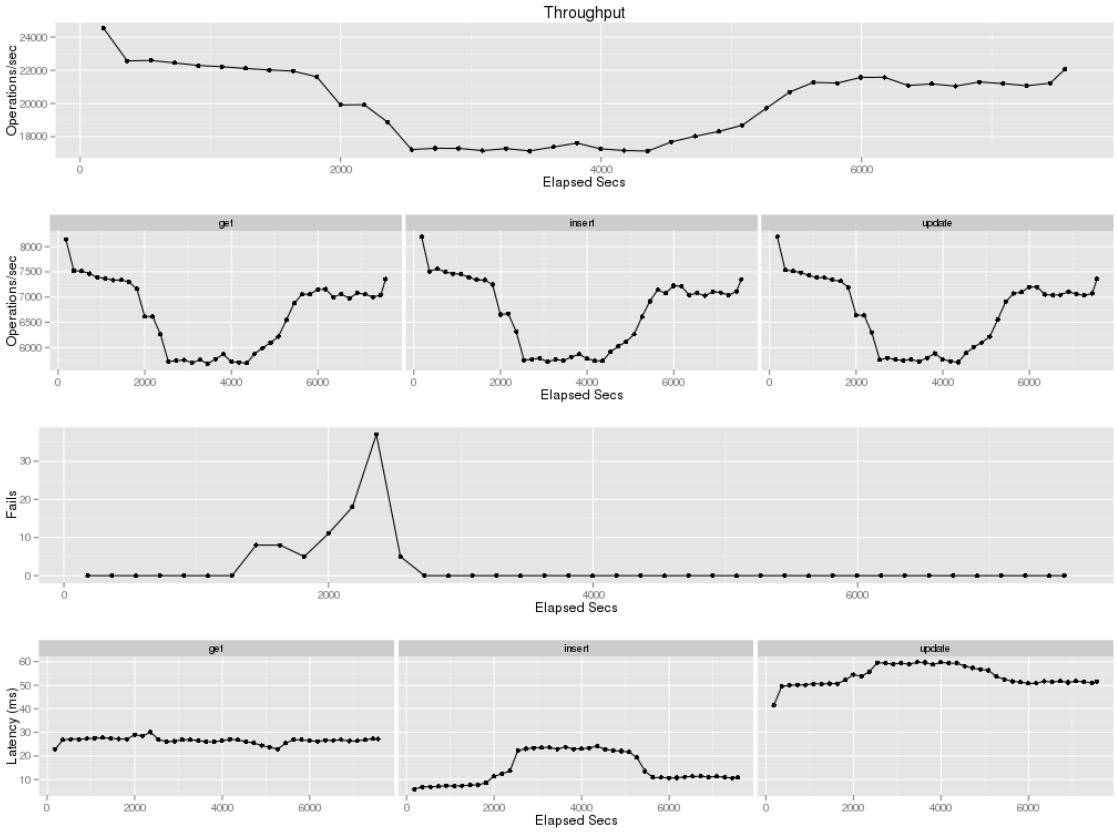

Figure 18: Throughput and Latency in the Availability and Elasticity Benchmark

Also, a comparison between network traffic of generators and Riak nodes (Figures 12 and 13) reveals that Riak nodes produce 5 times more network traffic than generator nodes. It could be because of replicating data over the Riak nodes.

2. In the availability and elasticity benchmark (Section 4.5), when the cluster has lost some of the Riak nodes (9 nodes), we see some failures (the maximum number of failures is 37), but the number of failures in comparison with the number of successful operations ($\approx$ 3.41 million) is negligible. When failed nodes come back up, the throughput has grown which means Riak has a good elasticity.

5 Conclusion and Future Work

We identified the requirements for scalable persistent storage in terms of data fragmentation, replication, availability and query mechanism. We evaluate some popular NoSQL DBMSs for Erlang applications against these requirements and we concluded that Dynamo-style DBMSs like Riak and Cassandra meet these requirements.

Our practical investigation reveals that Riak doesn't scale up beyond $\approx$60 nodes (Figure 6). Profiling the Riak source code reveals that RPC is a bottleneck for Riak scalability (Table2). The results from benchmarking the scalability of *RPC* or *spawn* reveals that the latency of *RPC* calls increase dramatically when cluster size grows (Figures 15). We think *RPC* calls limit the scalability of distributed Erlang, and hence the scalability of Riak.

The availability benchmark shows that Riak provides an available and fault-tolerant service (Figure 18).

Acknowledgements

This work has been supported by the European Union grant IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software'

The computations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) under RELEASE Project.

References

[1] André B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the second international workshop on Software and performance*, pages 195–203, New York, New York, USA, 2000. ACM Press. ISBN 158113195X.

