Master's Thesis

Adopting Random Slicing for Riak Core

Die Übernahme von Random Slicing für Riak Core

Pascal Grosch
p_grosch14@cs.uni-kl.de

January 28, 2021

TU Kaiserslautern
Department of Computer Science
Prof. Dr. Ralf Hinze
Dr. rer. nat. Annette Bieniusa
Albert Schimpf, M.Sc.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Adopting Random Slicing for Riak Core" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, 28.01.2021
Ort, Datum


Unterschrift

Contents

1. Introduction **1**
- 1.1. Riak Core . . . 2
- 1.2. Scope . . . 2
- 1.3. Outline . . . 2

2. Related Work **4**
- 2.1. Data Partitioning Algorithms . . . 4
  - 2.1.1. Consistent Hashing . . . 4
- 2.2. Replication Placement Strategies . . . 5
  - 2.2.1. Adaptive Data Replication Strategy . . . 5
  - 2.2.2. Chain Replication . . . 5
  - 2.2.3. Redundant Share . . . 6
- 2.3. Distributed Key-Value Stores . . . 7
  - 2.3.1. Dynamo . . . 7
  - 2.3.2. Cassandra . . . 8
  - 2.3.3. Hibari DB . . . 8

3. Background **10**
- 3.1. Consistent Hashing in Riak Core Lite . . . 10
- 3.2. Riak Core Lite . . . 10
  - 3.2.1. System Components . . . 12
  - 3.2.2. System Procedures . . . 13

4. Random Slicing **16**
- 4.1. Partitioning Algorithm . . . 16
- 4.2. Replication Placement Strategy . . . 19
  - 4.2.1. Random Replication . . . 21
  - 4.2.2. Ring Rotation . . . 23
  - 4.2.3. Ring Jumping . . . 26

5. Riak Core Lite and Random Slicing **29**
- 5.1. Comparing Consistent Hashing and Random Slicing . . . 29
- 5.2. Changed Components . . . 29
  - 5.2.1. Ring . . . 29
  - 5.2.2. Navigation . . . 30
  - 5.2.3. Claim . . . 30
- 5.3. Changed System Procedures . . . 31
  - 5.3.1. Leave Cluster . . . 31

5.3.2. Navigate the Ring . . . 31
5.3.3. Resize Ring . . . 31
5.3.4. Handoff . . . 31

6. Evaluation **33**
6.1. Hypotheses . . . 33
6.2. Setup . . . 34
6.2.1. Tools . . . 34
6.2.2. Configurations . . . 35
6.3. Metrics . . . 36
6.4. Results . . . 36
6.4.1. Common Anomalies . . . 37
6.4.2. Consistent Hashing . . . 38
6.4.3. Random Slicing Jumping . . . 40
6.4.4. Random Slicing Random . . . 42
6.4.5. Random Slicing Rotation . . . 44
6.4.6. Comparison . . . 45

7. Future Work **50**
7.1. Improve the Gap Collection Algorithm . . . 50
7.2. Reintegrate Handoff Optimizations . . . 50
7.3. Develop a More Refined replication placement strategy (RPS) . . . 51
7.4. Integrate the Replication Placement Strategy with the Preference List Creation . . . 51
7.5. Support Heterogeneous Nodes . . . 51
7.6. Evaluate Higher Scalability . . . 52

8. Conclusion **53**

A. Actions **58**
A.1. General Actions . . . 58
A.2. Start Cluster . . . 71
A.3. Join Cluster . . . 79
A.4. Leave Cluster . . . 81
A.5. Remove Node . . . 82
A.6. Navigate via riak_core_ring . . . 82
A.7. Navigate via chashbin . . . 84
A.8. Navigate via riak_core_apl . . . 87
A.9. Resize Ring . . . 92
A.10.Handoff . . . 93

B. Evaluation Data **94**

Acronyms

ADRS Adaptive Data Replication Strategy

RCL Riak Core Lite

RPS replication placement strategy

RS Random Slicing

Glossary

cloud computing is the usage of a set of connected network computing devices called *cloud* to perform a shared task. The .

cluster is a collection of potentially distributed entities that store and process data.

node is an entity that can store and process data and communicate with other nodes. A node is called *physical node* if it is represented by one or more *virtual nodes*. A *virtual node* is part of exactly one physical node .

section is a partition of a space. Sections are ordered by their starting index.

Abstract

Riak Core is a framework to build distributed systems based on the Dynamo architecture. It uses Consistent Hashing to map keys to nodes as well as to determine the placement of replicas of those keys. The usage of Consistent Hashing brings some drawbacks like a fixed ring size leading to reduced scalability. An alternative approach to map keys to nodes is Random Slicing.
We analyze the current state of Riak Core Lite and how Random Slicing can be integrated. Based on those findings we implement a prototype variant of Riak Core Lite with Random Slicing and conduct an evaluation via benchmarks to compare the variants. We show that even in a prototype implementation Random Slicing performs on the same levels as Consistent Hashing. As this shows that Random Slicing is a promising alternative to Consistent Hashing we propose further changes to Riak Core Lite that can raise the performance above its current levels.

Zusammenfassung

Riak Core ist ein Framework zur Erstellung verteilter Systeme, die auf der Dynamo Architektur basieren. Es benutzt Consistent Hashing, um sowohl Schlüssel auf Knoten abzubilden, als auch zur Bestimmung der Replikationsplatzierung dieser Knoten. Die Nutzung von Consistent Hashing bringt einige Nachteile wie eine feste Ringgröße mit sich, die zu reduzierter Skalierbarkeit führen. Eine alternative Methode, um Schlüssel auf Knoten abzubilden ist Random Slicing.
Wir analysieren Riak Core auf dem momentanen Stand und wie Random Slicing integriert werden kann. Auf der Grundlage dieser Ergebnisse implementieren wir eine Prototypvariante von Riak Core Lite mit Random Slicing und führen eine Auswertung mithilfe von Benchmarks durch, um die Varianten zu vergleichen. Wir zeigen, dass schon in einer Prototyp-Implementierung Random Slicing Leistungen auf dem gleichen Niveau wie Consistent Hashing zeigt. Da dies zeigt, dass Random Slicing eine vielversprechende Alternative zu Consistent Hashing ist, schlagen wir weitere Änderungen an Riak Core Lite vor, die die Leistung über das momentane Level heben können.

1. Introduction

As cloud computing[22] becomes more and more prevalent in a wide array of domains, the reliance on the quality of service guarantees for relevant tasks in many industries is growing. Typical tasks for a cloud include storing data as a scalable storage or solving a task remotely and delivering the results. Obviously, a cloud still performs its tasks on hardware. However, since a core quality of cloud systems is their scalability, they are often implemented as distributed systems with a dynamic cluster configuration. For example a company uses a cloud system both for storing its data as well as computing the results of data processing. In this scenario the storage requirements may grow over time and therefore the cloud storage needs to be scaled. Another aspect is that while the requirements on processing power for day-to-day data processing might not fluctuate, processing large amounts of data for an annual report represents a spike in necessary computing power. If the cloud provider wants to guarantee a fast processing time, the cloud needs to be scaled out. To this end, more machines are added to the cloud configuration and the incoming tasks are distributed to all machines.

One problem with distributed computing is how to distribute data or tasks on a number of nodes such that each node has approximately the same load to optimize the scalability. If this problem is not handled properly, some nodes will be overloaded while others are running idle. This leads to a worse response time and makes it unlikely that adding more machines will fix the problem. One approach is using a central load balancer[12] that tracks incoming requests and fairly distributes them to all nodes. However, a decentralized approach to solve this problem in a pure peer-to-peer setting does also exist. Developed by Amazon, the Dynamo[6] architecture is used as the backend for the Dynamo DB. It uses a variant of Consistent Hashing[13] to balance the load on each node in the cluster. In a simplified view, Dynamo places the nodes on a ring representing a hash space and uses hashing to map keys to nodes. We describe the workings of Dynamo in more detail in Section 2.3.1. The Riak key-value store and Riak Core are open-source implementations of Dynamo DB and Dynamo respectively.

One of Riak's former developers, Scott Fritchie, identifies some problems with the implementation of Consintent Hashing in Riak Core[8]. He proposes a different algorithm called Random Slicing (RS)[16] as an alternative data partitioning algorithm. The goal of this thesis is to replace the existing Consistent Hashing implementation with an implementation of RS. Unlike Consistent Hashing, RS does not allow for an inherent RPS strategy. Therefore an important part of this thesis is the development of possible RPSs. We will conduct an evaluation on the performance differences between the two variants and reason about the advantages of RS over Consistent Hashing in practice.

1.1. Riak Core

Riak[1] was developed by Basho Technologies as an open source implementation of the Dynamo architecture[6]. The initial Riak key-value store is a distributed, decentralized NoSQL database on which nodes form a cluster and communicate with each other. From the backend of this database the Riak Core[2] framework was developed. It enables to build distributed systems while already taking responsibility for some core tasks. The most notable capacity is the automatic formation of nodes to a cluster and keeping nodes updated on the cluster state[17]. It also allows to distribute workload to the nodes in a fair manner. To this end it uses a variant of Consistent Hashing[13] like the one Dynamo uses. Instead of directly storing key-value pairs like its database origin it uses the hashing of the key to compute a number of owner nodes of that key. This can for instance also be used to distribute tasks assigned to the key to a number of nodes. In this thesis we use a more simplified and streamlined version of Riak Core called Riak Core Lite (RCL). We give a more detailed overview in Section 3.2.

1.2. Scope

A production-ready integration of RS with Riak Core requires a detailed analysis of the existing system. We need to list how each component is affected by the architectural decision of using Consistent Hashing as the partitioning and replication placement algorithm. From this analysis we need to reevaluate design decisions under the aspect that we use RS instead of Consistent Hashing. The new design decision need to be implemented in a way that the system functionality and performance are not noticeably negatively impacted.

However this process of redesigning the system can not be achieved within the time constraints of this thesis. Therefore we decide to create a prototype implementation as a proof of concept from the existing system with only minor design decisions. From this prototype we will draw conclusions on whether RS is a feasible approach of improving Riak Core. To this end we analyze the most basic functionalities and workflows of Riak Core and constrain the redesign on the parts of the system that are relevant to them. We evaluate the adapted system on the basis of known fixable problems and improvements and try to give a reasoning on how RS can be a better partitioning and replica placement strategy than Consistent Hashing when the system is properly redesigned.

1.3. Outline

First we will present work related to this thesis. This includes the original version of Consistent Hashing[13] as an example for data partitioning algorithms. In the realm of RPSs we present the Adaptive Data Replication Strategy[15] representing approaches using current system metrics, Chain Replication[21] as an example for approaches using virtual nodes, and Redundant Share[4] which is based on a mathematical model. We close the related work chapter with an overview of different decentralized distributed

key-value stores, namely Dynamo[6], Cassandra[20], and Hibari DB[10]. In the following background chapter we first present RCL's implementation of Consistent Hashing before describing the structure and functionalities of RCL in detail. After this we formally introduce the concept of RS and present some simple RPSs to enable it to replace Consistent Hashing. The main chapter describes how the structure and functionality of RCL is impacted when replacing its Consistent Hashing algorithm with RS. In the evaluation chapter we present our hypotheses on the impact of RS on RCL and how we plan to test them. We also present the data from the evaluation and show how the hypotheses are either supported or contradicted. As the scope of this thesis does not include a full redesign of the existing RCL framework but rather a prototype implementation as a proof of concept, there are many issues left open and possible improvements. We address those issues in the future work chapter and present some abstract or concrete ideas on how to approach these problems. Closing the thesis we give an overview on what we achieved and how feasible we assume the integration of RS with RCL is based on the prototype developed in the scope of this thesis.

2. Related Work

The goal of this work is to replace Riak Core's implementation of Consistent Hashing with RS. Therefore one section of this chapter will explore different data partitioning algorithms. Since Riak Core's RPS is closely related to Consistent Hashing and cannot be reused, another important aspect of related work are RPSs. Concluding the chapter is an overview of other distributed key-value stores and how they handle partitioning and replication.

2.1. Data Partitioning Algorithms

Data partitioning algorithms decide how the data that is to be stored is distributed on different nodes. A common approach is to use a key as the input of a hash function and apply the partitioning algorithm on the hash space.

2.1.1. Consistent Hashing

Consistent Hashing was introduced by Karger et al.[13] as a tool to allow for different views of a node cluster in which placement of objects is determined by hashing. Different implementations of Consistent Hashing can be found in distributed systems including RCL[6][14][9]. Karger et al. motivate the technique by explaining that clients usually do not see every node in a cluster. Therefore using an ordinary hash function would lead to different mappings of objects to nodes for different clients.

They define a construction of a family of hash functions that overcomes those problems. Virtual nodes and objects are randomly placed on the unit interval by independent and uniform random functions. The hash function then assigns an object to the nearest node on the unit interval.

Implementation-wise the authors propose using balanced binary trees storing a mapping of sections to nodes. To improve runtime the interval is divided into $\kappa C \log(C)$ equal length sections, where C is the upper bound of nodes and κ is some constant. Each node is also replicated $\kappa \log(C)$ times and mapped randomly to the unit interval. Each section is assigned its own search tree with a low expected lookup-time. To avoid shrinking the sections with a growing number of nodes their length is chosen as $\frac{1}{2^x} \leq \frac{1}{\kappa C \log(C)}$ where x is chosen as the maximum fulfilling the inequation. The sections are bisected with the growing number of nodes and are therefore already of half length once the next power of 2 is reached.

2.2. Replication Placement Strategies

RPSs decide on which nodes replicas of an object are stored. The strategies considered in this section are all able to take heterogeneity of nodes into account. Often a preference list containing nodes ranked by how suitable they are to store the replica is either directly created or can be derived as the result of applying the strategy.

2.2.1. Adaptive Data Replication Strategy

Adaptive Data Replication Strategy (ADRS)[15] is a replication management system proposed by Mansouri that uses multi-objective optimization to create a replication distribution that fulfills given requirements. The model considers m heterogeneous data nodes and n different data entries of equal size that are to be distributed and replicated on the data nodes. Furthermore the access rate of the data entries have a Poisson distribution and data entries are immutable.

The cost function of the optimization problem considers mean service time (MST), load variance (LV), storage usage (SU), failure probability (FP), and Latency (L) which are computed from different metrics and hardware specifications. The objectives of the optimization can be weighted with weights $w_1, ..., w_5$ and therefore the cost function is $c = w_1 \cdot \mathrm{MST} + w_2 \cdot \mathrm{LV} + w_3 \cdot \mathrm{SU} + w_4 \cdot \mathrm{FP} + w_5 \cdot \mathrm{L}$. The replica is stored on the data node with the minimal cost function.

ADRS also considers the free space of the data nodes. If a data entry cannot be placed on the node with minimal cost it replaces data points already stored on the node. To this end all existing data points on the node are evaluated by their number of accesses NA, availability P, time between the last access LA and current time CT, and data size S. This function is therefore

$$V = \frac{w_1 \cdot NA + w_2 \cdot P}{w_3 \cdot (CT - LA) + w_4 \cdot S}$$

and data entries are ordered ascending by this value. Files are then deleted in this order until the new data entry can be placed on the node.

2.2.2. Chain Replication

Chain Replication[21] is a replication-management protocol for storage services that offers strong consistency guarantees. It offers a simple interface that supports queries and updates, of which writes are a special instance, on object IDs. The nodes in the cluster are assumed to be *fail-stop* which means failure is detected and nodes stop instead of continuing in an erroneous state.

A chain of length N consists of a head (virtual) node H, a tail (virtual) node T and $N - 2$ nodes S_i in between. In normal operation the update requests are sent to the head and queries are sent to and answered by the tail. The result of an update request is computed on the head node and the result is propagated along the chain. Queries are directly answered with the current replication of the object on the tail node.

To handle chain failures and reconfiguration a master server is used to update other members on the new configuration. If a node fails it is removed from the chain. In the case of the head its successor is assigned as the new head and all requests not yet forwarded are lost. A failure of the tail node is handled by assigning its predecessor the tail role without losing any information. The removal of any other nodes in the chain is handled by making its successor the successor of its predecessor. Additionally all requests forwarded to it by its predecessor are again forwarded to its successor before it resumes operation.

A new node is always added at the end of the chain and assumes the tail role. The old tail forwards all replications to the new node. Queries to the old tail are forwarded until the clients are informed of the new configuration. The distribution of nodes to chains is not handled by this protocol and left to the implementing system. Example setups can be found in an analysis of Hibari[9] (Section 2.3.3).

2.2.3. Redundant Share

Redundant Share[4] is a storage virtualization that supports heterogeneous clusters while balancing loads. It distributes m data blocks of uniform size to n nodes of heterogeneous capacity $b_0, ..., b_{n-1}$ with k redundant copies. The k copies of the data blocks are guaranteed to be stored on k different nodes.

The algorithm iterates over the nodes ordered descending by their capacity. For each node a random process decides if it stores the primary copy of the data block. The probability that node i stores the primary copy is computed under the assumption that the other copies are fairly distributed to nodes $i+1, ...n$:

$$\check{c}_i = \frac{2 \cdot b_i}{\sum_{j=i}^{n-1} b_j}$$

This probability is then compared to the result of the random value generated from the address of the data block and the node identifier, for example by applying Consistent Hashing. While there are more than 2 copies to be distributed the algorithm is called recursively. When there are only 2 copies left one is again assigned as a primary copy to node i as before and the last copy is then placed with a fair distribution strategy in node $i+1$, ... node $n-1$. In the special case that $\check{c}_i < 1$ and $\check{c}_{i+1} > 1$ the capacity b_{i+1} is adjusted to a capacity b^* that is ensures the node still receives the correct load.

To reduce the runtime from $\mathcal{O}(n)$ to $\mathcal{O}(k)$ in a trade-off with space the probability for the first copy is computed with a single hash function $p_i = \check{c}_i \cdot \prod_{j=0}^{i-1}(1-\check{c}_j)$ which is scaled to $\sum_{i=0}^{n-1} p_i = 1$. For every other copy $\mathcal{O}(n)$ hash functions are used. The hash function is chosen by which node i is chosen in the previous step and computes the nodes out of $i+1, .., n-1$ for the next copy.

2.3. Distributed Key-Value Stores

RCL originally stems from the Riak distributed key-value store. Since RCL implements core functionalities of Riak, we take a look into the workings of other distributed key-value stores. A special focus lies on how the data space is partitioned and mapped to physical nodes, replications are distributed among the nodes, and how those replicas are retrieved.

2.3.1. Dynamo

Dynamo[6] is an eventually consistent key-value store developed by Amazon and focuses on high availability and scalability. The system is highly decentralized and its interface only considers a single primary key. Both partitioning and replication of data is handled by an implementation of Consistent Hashing[13] (see Section 2.1.1).

Assumptions and requirements of Dynamo are the following:

- Queries only consist of reads and writes on a single binary object that are smaller than 1 MB.
- Dynamo only provides a weaker consistency in trade for high availability and no isolation guarantees at all.
- The environment Dynamo operates in is non-hostile and there are no security requirements.

Dynamo's implementation of Consistent Hashing treats the hash space of a hash function as a circular ring where the smallest and biggest value are neighbors. To determine which node is responsible for a key, Q virtual nodes are assigned at equally distributed positions on the ring. The same number of virtual nodes belong to each physical node. A node is responsible for the range between its position and the position of its predecessor which in turn defines Q same size sections. When the hash function is applied to a key the responsible node is identified via the next greater position of a node on the ring. When a node leaves the system, its sections are distributed equally to the remaining nodes. When a node enters the system, it takes sections from the other nodes until they are equally distributed again.

Dynamo's replication functionality makes use of its implementation of Consistent Hashing. The system is configured to replicate each data entry at N physical nodes. The node responsible for the replication is the one first responsible for the key according to the partitioning algorithm. Using the ring structure the next $N-1$ sections whose owning physical node is not already in the preference list are used as storage nodes for the replicas. Those nodes are called *preference list* and usually contain more than N nodes to make it more robust against failures. Therefore a *put*-operation hashes the given key to the coordinator node and places the data to the first N healthy nodes in the preference list. Failed nodes are skipped but are marked with a *hinted handoff* to ensure that they finally receive the replica in case of a recovery. Consistency is provided with the help of data versioning via vector-clocks per data entry. In a quorum-like manner the minimum

number of successful reads R and writes W can be configured. If less than $W - 1$ nodes respond to the replication call, the *put*-operation fails. Analogous, a *get*-operation determines the coordinator node and queries the preference list for R replicas. If it detects differing versions by comparing vector clocks all versions are returned.

2.3.2. Cassandra

Cassandra[14] is an eventually consistent key-value store developed and used by Facebook and is nowadays maintained as an open-source project by Apache[20]. It is a decentralized distributed structured storage system that was developed with strict requirements on performance, reliability and efficiency in mind. It especially focuses on a high scalability. Failures are expected to occur regularly instead of only occasional. Its data model is a key-value model where the key can be of arbitrary length and the value is the row of a multi dimensional table.

The partitioning algorithm is quite similar to Dynamo's implementation of Consistent Hashing(see Section 2.3.1). Instead of just distributing nodes evenly on the ring Cassandra in addition keeps load information and moves nodes with free capacities on the ring to balance loads similar to Chord[19]. At the start of a cluster a node chooses a random position on the ring and acts as a seed. When a new node enters it is placed such that it splits the section of a node under heavy load. That node is then responsible to transfer the affected data to the new node. A failure of a node is treated as transient and no action is taken to avoid unnecessary re-balancing and section assignment.

Since the partitioning algorithm is close to Dynamo's the same is true for the replication strategy. N is configurable as the number of replications. In addition to the simple setting "Rack-Unaware" that uses the $N - 1$ successor nodes on the ring as the preference list there are two settings called "Rack Aware" and "Datacenter Aware". For those settings the systems adapts a more centralized approach as one node is elected as a leader using *Zookeeper*[11]. This leader is polled by other nodes to get an assignment of a maximum of $N - 1$ ranges they are responsible for. In contrast to Dynamo there is no mention of minimal reads R or writes W.

2.3.3. Hibari DB

Hibari DB[10] is a strongly consistent key-value store developed by Gemini Mobile Technologies/Cloudian and left to be maintained by the open-source community. In addition to the application documentation details about the system are described in an analysis by S.L. Fritchie[9]. Its strengths are fault tolerance, high availability and the ability to update multiple keys in a single operation. It is conceptualized for a single data center and does not handle wide area network latency well. An admin server is used to monitor, repair, and configure the cluster. To partition the data to the physical nodes a variant of Consistent Hashing is used. Based on this partitioning a chain replication (see Section 2.2.2) algorithm distributes the replicas.

In Hibari's implementation of Consistend Hashing MD5 is used as the hash algorithm. The hash space is mapped to the unit interval $[0, 1)$ which is divided into an arbitrary

number of sections. Each section represents a chain and a single chain can own multiple sections. The size of the sections is determined by a relative weight assigned to the chain which represents the capacities of the physical nodes participating. When the structure of the cluster changes new relative weights are assigned and the sections are recalculated in a way that minimizes key migration. Keys no longer belonging to their former section need to migrate. To this end the old section map is kept until the key migration is done. During key migration the system is still operational. To achieve this the set of keys is separated into consecutive sweep windows of which one contains those keys that are currently migrating. Requests on keys in the currently active sweep window are deferred and forwarded to the correct chain when migration on that window is done.

Replication of the keys to physical nodes is handled by chain replication. The setup of the chains can be freely configured. The standard chain assignment uses the physical nodes in a ring and uses each node on the ring as a head of a chain and take the next $N-1$ nodes as the rest. Another possibility of building chains of length N is striping each node into N virutal nodes. N physical nodes are grouped together and the virtual nodes are distributed diagonally over the N chains such that each physical node acts as the same number of heads, tails, and middle parts each.

3. Background

This chapter contains the necessary information and descriptions of current circumstances to help understand the challenges and reasoning behind integrating RS with Riak Core in place of Consistent Hashing. First, we give a more detailed description of RCL's implementation of Consistent Hashing. The following section contains the technical description of RCL, a simplified variant of Riak Core.

3.1. Consistent Hashing in Riak Core Lite

As Riak Core is an open source implementation of the Dynamo architecture[6] its version of Consistent Hashing is quite similar to Amazon Dynamo's (see Section 2.3.1). At the core of the implementation is the ring representing a hash function's hash space. Here the function in use is SHA-1[7] which results in a hash space of $[0, 2^{160}) \subseteq \mathbb{N}$. Instead of pseudo-randomly distributing nodes on the ring and assigning keys to nodes by smallest distance the ring is split into n partitions of equal size. To ensure the equal size of all partitions the ring size n has to be set to a power of 2. The first index of each partition is an index of a virtual node. The virtual nodes are assigned to physical nodes by RCL's claim algorithm (see Section 3.2.1). Contrary to Amazon's implementation, the physical nodes are assumed to be homogeneous and get assigned the same number of vnodes if it is possible. To find the node responsible for a given key the key is hashed and the result is placed on the ring. The node owning the next virtual node index i is responsible for the key. The $(k+1)$th entry of the preference list for the key can be easily computed by looking at the node owning the index $i + k \cdot 2^{160}/n$.

Figure 3.1 shows an example of a key hashed to a Consistent Hashing ring and how the preference list is constructed. The ring has a size of 32 with 8 physical nodes. One can see, that the virtual nodes are responsible for the indices in counter-clockwise direction. As an example, assume the hash value of "key" is owned by node n6. Then the preference is computed by traversing all predecessor nodes on the ring. In this case the resulting preference list is [n6, n5, n7, n0, n1, n2, n3, n4, n5].

3.2. Riak Core Lite

Riak Core was extracted from the Riak key-value store to enable a simple creation of distributed decentralized systems. In an attempt to streamline and modularize the concept RCL[18] was created from Riak Core. One of the biggest motivations to use RCL as the system to integrate RS with is the simplified structure and shaved off features which leads to a lower potential for complications and errors while still upholding the basic

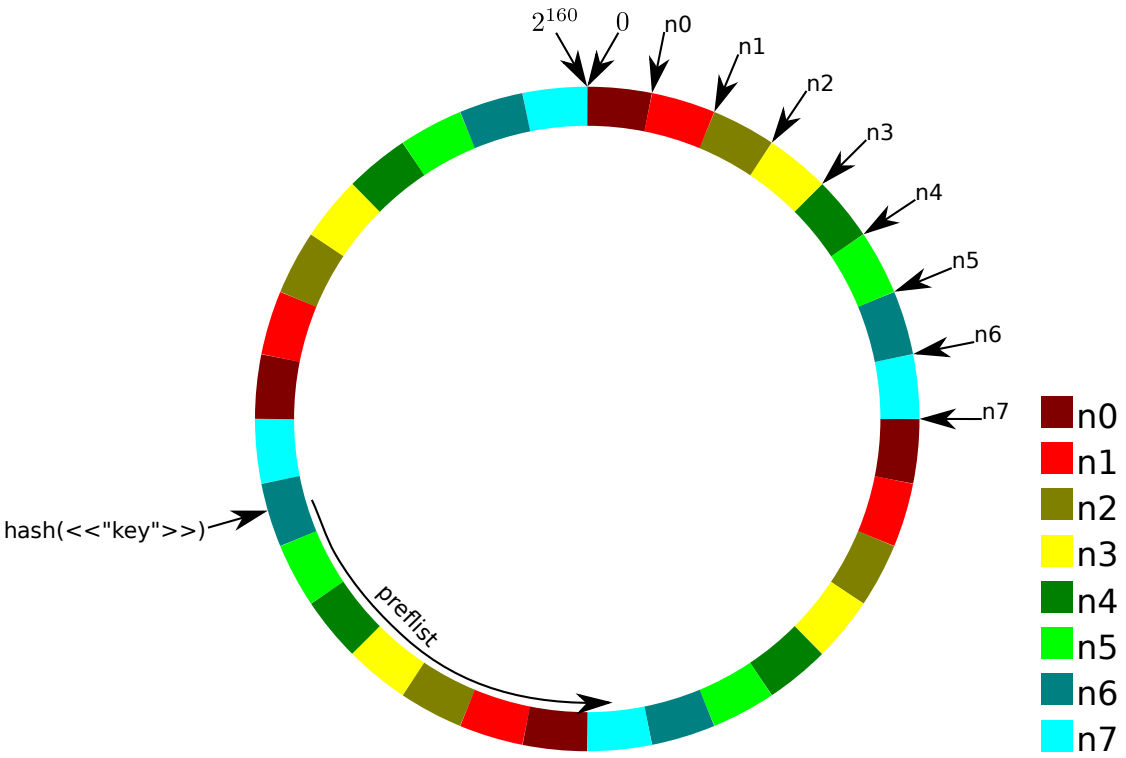


Figure 3.1.: Consistent Hashing Example. Ring with 32 partitions and 8 nodes.

principles of the Dynamo architecture. As RCL is rather a framework for distributed systems than a complete key-value store it's main functionality is the coordination of the nodes in the cluster and navigation of the Consistent Hashing ring. It does not handle the actual data transfer or replication and instead just returns a plan for the replication in form of a preference list. The following sections contain an overview of the basic system components and how they interact given some core use cases.

3.2.1. System Components

As the RCL System consists of over 40 erlang-modules this sections groups some modules into logical components. Here the descriptions of the components are on a more abstract level to keep the concepts simple but still enable a discussion of basic system functionalities in Section 3.2.2.

Ring

The ring is a central concept in the Dynamo architecture and Consistent Hashing. We discussed RCL's implementation of Consistent Hashing and the ring model this component is based on in Section 3.1. In a cluster every member has its current view of the ring. There is only one ring installed per instance, which is only accessible and mutable by this component. The ring enables an assignment of keys to nodes and to create the preference list via Consistent Hashing. To this end the ring guarantees that for each virtual node at least N predecessor virtual nodes are owned by distinct physical nodes. As node failures are assumed as transient in the Dynamo architecture downed nodes are not removed from the ring. Instead the Navigation component handles failed nodes by skipping them in the filtered preference list. An additional responsibility of the ring is to merge two divergent rings after a network partitioning is fixed.

Navigation

The navigation component enables a user to find the node responsible for a key and creating preference lists. It offers to query different kinds of filtered and annotated preference lists. For optimization purposes it offers a read-only binary encoding of the current ring with faster access time and lower memory consumption.

Claim

Claim handles the rebalancing of the load to nodes when the cluster changes. It is used when nodes join, leave or are recognized as down, and when the ring is resized. Therefore it is also the entry point of those actions through a stage-plan-commit life cycle in which several changes are staged, the resulting change plan is checked, and the changes are applied to the ring. The algorithm keeps the guarantees of the preference list and the ring intact. A central aspect of load balancing is to diagonalize the nodes on the ring instead of periodically repeating the same pattern to avoid unfinished patterns at the end of the ring.

VNode

In the context of RCL a virtual node represents the combination of the index of the partition on the ring and the owning physical node. A member can be a distinct physical machine or an instance on a physical machine. Those cases are not distinguished by RCL and have to be handle by the system administrator. In the component tasks are scheduled and requested at the responsible nodes in an attempt to avoid overloading. Those tasks include handling messages sent to the nodes and executing a data handoff to other nodes. The virtual node also acts as a behavior definition for those tasks and therefore as the entry point for the application logic when using RCL.

Handoff

The handoff implements the concept of transferring data a node previously owned to the node currently owning the keys the data belongs to. The handoff component is responsible for creating the connection, creating the correct handoff message and preparing the handoff state a node works on.

Gossip

The gossip protocol is responsible to communicate state updates to other cluster members via a random method that ensures eventual consistency. Each node gets assigned tokens periodically. As long as a node has tokens it sends the ring to randomly chosen other nodes. The gossip component also detects when a ring version changed and triggers the reconciliation in the ring component.