[2] EricssonAB_Introduction. Who uses Erlang for product development? URL http://www.erlang.org/faq/introduction.html#id49882.

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. 1 edition, 2007.

[4] Olivier Boudeville and Francesco Cesarini. RELEASE: a high-level paradigm for reliable large-scale server software. In *Proceedings of the Symposium on Trends in Functional Programming*, St Andrews, UK, 2012. URL http://www.release-project.eu/documents/release-tfp12.pdf.

[5] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 3 edition, 2011.

[6] Giuseppe Decandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo : Amazon's Highly Available Key-value Store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, USA, 2007. ISBN 9781595935915.

[7] David Kargerl, Tom Leightonl, and Daniel Lewinl. 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*, pages 654–663, New York, USA, 1997.

[8] Thomas M. Connolly and Carolyn E. Begg. *Database Systems: A Practical Approach to Design, Implementation and Management*. 4 edition, 2004.

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

[10] Werner Vogels. Eventually Consistent. *Queue - Scalable Web Services*, 6(6):14–19, 2008.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 6:10–10, 2004.

[12] F A Y Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable : A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2), 2008. doi: 10.1145/1365815.1365816.

[13] Avinash Lakshman. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing*, 1(2):1277–1288, 2008. doi: 10.1145/1807128.1807152.

[15] EricssonAB_Mnesia. Mnesia. URL http://www.erlang.org/doc/man/mnesia.html.

[16] Joe Lennon. Exploring CouchDB, 2009. URL http://www.ibm.com/developerworks/web/library/os-couchdb/.

[17] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, 1 edition, 2010.

[18] BashoConcepts. Concepts, 2013. URL http://docs.basho.com/riak/latest/references/appendices/concepts/.

[19] BashoLevelDB. LevelDB, 2013. URL http://docs.basho.com/riak/latest/tutorials/choosing-a-backend/LevelDB/.

[20] Apache Cassandra. Apache Cassandra, 2013. URL http://cassandra.apache.org/.

[21] BashoBench. Basho Bench, 2013. URL http://docs.basho.com/riak/latest/cookbooks/Benchmarking/.

[22] SNIC-UPPMAX. The Kalkyl Cluster, 2012. URL http://www.uppmax.uu.se/the-kalkyl-cluster.

[23] Elie Krevat, Vijay Vasudevan, Amar Phanishayee, David G. Andersen, Gregory R. Ganger, Garth a. Gibson, and Srinivasan Seshan. On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems. In *Proceedings of the 2nd international workshop on Petascale data storage held in conjunction with Supercomputing*, page 1, New York, USA, 2007. ACM Press. ISBN 9781595938992. doi: 10.1145/1374596.1374598.

[24] Scott Lystig Fritchie. TCP incast: What is it? How can it affect Erlang applications?, 2012. URL http://www.snookles.com/slf-blog/2012/01/05/tcp-incast-what-is-it/.

[25] Amir Ghaffari. DEbench, 2013. URL https://github.com/amirghaffari/DEbench.

[26] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010. doi: 10.1145/1807128.1807152.

[27] BashoBitcask. Bitcask. 2013. URL http://docs.basho.com/riak/1.2.0/tutorials/choosing-a-backend/Bitcask/.

Appendix

A Comparing Riak Client Interfaces

There are two ways for communicating with a Riak node: *HTTP interface* and *Protocol Buffers interface*. Figure 19 compares both of theses interfaces on 20-node cluster with 7 generators. We run this benchmark three times and Figure 19 represents the median of three executions. The result shows that Protocol Buffers is approximately 2 times faster than HTTP. Protocol Buffers is a binary protocol and in comparison with HTTP, request and response messages are more compact. On the other hand, the HTTP protocol is more feature-complete, i.e. setting bucket property and support secure connections (HTTPS). HTTP is also more familiar to developers and simpler for debugging. Moreover, HTTP is web-based protocol and clients are able to connect to the Riak nodes through firewalls and proxy servers.