Eventhandler

The eventhandler component allows to install different eventhandlers like one that listens to ring updates. Eventhandlers are used by RCL internally, however they are also intended to help tailoring the system to the application logic by implementing custom ones. Several components are listening to ring update events to trigger procedures. This includes starting and registering vnodes or triggering the gossip protocol.

3.2.2. System Procedures

The system procedures we list here only show a subset consisting of the most basic operations to demonstrate functionality offered by RCL and build a ground for comparison to the system adaptions implemented in this thesis. Any procedure that changes the cluster state, e.g. member status or ring size, have to be applied by the claim's stage-plan-commit cycle which is not explicitly mentioned for those procedures. In this lifecycle a number of changes is staged via the claimant and premarked in the corresponding place. Before one can commit the changes, a plan has to be created (see Action A.10, Figure A.10) and checked if the results are the desired ones. When committing scheduled changes all changes are applied and the new ring is installed (see Action A.16, Figure A.16).

Start Cluster

When starting the cluster all system processes are started. This includes starting the ring management system, initializing all virtual nodes and starting the gossip protocol. When no information about an existing ring can be found on the machine starting the cluster it is assumed this is a new cluster. Otherwise the ring is installed and the members are queried. For details see Action A.14, Figure A.14.

Join Cluster

A node in a singleton cluster can join the cluster of another node. To this end the other node is queried for the remote ring and the membership status of the ring is computed locally. The ring with the updated members list then is sent via the gossip protocol. To actually make the new node an owner on the ring the changes have to be planned and committed. There is no handoff from the joining node to the existing cluster, only in the other direction. Thus, any keys already stored on the joining node are lost. For details see Action A.20, Figure A.20.

Leave Cluster

A node can leave the cluster with a call to it's local instance. This causes the status of the local node to be marked as leaving in the ring. The new member status is then gossiped to other members in the cluster. When the claim is executed the next time and the executing node has knowledge of the leaving status the node is not considered for ring ownership and removed. Leaving is the only ring change that is not applied via the stage-plan-commit life-cycle. For details see Action A.22, Figure A.22.

Remove Node

Removing a node from the cluster causes it to directly hand off all of its data to other nodes. This leads to a handoff of any sections owned by this node and marking the node invalid. The status change will lead to the node being excluded from the ring and marked invalid on the next reconciliation. For details see Action A.23, Figure A.23.

Navigate the Ring

There are several ways of executing navigation tasks in RCL. Using only the ring one can

- find the index of the node responsible for a given key (see Action A.24, Figure A.24),
- find all indices owned by the local node (see Action A.25, Figure A.25),
- find the node owning a given index (see Action A.26, Figure A.26),
- compute a simple preference list assuming all nodes are up for a given key (see Action A.27, Figure A.27).

Via the navigation one can achieve the same results via an optimized structure:

- find the index of the section responsible for a given key (see Action A.28, Figure A.28),
- compute the position[1] of the section responsible for a given key(see Action A.29, Figure A.29),
- retrieve the node owning the given index (see Action A.30, Figure A.30),
- compute the preference list assuming all nodes are up for a given key (see Action A.31, Figure A.31).

Different kinds of active preference lists that only consider up nodes can be computed via the navigation component. Those include

- a plain preference list (see Action A.32, Figure A.32).
- a preference list in which nodes are annotated as primary or fallback node (see Action A.35, Action A.35).
- a preference list only containing primary nodes (see Action A.36, Figure A.36).

Resize Ring

A resize operation can be staged via the claim component. The ring size must be a power of 2 as otherwise load balancing cannot be achieved. When a resize is committed special precautions are taken when computing new ownership relations and handoffs to avoid too many unnecessary reassignments and data transfers. For details see Action A.41, Figure A.41.

Handoff

Nearly all kinds of cluster changes lead to a necessary data transfer. When such a cluster change is committed via the claim, an entry for scheduled handoffs is added in the updated ring. The vnode periodically reads these entries and checks for new ones. When it finds new handoff entries a handoff request is sent to the node handing off data. When the node is ready to handoff its data a handoff process folding over all stored data items is started by the node. For details see Action A.42, Figure A.42.

[1]The position of a section describes the index of the section on the ring starting from as opposed to the index resulting from the hash function.

4. Random Slicing

In this chapter we introduce the concept of RS in detail. The first section looks at RS purely as a partitioning algorithm without considering replication. Following this we discuss some simple approaches to integrate replication with RS.

4.1. Partitioning Algorithm

Miranda et al. have proposed RS[16] as an "efficient and scalable data placement for large-scale storage systems". The motivation to develop yet another data placement algorithm were several shortcomings of existing approaches like Consistent Hashing, especially when the cluster configuration changes often or scales to a high number of nodes. The goal for the system is to keep a good adaptivity and fairness when scaling a cluster while keeping look-up time and memory consumption low and supporting redundancy in the cluster. *Adaptivity* here means the ability to reconfigure the cluster without moving unnecessary data points stored on it. *Fairness* is defined as the ability to distribute data and requests to nodes according to their capacity.

The basic principle of RS is to assign one or more sections on the unit interval $[0, 1)$ to each node in the cluster such that

1. no assigned sections overlap,
2. the interval is assigned completely,
3. the sum of length of all sections assigned to a given node is equal to its relative capacity with respect to the total cluster capacity.

This basic functionality can be seen as another variant of Consistent Hashing[8].

In detail the system starts with an initial configuration of nodes $n_1, ..., n_k$ with capacities $c_1, ..., c_k$ and relative capacities

$$r_i = \frac{c_i}{\sum_{j=1}^{k} c_j}.$$

The unit interval is simply split into intervals of length r_i and each node is assigned the respective section. Whenever the cluster configuration changes, either by removing nodes, adding nodes or by changing the capacity of existing nodes, new relative capacities for all nodes are computed. Using the current assignment map and the new relative capacities a *gap collection* algorithm chooses which sections of the current mapping are split or completely reassigned. The authors propose and examine two simple algorithms and a

sorted variant for each. They determine that the *CutShift+Sorted* algorithm yields the best results with respect to keeping the number of new sections low, which is the biggest influence on memory consumption and lookup time. This algorithm tries to release only complete intervals first and alternates between splitting at the start and at the end of sections if splits are necessary to get bigger gaps. The sorted variant in addition assigns the nodes with the highest capacity to the largest gaps.

However, Scott Fritchie, the Riak developer who proposed using RS with Riak Core[8], implemented a greedy algorithm[1] for a different project which minimizes data transferred between nodes when the cluster is changed, as data transfers over a distributed network can be a bottle neck[8]. Its pseudocode can be found in Algorithm 1. This algorithm starts with computing the signed difference of relative capacities (Lines 4 to 6). Then, for each node with shrinking relative capacities gaps are collected greedily by iterating their sections and marking sections as unused until they meet the capacity difference (Lines 11 to 23). At most one section is split per node, which happens if the currently viewed section is larger than the remaining difference. The created gaps are then greedily assigned to the nodes with a growing relative capacity in the same manner (Lines 26 to 40). Therefore, at most one gap is split per node for the same reasons. Now that the complete interval is assigned, neighboring sections with the same owner are merged into one section (Line 42).

Determining the node a given key belongs to can be achieved by applying a hash function to the key and mapping the hash space to the unit interval. This is done both for storing and retrieving data. Redundancy can be achieved by different strategies and only some simple solutions are given in the original paper. We discuss some simple approaches to solving the redundancy problem in Section 4.2.

In addition to the description of their approach, Miranda et al.[16] conducted an analysis on different data placement strategies including two variants of Consistent Hashing in a simulated environment to compare them with respect to fairness, memory usage, lookup time, and adaptivity. The summary of their qualitative results can be found in Table 4.1 taken from the original paper[16]. As one can see RS performs better than the other strategies when looking at each of their weak points. It is however important to note that the analysis was done for large-scale system together with high redundancy for which some of the strategies were not developed. As mentioned before the adaptivity can be improved by using a different gap collection algorithm at the cost of memory usage and lookup time.

Figure 4.1 shows an example of RS with Fritchie's gap collection algorithm . In this example the unit interval is interpreted as a ring running clockwise with the 0/1 point at the top like in Consistent Hashing to make a comparison easier. The used nodes all have the same capacity. In the example each ring represents a configuration of the cluster, with the most outward ring being the initial and the inner ring being the final configuration. Initially 4 nodes get assigned a quarter of the ring each. In the second configuration one node is added. The new node takes an equal part of each existing section such that each node owns a fifth of the interval. Two new nodes are added in the

[1] `https://github.com/basho/machi/blob/master/src/machi_chash.erl`

Data: List of sections sections, List of new relative capacities capacities
Result: New assignment of sections to nodes sections

```
for s ∈ sections do
    old_capacities[GetOwner(s)] ← old_capacities[GetOwner(s)] + Length(s)
end
for {o, c} ∈ capacities do
    capacity_diff[o] ← c − old_capacities[o];
end
for s ∈ sections do
    o ← GetOwner(s);
    d ← capacity_diff[o];
    l ← Length(s);
    if d < 0 then
        if |d| ≤ l then
            {s1, s2} ← Split(s, l - |d|);
            s2 ← SetOwner(s2, unowned);
            sections ← Replace(sections, s, (s1, s2));
            capacity_diff[o] ← 0;
        end
        if |d| > l then
            s ← SetOwner(s, unowned);
            sections ← Replace(sections, s, s);
            capacity_diff[o] ← d + l;
        end
    end
end
for (o, d) ∈ capacity_diff do
    while d > 0 do
        s ← NextUnowned(sections);
        l ← Length(s);
        if |d| ≤ l then
            {s1, s2} ← Split(s, l - |d|);
            s1 ← SetOwner(s1, o);
            sections ← Replace(sections, s, (s1, s2));
            capacity_diff[o] ← 0;
        end
        if |d| > l then
            s ← SetOwner(s, o);
            sections ← Replace(sections, s, s);
            capacity_diff[o] ← d − l;
        end
    end
end
sections ← MergeSameNeighbors(sections);
return sections;
```

Algorithm 1: Gap Collection

Strategy	Fairness	Memory usage	Lookup time	Adaptivity
Consistent Hashing (fixed)	Poor	High	Moderate	Good
Consistent Hashing (adapt.)	Moderate	High	High	Poor
Redundant Share	Good	Low	Very High	Good
$RUSH_P$	Poor	Low	Very High	Very Good
$RUSH_R$	Good for low number of replicas	Low	Low	Very Good
$RUSH_T$	Good	Low	Very High	Very Good
Random Slicing	Good	Low	Low	Good

Table 4.1.: Qualitative analysis of different data placement strategies. Reprinted from [16].

next step. One can see, that some small intervals are reassigned completely and some are split. The transition is from 7 to 10 nodes in which one can easily see the mechanism of splitting gaps while assigning nodes. The fact that gaps are only assigned to new nodes implies that there are no transfers of data between existing nodes. This shows that the amount of transferred data is optimal for each reconfiguration.

4.2. Replication Placement Strategy

After the initial node to store a data block by its key is chosen through hashing on the intervals created by RS, additional $N-1$ copies have to be distributed to $N-1$ distinct physical nodes. For the correctness of RPSs it is assumed that the system configuration is correct, which means at times of operational states there are $n \geq N$ physical nodes in the cluster and $R \leq W \leq N$. The RPS should return a preference list of at least N nodes, where the initial node is the head and the tail consists of nodes where replicas are stored in descending preference.

Load balancing between all nodes should be achieved at least asymptotically such that it gets closer to the ideal value when the ratio of replicas to nodes in the cluster $\frac{N}{n}$ gets smaller. Considering only the primary replica, i.e. the head of the preference list, a placement strategy should always achieve perfect load balancing for a uniform distribution of data. The ideal case is that for relative capacities $c_1, .., c_n$ of the nodes their relative load l_i should be equal to the capacity c_i. The relative capacity of node n_i with weight w_i is $c_i = \frac{w_i}{\sum_{j=1}^{n} w_j}$. The relative load of node n_i after k keys have been distributed and k_i keys are stored on n_i is $l_i = \frac{k_i}{Nk}$.

To support the gossip protocol and local functionality of RCL the replication strategy must not use a central server or other communication between the nodes. The result should only be based on the current local view. If two local views are the same, the preference list should be the same, i.e be deterministic.

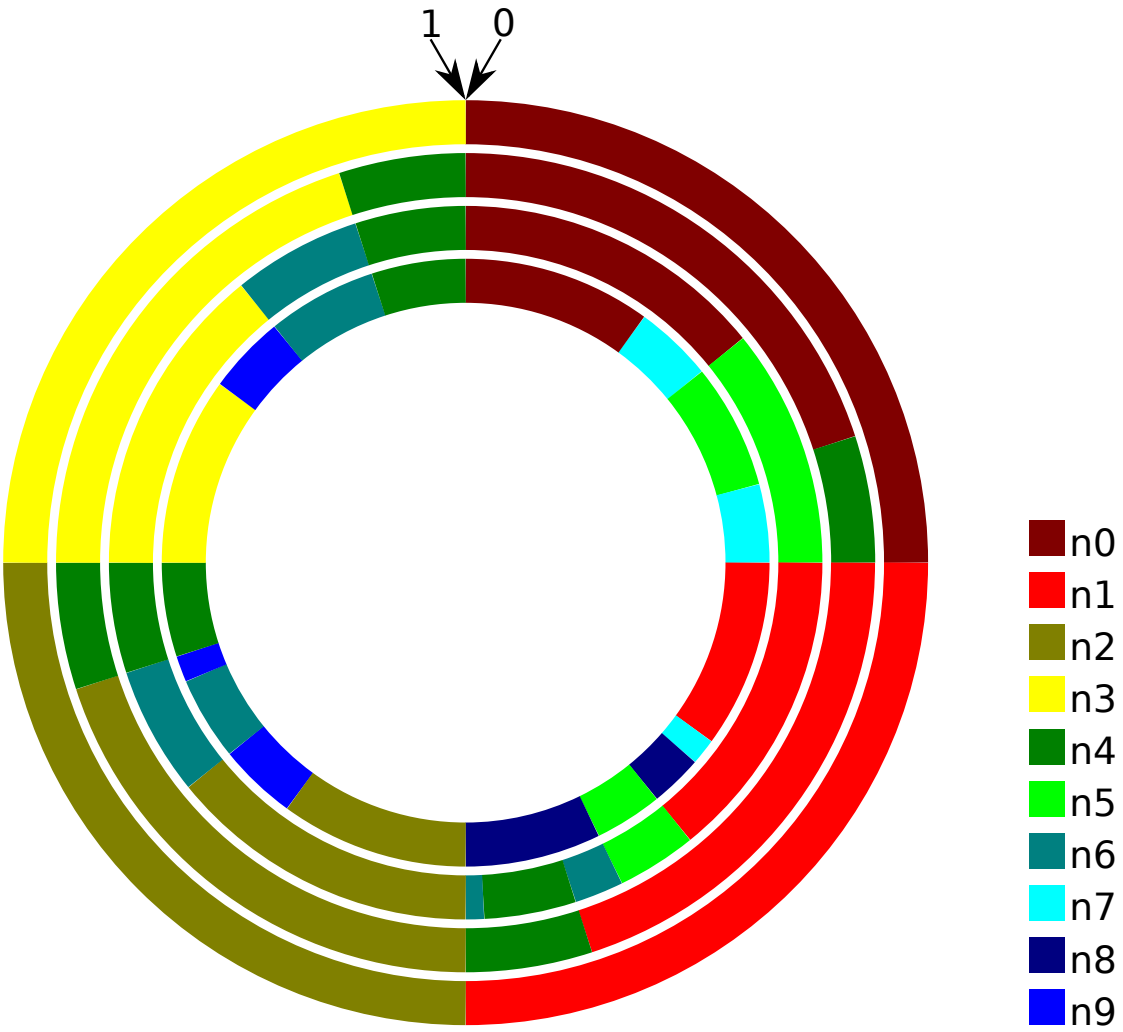


Figure 4.1.: Random Slicing example scaling from 4 to 5 to 7 to 10 nodes with homogeneous capacity.

4.2.1. Random Replication

This simple approach uses the outputs of a random uniform function as keys until N distinct physical nodes are hit. To achieve the same preference list for the same partitioning across all nodes the random function is seeded with the initial key. A pseudo code version of the algorithm is shown in Algorithm 2. Random Replication was chosen by Miranda et al. as the replication strategy in the analysis of RS[16]. Initially Brinkmann et al. mention this approach as a motivation for Redundant Share[4]. Theoretical results showing non-optimal capacity usage in a heterogeneous system seem to have a neglectable impact on load balancing according to the simulations done by Miranda et al.

```
Data: List of sections secs, number of required replicas N, key of data to be
      replicated k
Result: preflist of at least length N
1 i ← 0;
2 preflist ← ∅;
3 while Length(preflist) < N do
4 |   r ← Rand(i, k);
5 |   if GetNode(secs, r) ∉ preflist then
6 |   |   Append(preflist, GetNode(secs, r));
7 |   end
8 |   i ← i + 1;
9 end
```

Algorithm 2: Trivial Replication

Worst Case Runtime

Since the process of finding nodes to store replicas on is random the worst case is that not enough nodes are hit after a finite number of steps and the replication algorithm does not terminate. However, this is highly improbable. We show this by computing probabilities of a fixed node with relative capacity c being picked after k steps. To this end we model a random experiment with the geometric distribution. Each independent trial has a probability of c that the fixed node is drawn and $1 - c$ that it is not drawn. Therefore the probability to draw the node after exactly k steps is

$$Pr(X = k) = (1 - c)^{k-1}c,$$

the expected value for k is

$$E(X) = \frac{1}{c}$$

and the probability to draw the node after at most k steps is

$$Pr(X \leq k) = \sum_{i=1}^{k}(1 - c)^{i-1}c = c\sum_{i=0}^{k-1}(1 - c)^{i} = c\frac{(1 - c)^0 - (1 - c)^k}{1 - (1 - c)} = 1 - (1 - c)^k.$$

Scenario	$E(X)$	Pr_{1n}	Pr_{2n}	Pr_{3n}	Pr_{4n}	Pr_{5n}	$k_{0.95}$	$k_{0.99}$
S_1	10	0.651	0.878	0.958	0.985	0.995	29	44
S_2	1000	0.632	0.865	0.950	0.982	0.993	2994	4603
S_3	100	0.182	0.331	0.453	0.552	0.634	298	458

Table 4.2.: Example Probabilities with Random Replication. Shows expected value for k, probabilities that the node is chosen within a given number of steps and number of steps needed to be chosen with a given probability.

Solving the inequality $Pr(X \leq k) = 1 - (1 - c)^k \geq p$ for k yields

$$k \geq \log_{1-c}(1 - p)$$

and therefore the minimal value fulfilling this inequality is

$$k = \lceil \log_{1-c}(1 - p) \rceil$$

To further illustrate these values first we choose three different scenarios and compute $E(X)$, $Pr(X \leq k)$ for $k \in \{i \cdot n \mid i \in \{1, ..., 5\}\}$, and the minimal values for k such that $Pr(X \leq k) \geq 0.95$ and $Pr(X \leq k) \geq 0.99$. The first scenario S_1 is derived from the example shown in Figure 4.1. Here there are 10 homogeneous nodes with the same capacity. Therefore $n_1 = 10$ and $c_1 = 0.1$. The second scenario S_2 is motivated by a high scaling with 1000 homogeneous nodes. Therefore $n_2 = 1000$ and $c_2 = 0.001$. The last scenario S_3 represents a heterogeneous system in which we consider a node with only a tenth of the average capacity. We choose $n_3 = 10$ and $c_3 = 0.01$

The numbers for those scenarios are shown in Table 4.2. An interesting aspect is that for the homogeneous scenarios all values appear to scale with the number of nodes. The number of steps needed to guarantee that a given node is chosen with a specific probability also scales with the ratio of the actual capacity of the node to the average capacity. While these results are not surprising when looking at the equation we intend to give an easier understanding with those examples.

Example

Using the final ring from the example in Figure 4.1 and the key 0.345 the resulting preference list is [n1,n8,n4,n9,n5,n2,n7,n6,n0,n3]. The random algorithm used to create the random keys is Erlang's implementation of the Xorshift116 generator combined with the StarStar scrambler[3].

Discussion

Using random replication leads to near perfect load balancing according to simulations by Miranda et al[16]. It is highly unlikely that the same node is drawn over and over when the ratio of available nodes to required nodes is high enough and therefore the

runtime can be expected to be nearly linear in the number of required nodes. Since the random function is seeded with the primary key, the placement on the unit interval of replica candidates is deterministic for each key. Therefore after a cluster reconfiguration not all replica placements have to be recomputed as they can be moved to the node now responsible for that section. Recomputation is only necessary if the positions of two replicas are owned by the same node after a reconfiguration and therefore the preference list not having enough distinct members.

4.2.2. Ring Rotation

Another approach has the intention of keeping Riak Core's ring structure and making use of it. A pseudo code version of this algorithm is shown in Algorithm 3. The ring rotation is visualized in Figure 4.2. Its idea is to stepwisely rotate the ring counter-clockwise, or from another perspective, the key clockwise, by the lengths of its sections and use the original hash value to create the preference list. The aim of this approach is to achieve load balancing by considering the lengths of the sections. However, there is a possibility that after a whole rotation of the ring less than N nodes are in the preference list. To overcome this, in the i-th rotation the sections are split into 2^i subsections and therefore the rotation step lengths get shorter (Line 6). This leads to smaller segments that were previously skipped by rotating by a bigger length to be added to the preference list. Since the rotation of the subsections leads to querying the same key value multiple times, a small optimization would be to skip those values (Line 15). This is done by rotating the ring by $\frac{1}{2^i}$ of the original length once to offset the value, then rotate it $2^{i-1} - 1$ times by $\frac{1}{2^{i-1}}$ of the original length and finally the offset is reverted without querying the ring. Implementation wise instead of the ring being rotated counter-clockwise the key is rotated around the ring clockwise.

Correctness

The algorithm is seen as correct if it constructs a preference list of N distinct physical nodes for any correct system configuration with $n \geq N$ nodes. The maximum rotation length will be eventually smaller than the shortest section. Therefore after a finite amount of rotations each section on the ring will be queried which leads to a preference list containing each physical node. Since the system configuration is assumed to be correct the preference list contains at least N distinct physical nodes.

Worst Case Runtime

In the worst case N is equal to the number of distinct physical nodes and the ring actually has to rotate the maximum number of rounds. Let s be number of segments, l_{max} length of the largest segment, l_{min} length of the shortest segment. Starting with round $i = 0$ the number of steps per round is $r(i) = 2^i \cdot s$ Therefore the total number of steps for k

Data: Mapping of ranges to nodes m, number of required replicas N, index of data to be replicated k
Result: preflist of at least length N

```
1  i ← 0;
2  preflist ← ∅;
3  m ← Append(SubList(m, GetNode(k), Length(m)), SubList(0, GetNode(k)));
4  while Length(preflist) < N do
5      for s ∈ m do
6          step ← GetLength(s) · 2^−i;
           // One step to offset the key
7          k ← (k + step);
8          if k ≥ 1.0 then
9              k ← k − 1.0;
10         end
11         if GetNode(m, k) ∉ preflist then
12             Append(preflist, GetNode(m, k));
13         end
           // Skip previously visited indices
14         for j ← 1 to 2^(i−1) − 1 do
15             k ← (k + 2 · step);
16             if k ≥ 1.0 then
17                 k ← k − 1.0;
18             end
19             if GetNode(m, k) ∉ preflist then
20                 Append(preflist, GetNode(m, k));
21             end
22         end
           // One step to change offset to first index of the next node
              without lookup
23         k ← (k + step);
24         if k ≥ 1.0 then
25             k ← k − 1.0;
26         end
27     end
28     i ← i + 1;
29 end
30 return preflist
```

Algorithm 3: Rotation for each segment

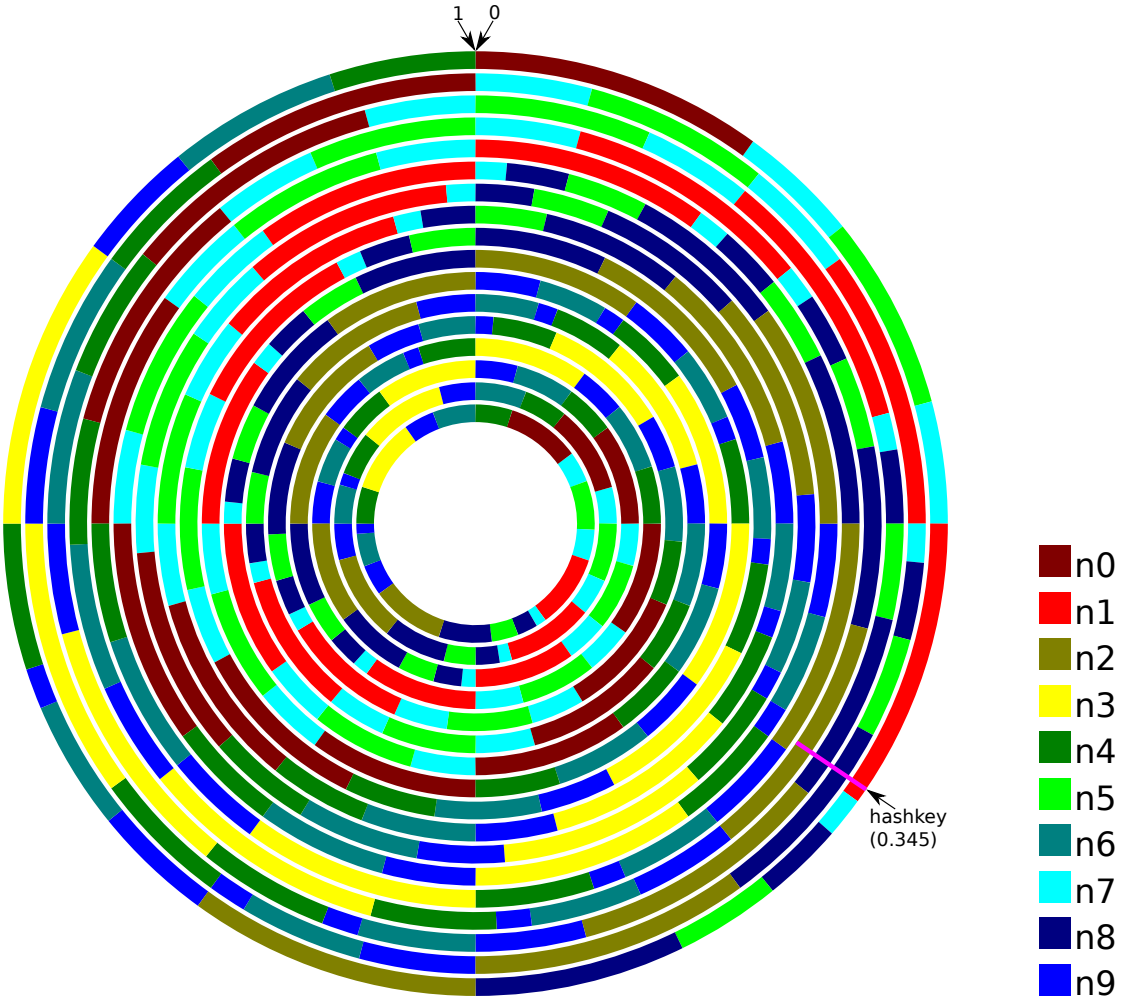


Figure 4.2.: Visualization of the ring rotation replication. n0...n9 are the physical nodes. The ring is rotated counterclockwise by the length of each section. From the outer to the inner ring each rotation step is shown.

rounds is

$$\sum_{i=0}^{k-1} 2^i \cdot s = s \cdot \sum_{i=0}^{k-1} 2^i = s \cdot \frac{1-2^k}{1-2} = (2^k - 1) \cdot s$$

and the number of rounds in the worst case is

$$\begin{aligned} \frac{l_{max}}{2^k} &\leq l_{min} \\ \frac{l_{max}}{l_{min}} &\leq 2^k \\ k &\geq \log_2(\frac{l_{max}}{l_{min}}) \\ k &= \lceil \log_2(\frac{l_{max}}{l_{min}}) \rceil. \end{aligned}$$

Load Balancing

RS guarantees load balancing for the primary copy of a data item. Looking at Figure 4.2 one can see that sections owned by the same node do overlap from one rotation step to another. This complicates a quantitative analysis of load balancing and we have to rely on simulated data for quantitative results. However, the overlaps intuitively lead to an imperfect load balancing as the overlapping section is unused in all rotation steps but the first it appears in.

Example

Using the example from Figure 4.2 and hash key 0.345, the resulting preference list is [n1, n8, n2, n9, n4, n3, n6, n0, n5, n7]. To achieve a preference list for $N = 3$, 3 rotation steps are necessary for that key. To achieve a preference list for $N = 10$, 15 rotation steps are necessary for that key.

Discussion

From the initial look the ring rotation replication seems to be sub-optimal both with respect to the expected runtime as well as the expected load balancing. Since the rotations are based on the length of sections a reconfiguration of the cluster leads to a total rearrangement of replica placements. Overall this approach does not look like a promising solution to the placement of replicas.

4.2.3. Ring Jumping

Similar to the ring rotation approach ring jumping makes use of the original ring structure. A pseudo code version of the algorithm is shown in Algorithm 4 and a visualization can be seen in Figure 4.3. However, instead of rotating the ring in the order of its sections

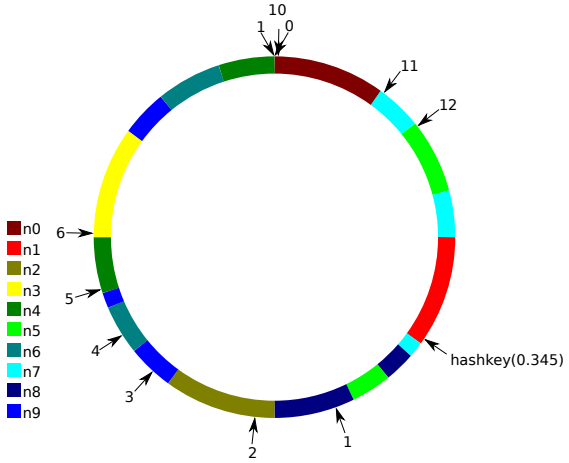


Figure 4.3.: Ring Jumping

it is rotated by the length of the section the key currently belongs to. This is repeated until N distinct physical nodes are added to the preference list.

Data: List of sections secs, number of required replicas N, index of data to be replicated k
Result: preflist of at least length N

```
1  preflist ← ∅;
2  while Length(preflist) < N do
3  |  k ← k + GetLength(secs, k);
4  |  if k ≥ 1.0 then
5  |  |  k ← k − 1.0;
6  |  end
7  |  if GetNode(secs, k) ∉ preflist then
8  |  |  Append(preflist, GetNode(secs, k));
9  |  end
10 end
```

Algorithm 4: Ring Jumping

Correctness

Assume that the key is rotated around the ring such that it is within the initial section again for the first time. If it is at its exact starting place then the sum of lengths of sections it hit is 1, which can only be the case if it visited every section and therefore every physical node was added to the preference list. Otherwise, the starting place of the key for the next rotation is less than the initial one. Then there is a chance that segments that where skipped before are now added to the preference list. This is done until the key is at an initial position between the start of the starting section and its offset by the length of the shortest section. Then it is guaranteed that every section is

visited in the following rotation.

Worst Case runtime

The worst case occurs when the initial key is at the right edge of the longest segment and the shortest segment has to be visited. Additionally only the shortest segment is not visited and therefore the position offset for each round is minimal. Let s be number of segments, l_{max} length of the largest segment, l_{min} length of the shortest segment. Assume that only the shortest segment is not visited. Therefore the start index of the next round moves by l_{min}: $s_{i+1} = s_i - l_{min}$. The shortest segment is guaranteed to be hit in round k if $s_k \leq l_{min}$. So the number of rounds needed is $\lceil \frac{l_{max}}{l_{min}} \rceil$ and the number of steps needed is $s \cdot \lceil \frac{l_{max}}{l_{min}} \rceil$.