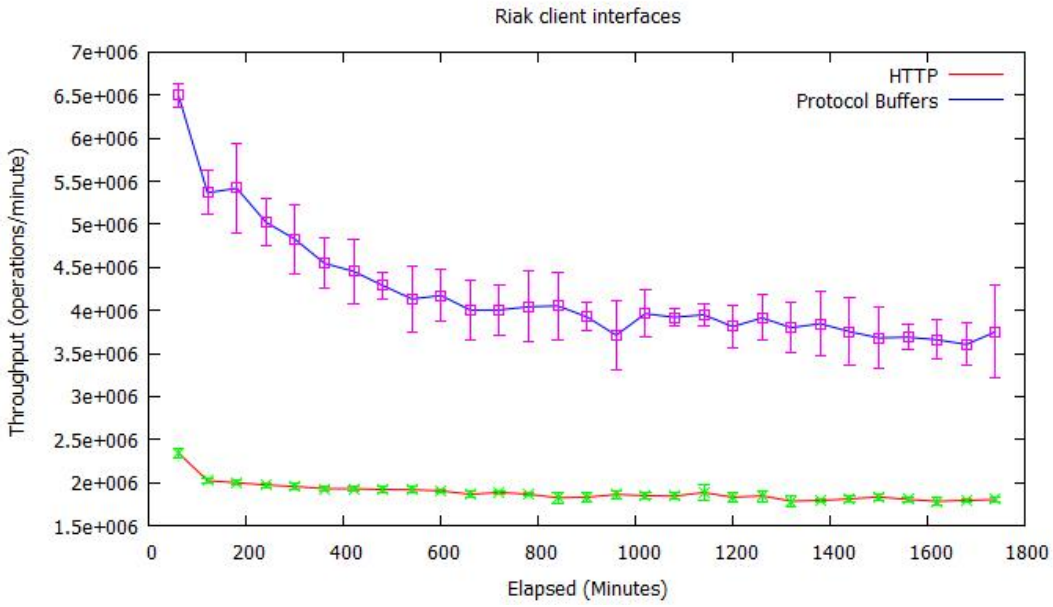

Figure 19: Comparing Riak Client Interfaces (20 nodes, 7 generators)

B Comparing Bitcask with LevelDB

In Riak, storage backends are pluggable and this allows us to choose storage engine that suits our requirement. *Bitcask* and *LevelDB* are two common persistent backends in Riak. Figure 20 compares them on 20-node cluster with 7 generators and shows that LevelDB throughput is 1.1 more than Bitcask throughput. In sake of accuracy, the benchmark is run three times

and Figure 20 represents the median of three executions.

Bitcask keeps all keys in memory and a direct lookup from an in-memory hash table point directly to locations on disk where the data lives and just one seek on disk is needed to retrieve any value [27]. On the other hand, keeping all keys in memory could be a limitation because system must have enough memory to host all the keys. In this benchmark the keys are integer value, so they are not too large to be accommodated in RAM. Thus, LevelDB is considered for systems with large number of keys.

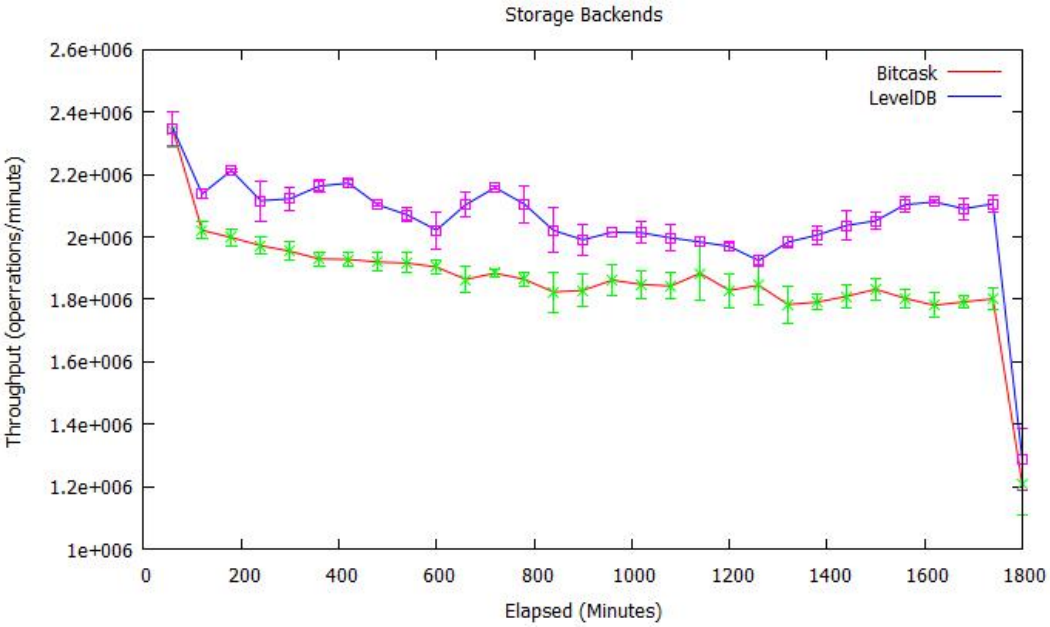

Figure 20: Comparing Riak storage backends (20 nodes, 7 generators)