Load Balancing

Analyzing load balance properties of jumping approach qualitatively is complicated by the influence of section lengths and order. Assuming equal section length and distribution this approach is equivalent to Riak Core's RPS as the N next sections are chosen by definition. Abstracting from this special case the load on a section is expected to scale with its relative size. However, the actual numbers need to be determined experimentally in a simulated environment.

Example

In Figure 4.3 one can see how the 10 nodes are selected for the preference list for the key 0.345. This results in [n1, n8, n2, n9, n6, n3, n0, n7, n5]. Defining the *efficiency* of the approach by the ratio of number of selected nodes to number of queries shows an efficiency of 100% for the first six nodes. To get a complete preference list with 10 nodes 13 queries are required resulting in an efficiency of 76.9%.

Discussion

While not having the best expected runtime and unknown load balancing after the initial look this approach still has the potential to prove itself as a feasible solution for the replica placement in simulations. As the approach is derived from Riak Core's algorithm it may also make visualizing differences easier. One drawback of the approach is the dependency on the length and order of sections. This creates a need to recompute the replica placements for any cluster reconfiguration and a repair operation. If this is not done, keys can be lost with each cluster change.

5. Riak Core Lite and Random Slicing

This chapter covers how the architecture and behavior of RCL was changed while replacing Consistent Hashing with RS. First we give an explicit recapture of the responsibilities and guarantees of Consistent Hashing in RCL, how RS compares in those aspects, and what open problems directly follow from this comparison. Analogous to Section 3.2 the next sections contain changed and new components as well as system actions that changed.

5.1. Comparing Consistent Hashing and Random Slicing

In RCL Consistent Hashing provides a homogeneous partitioning of the ring. Together with the claim algorithm it guarantees that for each partition the N successor partitions are owned by N distinct nodes if there are enough nodes in the cluster. This enables a simple and fast computation of the preference list. The indices on the ring are fixed unless a special resize operation is executed. We visualize the difference between the ring structures in Figure 5.1.

On the other hand RS partitions the ring into dynamic sections. It does not give a guarantee on the order of owners and therefore does not enable a simple and fast computation of the preference list. Since the partitioning of the ring happens dynamically there is no resize operation.

From this one can conclude that open problems are the computation of the preference list and if there are changes that originally happened during the resize operation and are still relevant and therefore need to be applied elsewhere. We discuss solutions for the computation of the preference list in Section 4.2 and other open problems are discussed in the appropriate places in the following sections.

5.2. Changed Components

In this section we explain how the components listed in Section 4.2 were changed during the integration of RS both with respect to their responsibilities as well as their basic functionality. Any component that is not mentioned in this section was left unchanged or only changed on a technical level.

5.2.1. Ring

On an abstract level there are only minor changes in the ring component. The model of the ring changed to that of RS and therefore the computation of the preference list

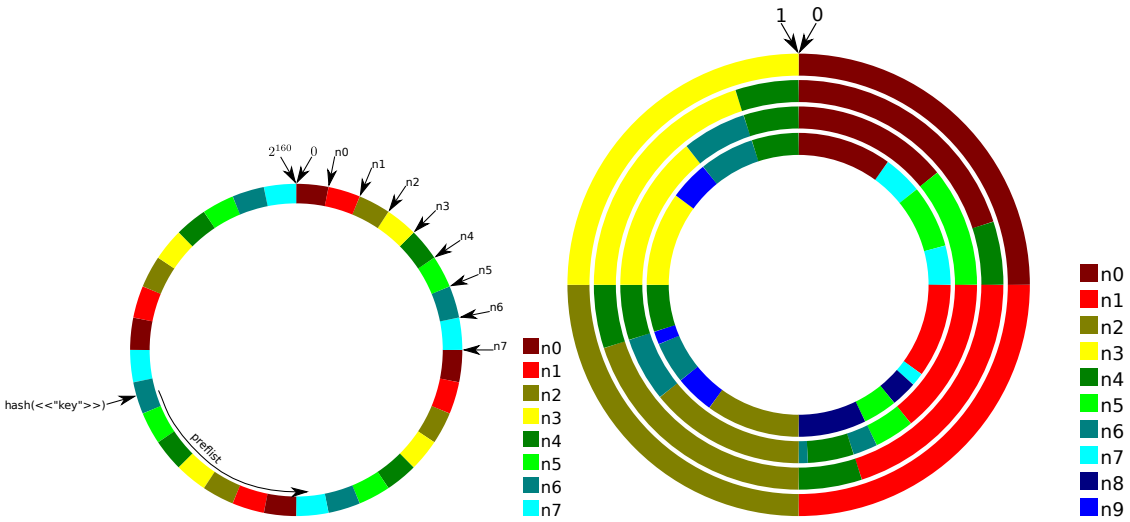


Figure 5.1.: Side by Side Comparison of Consistent Hashing and Random Slicing. On the left one can see the homogeneously partitioned Consistent Hashing ring while on the left one can see the dynamic changes in the RS ring.

.

is now solely relying on the navigation component. As the indices are not fixed in the new ring model the ring reconciliation algorithm had to be overhauled and relies now on choosing the most up-to-date ring and adding all missing members to it via the RS algorithm.

5.2.2. Navigation

The navigation component had to be reworked almost completely. First of all, the Consistent Hashing model at its core was replaced by an adaptation of Scott Fritchie's RS implementation (see Sec 4.1). From that it directly follows, that the preference list cannot be computed by iterating over the successor nodes in the ring. Instead the component uses the RPSs we present in Section 4.2. This also leads to strong restrictions on the use of the optimized read-only structure this component uses as it cannot be used to compute the preference list.

5.2.3. Claim

As the core component of the claim component the claim algorithm is obsolete as load balancing on the ring is directly achieved by Random Slicing. Since the claim component is also responsible for the stage-plan-commit life cycle it still stays relevant and is the entry point for changes to the ring. Instead of applying the claim algorithm in the commit phase the new ring is computed by using RS with the old ring and new cluster configuration. Through this it is assured the new guarantees given by the RS model are fulfilled in the actual ring.

5.3. Changed System Procedures

In this chapter we explain the changes we made to the system procedures while integrating RS with Riak Core. Procedures that are mentioned in Section 3.2.2 but do not appear in this section are not changed significantly.

The system still uses the stage-plan-commit life-cycle for cluster changes. The stage phase has not changed as the cluster changes are still marked in the ring and gossiped to all members. However, in the commit phase the updated ring is now computed by simply applying RS to the old ring with the new member configuration.

5.3.1. Leave Cluster

The procedure of leaving the cluster remains unchanged in the most parts. However, the leaving node is now actually removed from the ring on the next update-ring-event instead of the next claim execution.

5.3.2. Navigate the Ring

As we mentioned in Section 5.2.2 the navigation component only offers reduced functionality with the optimized binary structure. The removed functionalities include computing the preference list via that structure. It now uses the ring model directly. This does not restrict the functionalities of the navigation component but it may have a negative impact on the performance. Other procedures navigating the ring were only changed in the aspect that they now use the RS ring model together with the new replication placement algorithms.

5.3.3. Resize Ring

As there is no fixed ring size and the ring structure is changed with every cluster change there is no explicit resize operation anymore. In the current state of the system any optimization and corner case handling in the handoff and ring update procedures during a resize operation is lost.

5.3.4. Handoff

While the actual execution of the handoff from the old owner of an index to the new owner has not changed the preparation of the handoff has changed significantly. As the ring has no fixed indices anymore it has to be assured that there are virtual nodes running for all old and new indices. This is achieved by computing a handoff-ring from the old and new ring that includes all indices of both rings and each index is owned by the onwer that would own it in the old ring. This procedure is visualized with an example in Figure 5.2. From this handoff ring the correct virtual nodes are running and can start the handoff procedure as usual. When there are two neighboring sections owned by the same physical node the sections are merged on the next ring-update-event.

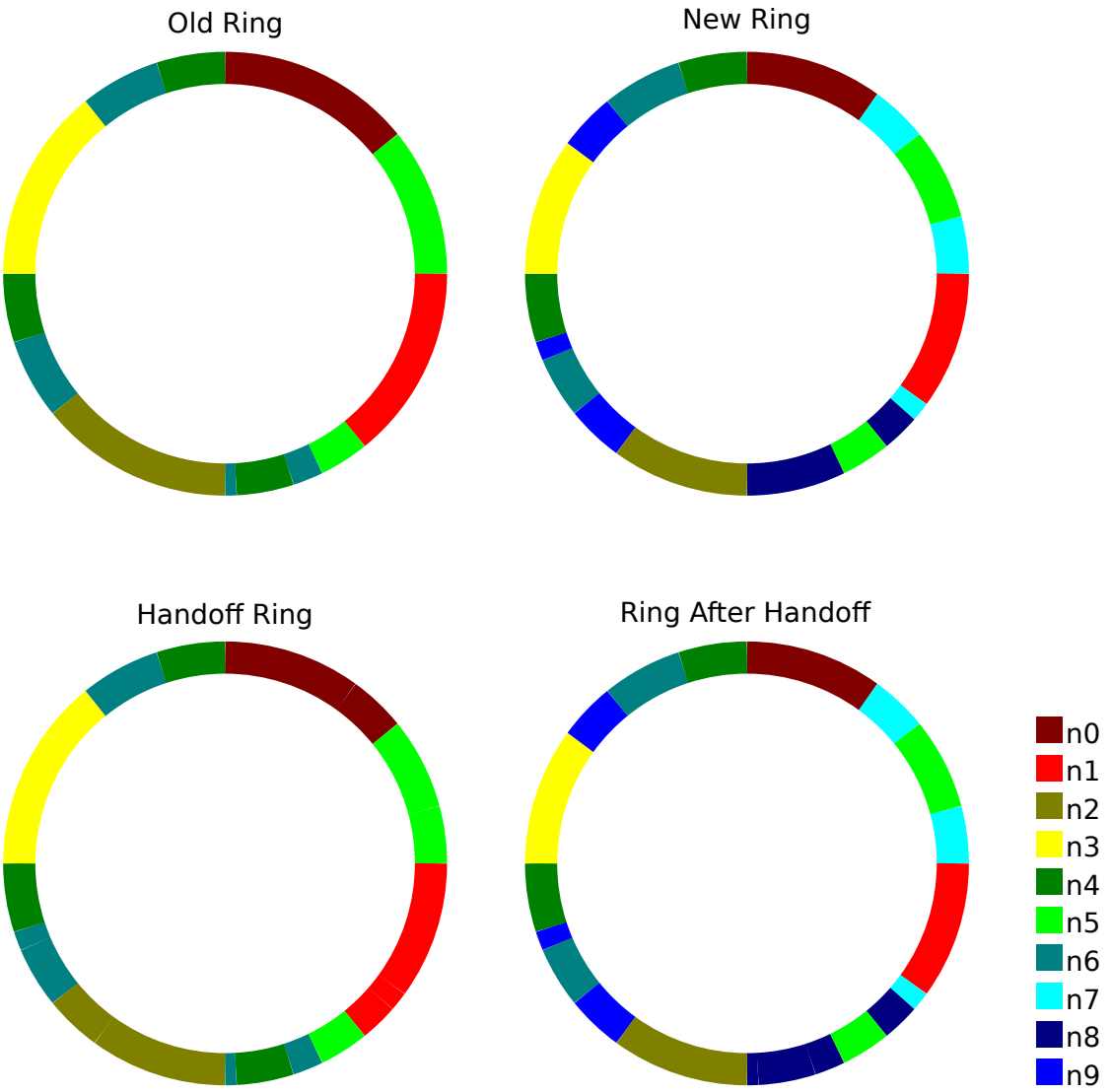


Figure 5.2.: Handoff Ring. First the indices of the old and new rings are merged and then assigned to the old owners. After the handoff the new ring results.

6. Evaluation

In this chapter we present the evaluation of the integration of RS with RCL. First we present hypotheses on how the adapted system will perform with respect to different aspects compared to the current RCL implementation. Following we explain which tools we use and how the environment and benchmarked systems are configured. After this we explain which metrics we plan to take from the benchmark data to test our hypotheses. Closing this chapter we show and interpret the results with respect to our hypotheses.

6.1. Hypotheses

In this section we present and reason about the hypotheses we test in this evaluation. The hypotheses are based on the theoretical thoughts presented in Chapters 3 and 4.

One variant in the implementation of RS in RCL having a big impact is the replication placement algorithm. The aspects most interesting to examine in this evaluation are the impact on load balancing and throughput. From the initial discussion Random Replication seems to be the most promising approach of the simple ones in those aspects. Therefore we postulate the following hypotheses:

Hypothesis 1 *The average divergence from optimal load balancing over all nodes for Random Replication is less than for Ring Rotation or Ring Jumping.*

Hypothesis 2 *With Random Replication the system has a higher average throughput than with Ring Rotation or Ring Jumping.*

In the following when we speak of RS we mean RS with the best performing of the replication placement algorithms. From Miranda et al.'s analysis of RS[16] it follows that it performs better than the original Consistent hashing both with respect to load balancing as well as performance. However, RCL uses an implementation of Consistent Hashing that is quite different from the original one and the system architecture is built around that version. Additionally, in the scope of this thesis RS was integrated with RCL in a prototyping manner as a proof of concept and misses much architectural planning and loses a lot of optimization. For those reasons the hypotheses concerning the comparison of RCL and RS with RCL and Consistent Hashing are more conservative. With respect to load balancing we assume that the current RCL implementation is already optimized. As the replication placement strategy has a big impact on the load balance and we only use simple approaches we assume RCL will perform slightly worse with RS than with Consistent Hashing. We quantify "slightly worse" as a 10 percent higher deviation from optimal load balance.

Hypothesis 3 *Let o be the optimal load balancing, l_{CH} the average divergence from o for Consistent Hashing, l_{RS} the average divergence from o for RS. Then $l_{RS} \leq 1.1 \cdot l_{CH}$, meaning the average divergence is less than 10% higher.*

Hypothesis 4 *While the system is running without cluster changes the throughput t_{RS} with RS is at least 90% of t_{CH} with Consistent Hashing ($t_{RS} \geq 0.9 \cdot t_{CH}$).*

Hypothesis 5 *The recovery time after a handoff of RCL is shorter with RS than with Consistent Hashing.*

6.2. Setup

In this section we describe the setup of the evaluation process in which we test the hypotheses. First we shortly present the tools needed to evaluate RCL and after this how the environment and execution run is setup.

6.2.1. Tools

As RCL is only a framework and cannot be evaluated on its own an implementation using it as the backend is needed. To this end we use the reference implementation rclref. To execute the benchmark we use rcl_bench.

rclref

rclref[1] is a reference implementation demonstrating how RCL can be used to build a distributed key-value store. It was created by Riki Otaki in Google's Summer of Code 2020. Two of its features are a consensus implementation and enabling read-repair. It implements simple put and get request and enables the configuration of the storage to be in-memory or persistent on disk. In the evaluation scenario we use in-memory storage. As RCL is simply a dependency that is started on application start we only need to change the dependency path between the RCL versions.

rcl_bench

rcl_bench[2,3] is a simple benchmarking tool for RCL applications. It requests defined operations to nodes and records successful operations with a timestamp. It uses a driver which implements the operations to be executed during the benchmark and has configurable settings on the percentage of each operation, duration of the benchmark, and number of concurrent workers. The driver used here uses Riki Otaki's implementation of rclref operations[4] for an old version of rcl_bench ported to the current version.

[1] `https://github.com/Tronso/rclref/tree/1e9e41e9c547da13cc200c620cb6b7e25b47b65d`
[2] `https://github.com/Tronso/rcl_bench/tree/5e3c433a3b8822fd92cf3b6892c291075c127e37`
[3] `https://github.com/Tronso/rcl_rs_bench/tree/4ef3ec5c322eb5c607a1931f8604303bf15b55ad`
[4] `https://github.com/wattlebirdaz/rcl_bench`

6.2.2. Configurations

In this section we describe different parameters of the setup and which values we choose for them. First we describe the environment the evaluation is run on. After that we explain some parameters of RCL that are relevant for the evaluation. Finally, we explain how the execution of the benchmarking process is set up.

Environment Configuration

The environment for the benchmarking process consists of two machines with four physical cores each which are connected on a local area network. The first machine (M_1) consists of an Intel Core i5-7600K with 4 physical cores at 3.80GHz and 16GB memory. The second machine (M_2) consists of an Intel Core i5-8250U with 4 physical cores at 1.60GHz and 8 GB memory.

Riak Core Lite and rclref Configurations

In rclref the relevant parameters are the values for the number of replicas n, the least amount of successful reads r, and the least amount of successful writes w. As it is standard for the Dynamo architecture[6] we set $(n, r, w) = (3, 2, 2)$.

In RCL the relevant parameters differ between RS and Consistent Hashing. With Consistent Hashing the initial ring size is relevant while it is not needed with RS. Having 8 physical cores available and one being used by the benchmarking process there are a maximum of 7 nodes in the cluster. Trading off between load balancing and file accesses the ring size is set to 64.

With RS the only relevant parameter for the evaluation is the replication placement algorithm. Since we compare all presented algorithms there will be a configuration for each of them.

Execution Configuration

To get a more fair comparison there will be different configurations for execution of the benchmarking process. To minimize the non-deterministic impact of thread scheduling on the process only one erlang node is to be started for each physical node and an unnecessary background processes are shut down. As 8 physical cores are available and one is reserved for the benchmarking tool the cluster can consist of a maximum of seven nodes. As the n value is set to 3 at minimum 3 nodes have to be a member of the cluster for the system to run properly. This leads to the base configuration C_0 where on M_1 the rcl_bench node as well as three rclref nodes are running. To allow for perfect load balancing with Consistent Hashing a cluster size of a power of two is desirable. To this end configuration C_1 in which one additional node jons the cluster on M_2 after the system is in configuration C_0. In the final configuration C_2 the final three nodes join the cluster to reach of the maximum of seven nodes.

As the three nodes on M_1 are running in each configuration we start three workers in rcl_bench that send requests to these nodes while nodes running on M_2 only get

Workload	get	get_own_puts	put
write heavy	25%	25%	50%
read heavy	47.5%	47.5%	5%

Table 6.1.: Workloads

Parameter	Values
Riak Core Lite Configuration	ConsistentHashing, RandomSlicing_Jumping, RandomSlicing_Random, RandomSlicing_Rotation
Cluster Configuration	C0, C1, C2, dynamic
Workload	read_heavy, write_heavy

Table 6.2.: Configuration Parameters

forwarded keys they are responsible for. The operation mix consists of get_own_puts, put, and get where get_own_puts guarantees the get-request to be for a key owned by the node the request is sent to. We run the benchmark with a read-heavy and write-heavy workload configuration. According to Cooper et al.[5] a write-heavy workload consists of 50% reads and 50% writes while a read-heavy workload consists of 95% reads and 5% writes. Of the reads in each configuration 50% are reads on keys owned by the node. This results in the workloads shown in Table 6.1 One benchmark execution runs for 30 minutes. The keys used for the operations are uniformally distributed between 0 and 100000.

We define two types of benchmarking types. The static benchmark starts with a fixed cluster configuration and runs without changes for the 30 minutes. The dynamic benchmark starts in C_0, changes to C_1 after 10 minutes and changes to C_2 after an additional 10 minutes. Therefore there are three static and one dynamic benchmark runs for the four RCL configurations each with one read-heavy and one write-heavy run which results in 32 benchmark runs to be used for the evaluation. An overview of the configuration parameters and their values can be found in Table 6.2.

6.3. Metrics

To test the presented hypotheses some metrics can be taken directly from the benchmark results or rclref while others have to be computed. An overview of all metrics and where they stem from can be found in Table 6.3.

6.4. Results

In this section we present parts of the evaluation data representative for the typical behavior of the different RCL Configurations. The full data can be found in Appendix B.

ID	Description	Source
D01	Operations over time	Accumulated benchmark results
D02	Average throughput	Total number of operations in D01 divided by execution time
D03	Average throughput at handoff	Total number of operations in a sliding time window divided by window size.
D04	Keys owned by node	rclref API
D05	Optimal load	Total number of keys from D04 divided by number of nodes
D06	Divergence from optimal load	Absolute difference of the number of keys owned by a node and optimal load.

Table 6.3.: Metrics

Additionally we will compare the data and qualitatively assert how it supports or contradicts our initial hypotheses. We will also reason about causes for any unexpected results and other anomalies with a view of more technical details.

6.4.1. Common Anomalies

Before looking at the actual data we are showing and explaining unexpected behavior that is present to a more or lesser degree in all configurations. Specific anomalies that only occurred in single runs are addressed in the according section.

Decrease of Throughput

Especially with Consistent Hashing it was expected for the throughput to be constant during a benchmark run as the system itself does not produce a growing overhead without changes to the cluster. However, as Figure 6.1 shows exemplary, the throughput initially decreases steeply until after sometime it seems to decrease at a slower rate. We assume the phase of decrease stems from growing ets tables used by rclref that fill with new keys. The slower decrease stems from ets tables that grow because for each updated value on an existing key the vector clock associated with that key is updated. Therefore the size of the table still grows however not as much as with freshly inserted keys.

High Throughput During Handoff

We expected the benchmark to show a lower throughput and higher latency during the handoffs after a cluster change in the dynamic configuration. However, as Figure 6.2 shows, shortly after the cluster changes at 600s and 1200s the throughput rises to rates much higher than before the change and the latency goes to 0. This can be explained in one part by rclref vnodes are not handling requests during the handoff and just ac-

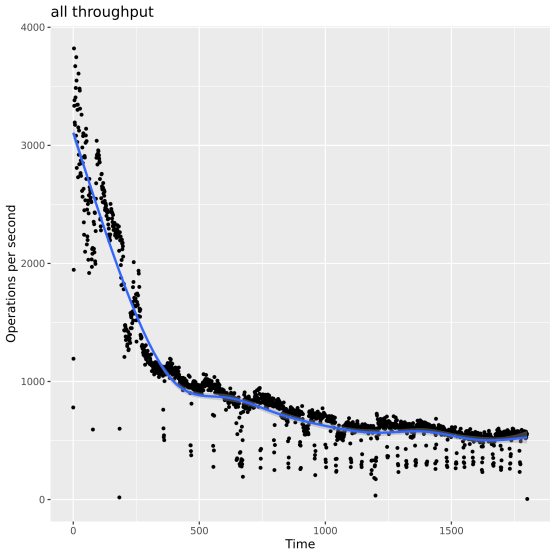


Figure 6.1.: Throughput Anomaly. The throughput shows a steep linear decrease before changing to a shallower linear decrease.

knowledging it, and in another part that rclref uses one ets table per vnode and as a consequence of the cluster change there are new vnodes with empty ets tables.

Optimal Load Balance in C0 with Random Slicing

In configuration C0 with just three nodes RS achieves a perfect load balance with the exact same number of keys stored on each node with any of the RPSs. This can easily be explained by the nature of RS. Since the three nodes are the initial configuration, RS produces a ring with exactly three sections, one for each node. With $n = 3$ set in the configuration each key is therefore replicated on three sections, which leads to each node storing each key and therefore owning the same number of keys. As for configurations with Consistent Hashing the ring size is set to 64 the sections cannot be perfectly assigned to three nodes and therefore this configuration does not achieve a perfect load balance.

6.4.2. Consistent Hashing

With Consistent Hashing there is no significant difference between read-heavy and write-heavy workloads. A visualization of runs in C0 and C2 is shown in Figure 6.3 There is a clear trend of lower throughput with a higher number of nodes. In configuration C0 with three nodes the average throughput starts at about 3500 operations per second, steeply decreases to 1000 operations per second at about 450s and from there slowly decreases to 500 operations per second. In configuration C1 with four nodes there is no clear difference to C0. However, in configuration C2 with seven nodes the average throughput starts at about 750 operations per second, there is no steep decrease, and it

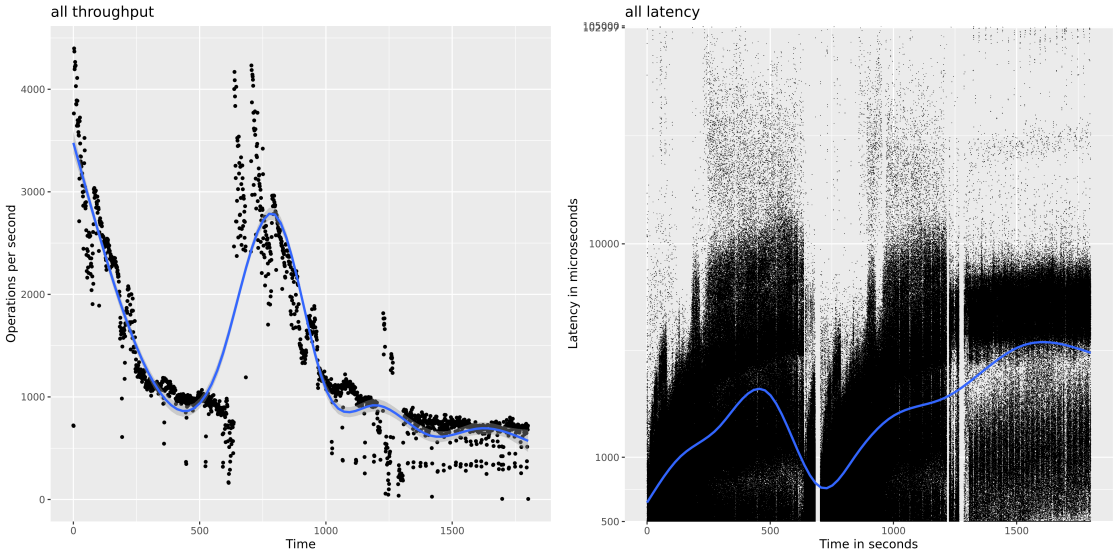


Figure 6.2.: Handoff Anomaly. Instead of a low throughput and high latency after a cluster change the system has a high throughput and low latency.

slowly decreases to 450 operations per second. One can see, that while starting out at a way lower rate the throughput ends up at a closer rate between the configurations.

For the dynamic benchmark runs there are significant differences between read-heavy and write-heavy workload. Ignoring the impact of the cluster changes the throughput plot seems to be quite similar as it decreases to 1000 operations per second within 450 seconds and after that decreases to about 600 operations per second. However, it seems that for the read-heavy workload the change from C0 to C1 does not have a great impact on the throughput or latency while it has a significant impact during the write-heavy workload. This can be seen both in the throughput and latency plots in Figure 6.4. For the change from C1 to C2 at 1200s the impact of the workload seems to be inverse to the ones of the change from C0 to C1. Ignoring the read-heavy dynamic run as an inexplicable anomaly looking at the latency plot for the write-heavy dynamic run seems to show a short impact at 600s and 1200s each that is followed by a longer disruption after a short delay. Both the phase of initial disruption and the following disruption seem to be longer for the change from C1 to C2. The initial disruption may be caused by the reconfiguration of the ring and nodes being set to forwarding while the following longer disruption can be explained by the actual handoff operation and potential repair operations.

Regarding the divergence from optimal load balance with Consistent hashing there is an absolute overall average divergence for key load of 0.003821643008703, for put load of 0.032386205130647, and for get load of 0.032384489514972. The highest divergence is measured with seven nodes in C2 and the lowest in C1 with four nodes. This is quite intuitive as with four nodes the nodes can be perfectly assigned to the 64 sections of the

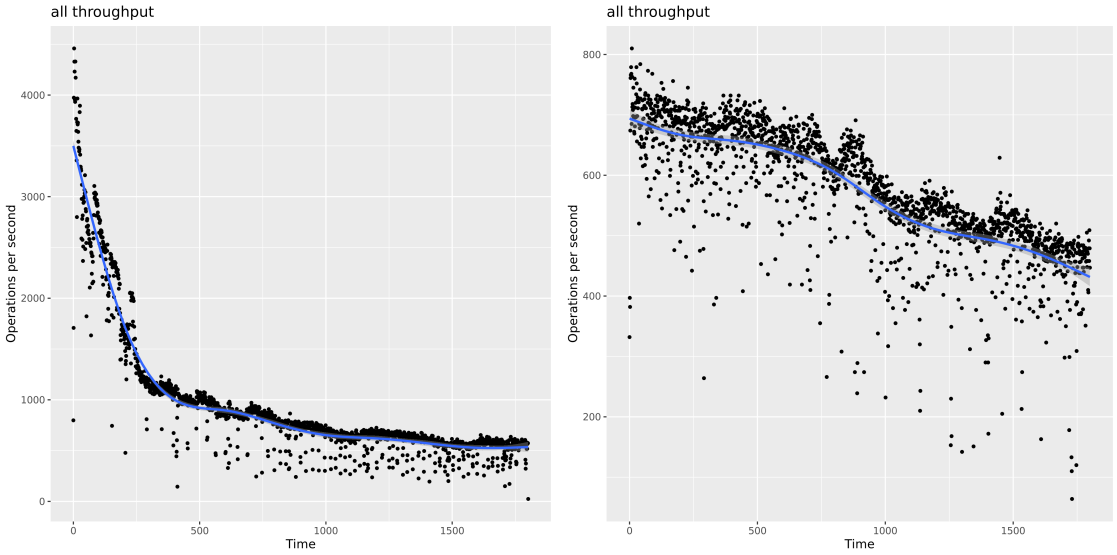


Figure 6.3.: Consistent Hashing Throughput Comparison. Showing the throughput for Consistent Hashing in C0 and C2 with write-heavy workload.

ring, while with three and seven nodes some nodes have one section more assigned to them than others.

6.4.3. Random Slicing Jumping

As with Consistent Hashing one cannot see a significant difference between read-heavy and write heavy workloads when looking at the throughput of the system. A visualization of a run in C0 and C2 is shown in Figure 6.5. One can see an influence of the number of nodes in the system on the performance. This can easily be explained by the higher number of sections and therefore higher lookup time. In configuration C0 with three nodes the average throughput starts at about 3500 operations per second, steeply decreases to 1000 operations per second at about 450s and from there slowly decreases to 500 operations per second. The change to four nodes does not significantly change the throughput plot as there are only three new sections on the ring. However, in configuration C1 with seven nodes the average throughput starts at 1300 and 900 operations per seconds for the read-heavy and write-heavy workload respectively and then decreases for both cases to 500 operations per second.

In the dynamic benchmark runs the decrease in throughput after each configuration change is in accordance with the values measured for those configurations in the static run. An exemplary visualization of the write-heavy dynamic run is shown in Figure 6.6. Just before the change from C0 to C1 the throughput is at about 1000 operations per second which is the same as just before the change from C1 to C2. At the end of the run the throughput decreased to about 550 operations per second. There is a clear edge in the latency plot at the 600s and 1200s mark but there is no visible phase of disruption

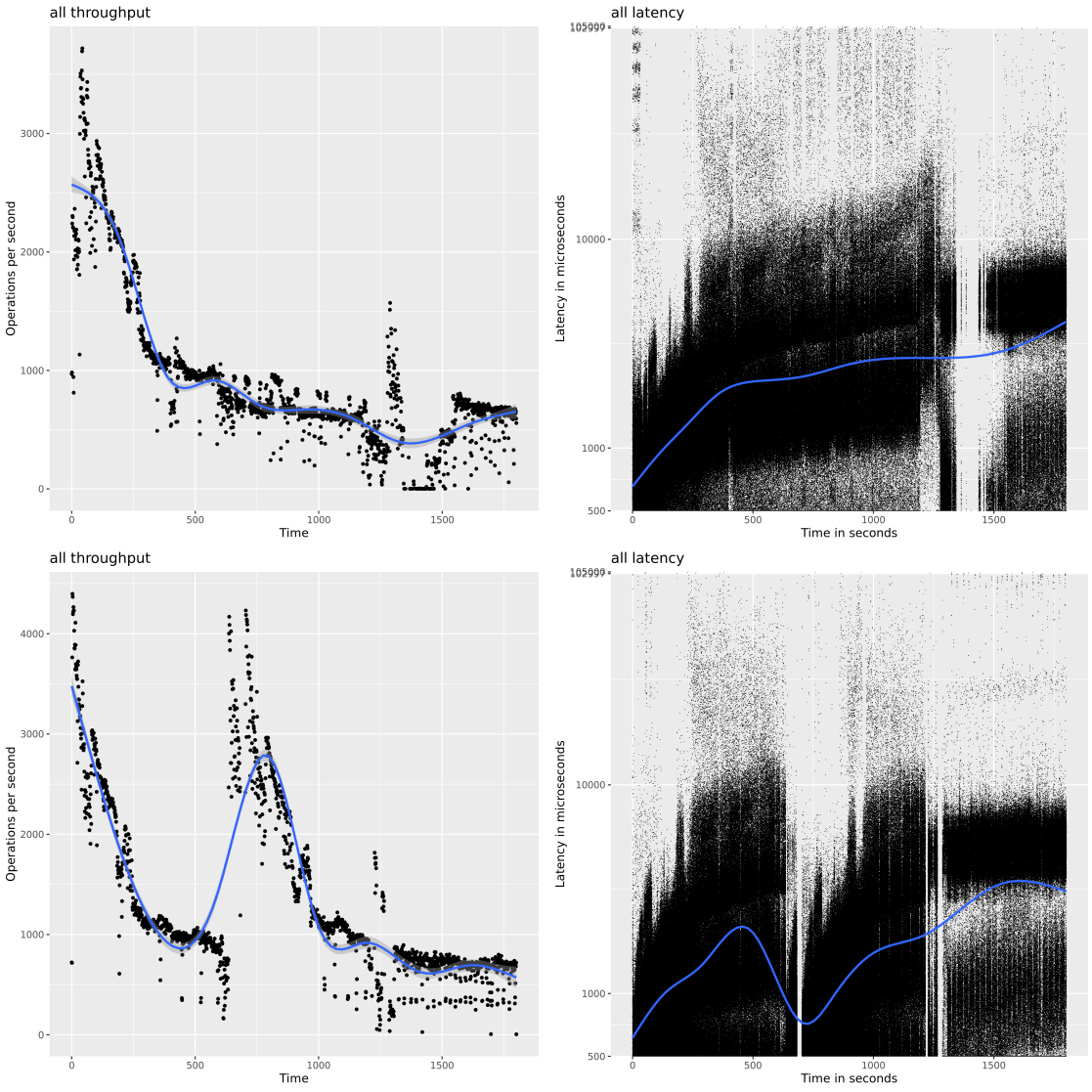


Figure 6.4.: Dynamic Runs with Consistent Hashing. There is a difference how cluster changes influence throughput and latency for read-heavy (top) and write-heavy (bottom) workloads.

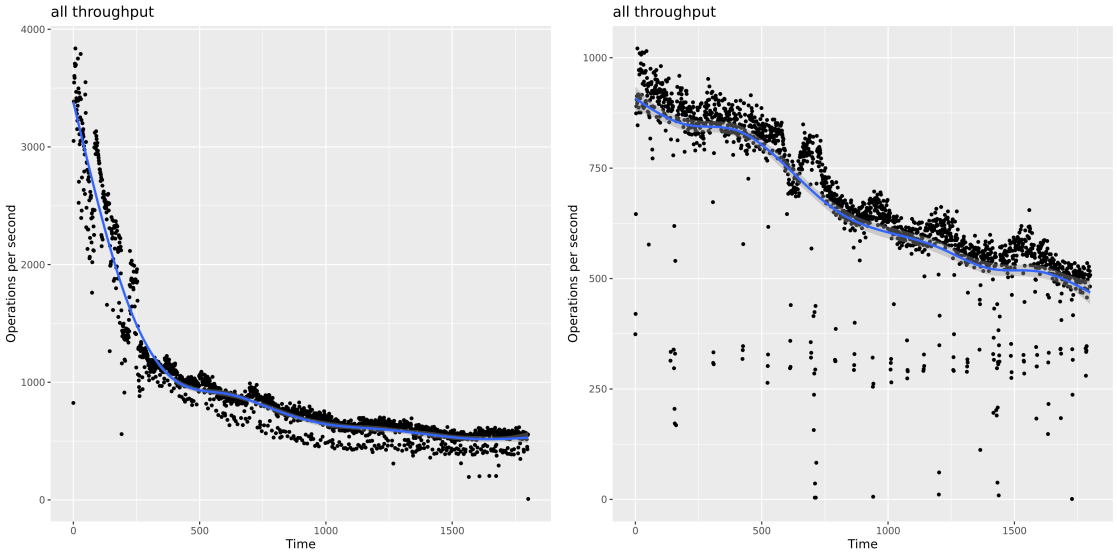


Figure 6.5.: RS with Ring Jumping Throughput Comparison. Showing the throughput for RS with Ring Jumping in C0 and C2 with write-heavy workload.

which implies that there is not a long phase of handoff operations.

Regarding the divergence from optimal load balance with RS and Ring Jumping there is an absolute overall average divergence for key load of 0.0303095841515983, for put load of 0.036558287305248, and for get load of 0.03657058797636. Ignoring C0 with exactly three sections the highest divergence is measured in the dynamic configuration and the lowest is measured in C1. This is quite intuitive as in C1 there are not many section and the sections are of a reasonable size while in the dynamic run the keys are potentially to the wrong node initially and will only be balanced after the repair operations are started for each key.

6.4.4. Random Slicing Random

Apart from one abnormal run there are no significant differences between read-heavy and write-heavy runs with respect to the throughput but a slightly higher rate for read-heavy workloads at the start. A comparison between the throuput in C0 and C2 is shown in Figure 6.8. In configuration C0 the throughput starts at about 2800 operations per second, steeply declines to 1000 operations per second at about 500s and then slowly decreases to 500 operations per second. The read-heavy run in C1 results in about the same throughput plot, however it ends at about 600 operations per second. The write-heavy run in C1 on the other hand shows an anomaly of a throughput spike at about 1300s and a sharp edge in the latency plot. Those anomalies can be seen in Figure 6.7. These are signs that imply a cluster change and handoff started at that time according to all dynamic runs. Assuming this is the case this run has to be excluded from the results. There is a clear decline in performance when looking at configuration C2 with seven as

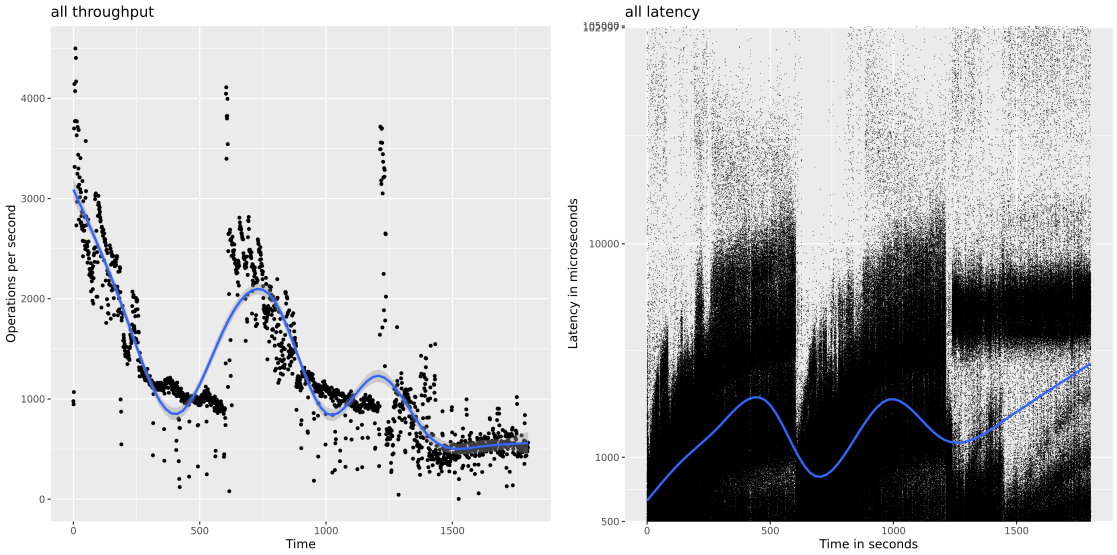


Figure 6.6.: Dynamic Runs with RS and Ring Jumping.

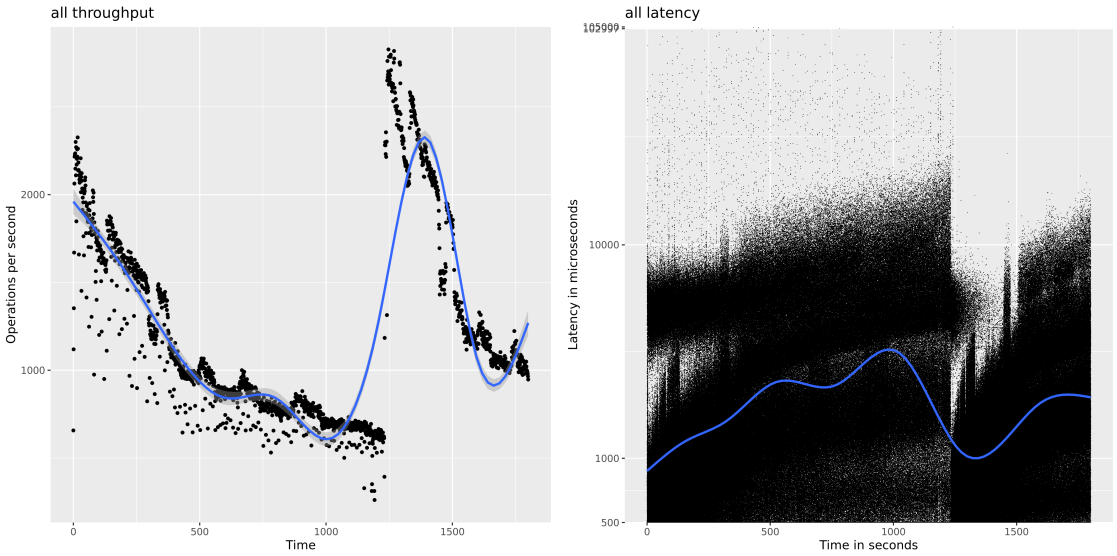


Figure 6.7.: RS with Ramdom Replication Anomaly. Even though the configuration is supposed to be static the throughput and latency plots show signs of a cluster change at 1300s.

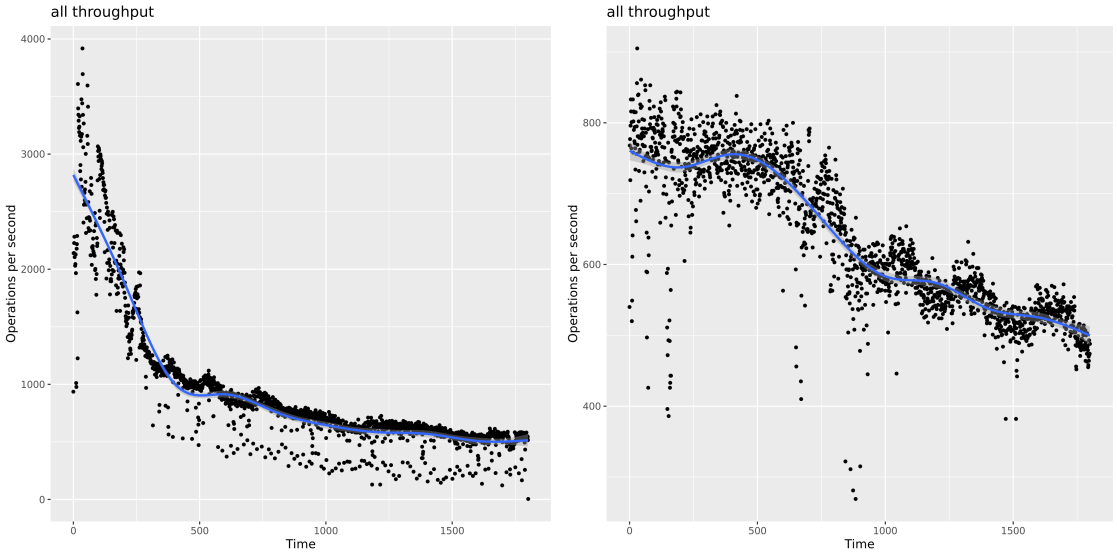


Figure 6.8.: RS with Ramdom Replication Throughput. Comparison of the throughput in C0 and C2.

the throughput starts at 750 operations per second and declines to 500 operations per second.

In the dynamic runs there seems to be a big difference in how the cluster changes are handled. While the throughput seems to be quite similar in the sense that the second cluster change seems to have no big influence on the throughput the latency shows quite different images. The latency plots of both workloads can be seen in Figure 6.9. For the read-heavy workload there is an initial disruption at 600s and 1200s followed by a few smaller disruptions. For the write-heavy workload there is a periodical occurrence of bigger disruptions starting at 600s and lasting up until 1000s. These periodic disruptions do not show up as strongly after the second cluster change.

Regarding the divergence from optimal load balance with RS and Ring Jumping there is an absolute overall average divergence for key load of 0.022889616527552, for put load of 0.027008232485792, and for get load of 0.027078968645263. Ignoring C0 with exactly three sections the highest divergence is measured in the dynamic configuration and the lowest is measured in C1. This is quite intuitive as in C1 there are not many section and the sections are of a reasonable size while in the dynamic run the keys are potentially to the wrong node initially and will only be balanced after the repair operations are started for each key.

6.4.5. Random Slicing Rotation

The Ring Rotation RPS implementation shows a lot of problems in the evaluation. Apart from C0 and C1 every run with other configurations was far off any expected result or could not even be finished. In C0 the throughput starts at about 3000 operations per

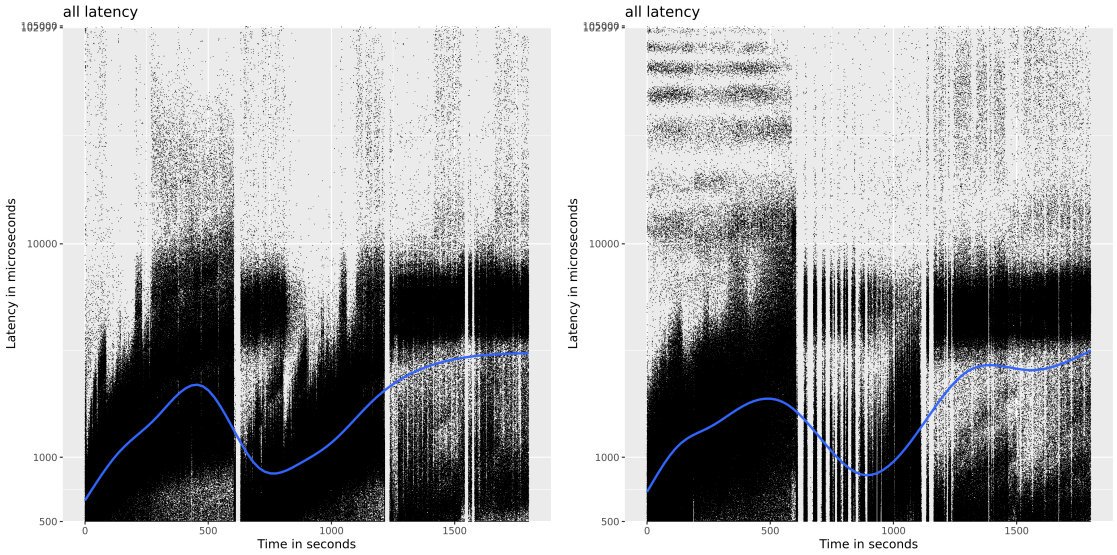


Figure 6.9.: RS with Ramdom Replication Latency. Comparison of the latency plots in the dynamic runs.

second, steeply decreases to 1000 operations per second at 450s and from there slowly decreases to 500 operations per second. In C1 the throughput starts at about 2100 operations per second, steeply decreases to 1000 operations per second at 500s and from there slowly decreases to 500 operations per second. The first problematic run is the write-heavy run in C2, as the throughput stays at single-digit operations per second over the complete run, which can be seen in Figure 6.10. Trying to rerun this configuration to confirm the result showed a potential memory leak that seems to be occur more frequently with a higher number of sections.

This memory leak became quite apparent as even after multiple restarts the benchmark runs could not be finished in the dynamic configurations as the memory leak caused the run to crash each time. An assertion on the overall load balance is also impossible through this.

6.4.6. Comparison

In this section we compare the evaluation results of the different RCL configurations and focus on how our hypotheses are supported or contradicted. As there is not enough data and configurations to properly test the hypotheses we use the available evaluation data to quantitatively compare the different approaches.

Hypothesis 1

Hypothesis 1 claims that RS has the best load balance when Random Replication is used as the replication placement strategy. As Random Replication produces the better

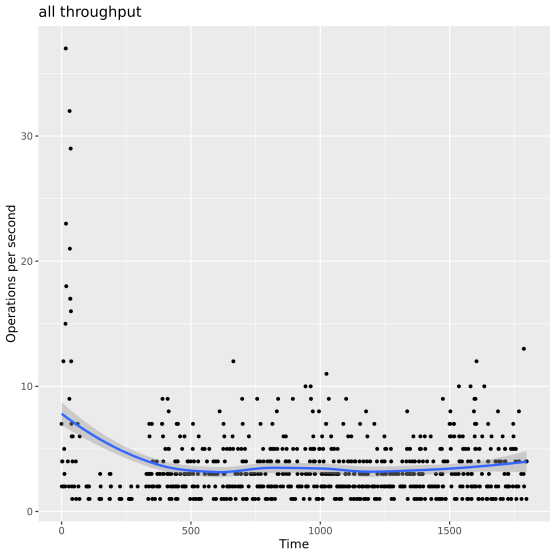


Figure 6.10.: Low Throughput with Ring Rotation

overall load balance and also has a lower absolute divergence for every configuration and parameter than Ring Rotation and Ring Jumping we can assume Hypothesis 1 as plausible.

Hypothesis 2

Hypothesis 2 claims that RS has the highest throughput when Random Replication is used as the replication placement strategy. Apart from the Ring Rotation strategy, which shows a strong decline in throughput with a growing number of ring sections, the throughput plots have the same characteristics for the different RPSs. They show that the throughput starts at about the same rate, decreases to similar values at equal times and also ends at the same rate. Therefore there is no argument for or against the hypothesis with the available data and as is shown with Ring Rotation it may be a matter of scalability of the strategies.

Hypothesis 3

Hypothesis 3 claims that the load divergence of RS with the best replication placement strategy is at most 10% higher than with Consistent Hashing. We use the results of RS with Random Replication as this yielded the best balancing properties and a high throughput. While the divergence of load concerning unique keys with RS is nearly 6 times the one with Consistent Hashing, the divergence concerning operations is only slightly more than 10% higher. However, this clearly shows strong arguments against the presented hypothesis for the current integration of RS with RCL. As a side remark it is interesting to see that RS with Consistent Hashing achieves a way better load balance

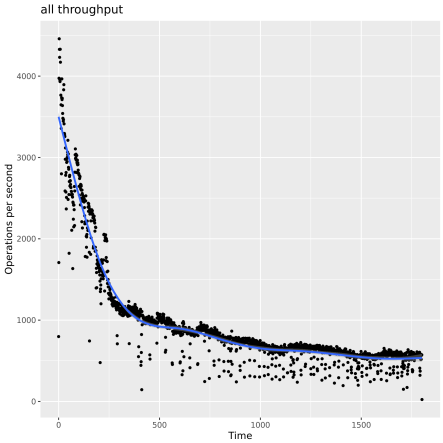
in configuration C2 than Consistent Hashing and it will be interesting to see how both systems scale for more nodes.

Hypothesis 4

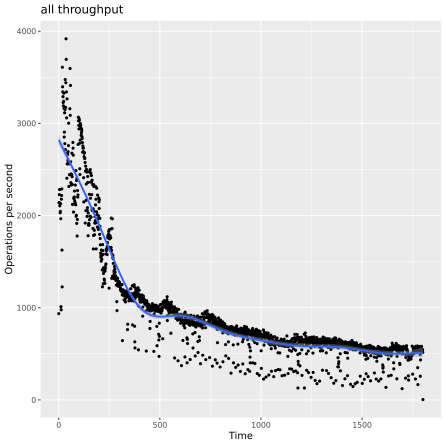
Hypothesis 4 claims that RS generates a throughput that is nearly as good as Consistent Hashing in a static cluster configuration. As there is no clear difference in throughput we choose Random Replication as the replication placement strategy. As can be seen in the side-to-side comparison in Figure 6.11 there is no significant difference between Consistent Hashing and RS in any static configuration. Therefore the hypothesis can be seen as plausible.

Hypothesis 5

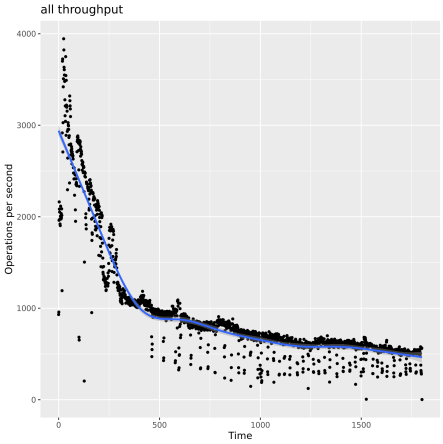
Hypothesis 5 claims that a system with RS will recover faster after a cluster change than with Consistent Hashing. Again we are using Random Replication as the replication placement strategy to enable a fair comparison. As the behavior of the throughput during handoff operations is quite unexpected it is hard to tell from its plot when a system has actually recovered. However, there are clear edges in the latency plots that can be used to approximate the duration of system disruption. The latency plots of the dynamic runs can be seen in Figure 6.12. One can easily see that for the read-heavy workload the disruptions with RS are way shorter than with Consistent Hashing. However, with the write-heavy workload there are multiple small disruptions with RS and less and shorter disruptions with Consistent Hashing. Overall we can make no assertion on the hypothesis with the available data as the results are currently ambiguous and there is a need for more and different kinds of cluster changes to derive a conclusion.



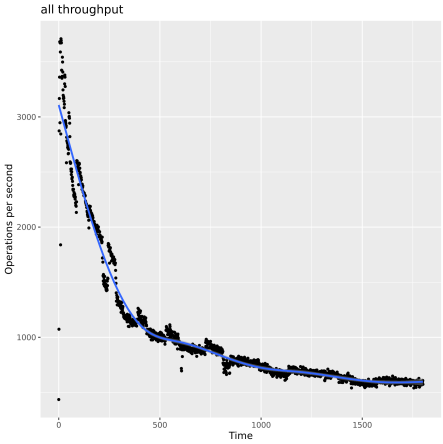
(a) Throughput for Consistent Hashing in C0



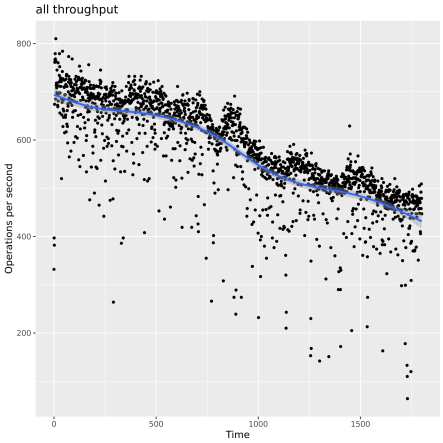
(b) Throughput for RS in C0



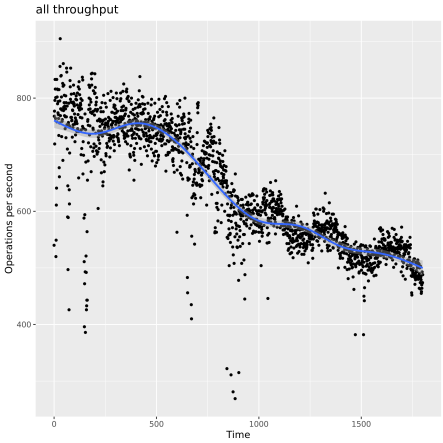
(c) Throughput for Consistent Hashing in C1



(d) Throughput for RS in C1

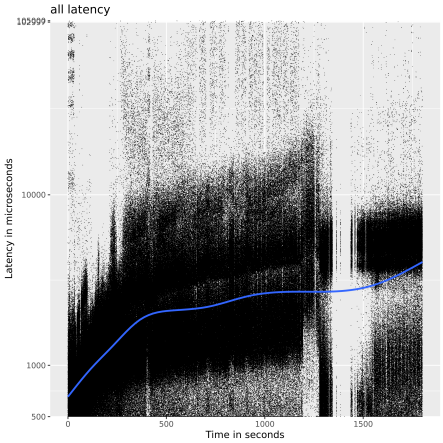


(e) Throughput for Consistent Hashing in C2

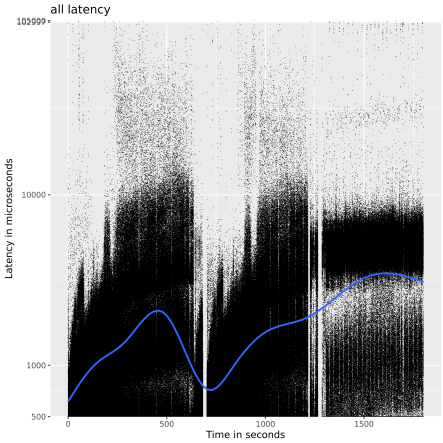


(f) Throughput for RS in C2

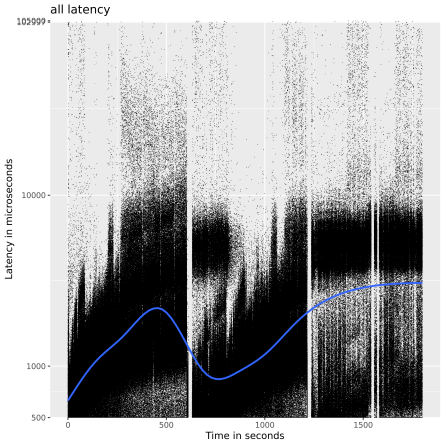
Figure 6.11.: Throughput Comparison. On the left are the throughput plots of Consistent Hashing, on the right the ones of RS.



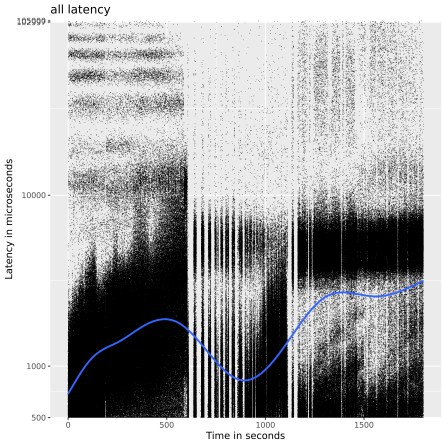
(a) Latency for Consistent Hashing with Read-Heavy Workload



(b) Latency for Consistent Hashing with Write-Heavy Workload



(c) Latency for RS with Read-Heavy Workload



(d) Latency for RS with Write-Heavy Workload

Figure 6.12.: Handoff Comparison. The hard edges in the latency plot can be used as an index to when the system is disrupted by handoff and repair operations.

7. Future Work

In this chapter we present future changes to the system that we see as necessary to either improve the system performance or make use of all features of RS. We will list each problem we identify in the current state of the system and then present potential solutions if we already thought of one.

7.1. Improve the Gap Collection Algorithm

The system currently uses a simple greedy gap collection algorithm. As Miranda et al. show in their evaluation of gap collection algorithms[16], there are alternatives that produce way less gaps without much computation overhead which in turn reduces lookup time. Especially the large difference for a low number of nodes has a strong impact on systems using Riak Core as they are not expected to scale to a high number of nodes. Since Riak Core is conceptualized for systems with infrequent cluster changes the lower lookup time will improve the system's throughput.

Another problem we noticed with the gap collection algorithm during the evaluation are some minuscule sections that are results from rounding errors when scaling the weight of the nodes. As those sections also increase lookup time it is necessary to prevent their creation. One possible solution is to implement a guard that prevents the creation of sections below a given size which would then also prevent a future feature of creating custom sections for special use cases. Another solution is to use rational numbers to scale the weights which has a higher memory usage but also higher precision.

7.2. Reintegrate Handoff Optimizations

RCL with Consistent Hashing differentiates between different kinds of handoffs to optimize other system procedures and prevent erroneous handoff operations that need to be fixed later. For example there are different handoff procedures for a simple cluster change and a resize change. During the proof-of-concept integration of RS with RCL only a basic handoff workflow was realized. There are most likely still many corner-cases of cluster changes that could lead to either a failure during handoff or a bad performance during handoff. As in principle all cluster changes with RS lead to a resize handoff the next step is to closely look at how resizing impacts RCL with Consistent Hashing and if those corner cases may work with RS. If this is the one can simply adapt the resize handoff for RS. Otherwise the handoff workflow needs to be redesigned completely to handle the new cases brought in by RS.

7.3. Develop a More Refined RPS

To offer the same functionality with RS as with Consistent Hashing, we need to develop a RPS. We presented three simple strategies, Random Replication, Ring Jumping, and Ring Rotation. The evaluation has shown that none of these is better than replication with Consistent Hashing and show weaknesses when the cluster changes. Therefore a simple RPS may not be feasible with RS a more sophisticated approach needs to be developed. Fritchie[8] proposed to include a level of indirection and map clusters of nodes with a replication strategy to the sections of the ring. However this then creates the problem of forming these clusters and balancing them, unless we change Riak Core's functionality and leave the replication completely to the implementing system.

7.4. Integrate the Replication Placement Strategy with the Preference List Creation

With Consistent Hashing a complete preference list consisting of all nodes starting at the one owning the key is computed and then split into primary and fallback nodes. The final preference list is then created by filtering the primary nodes by the ones that are up and inserting fallback nodes when necessary. The current implementation of the preference list creation with RS uses the RPS to compute a complete preference list and afterwards filters it in the same way as with Consistent Hashing. As in most cases not all nodes in the complete preference list are needed calculating the complete preference list creates an overhead. This overhead is non-existent with consistent hashing as the preference list is simply the underlying data structure of the ring split at the initial node. With the RPSs we presented the overhead potentially grows with each additional node that is needed to be appended to the preference list. Therefore we need to interlace the filtering of the preference list with its computation. An easy way to achieve this is to change the interface of the replication strategies to offer an iterator-like operation that only computes and returns the next entry of the list. Instead of filtering the complete preference list we can check the status of each node and poll more nodes if we need to use fallback nodes.

7.5. Support Heterogeneous Nodes

One of the strengths of RS over Consistent Hashing is its native support of heterogeneous nodes. While Amazon's Dynamo implementation[6] offers a support of heterogeneous nodes by assigning each node a different number of sections on the ring this approach can only approximate a balanced distribution of heterogeneous nodes. With RS a more precise balancing of heterogeneous nodes can be achieved. Currently the support for weights assigned to nodes is implemented in the internal code of the ring component. However, there is no API to actually specify or change a node's weight. This seems to be a rather simple change to add a weight parameter to the join operation and implement a new update-weight operation and add it to the stage-plan-commit lifecycle.

7.6. Evaluate Higher Scalability

We only evaluated small cluster sizes with less than ten nodes as is typical for Riak Core applications. However, Miranda et al. introduce RS as a solution for highly scalable replication[16]. Therefore it is interesting to see how in RCL Consistent Hashing fares against RS when scaling the cluster size to higher values and if it actually enables more use cases as there is no ring size to be set and changed by an admin.

8. Conclusion

In this thesis we gave an overview on the inner workings of RCL and its Consistent Hashing implementation. Inspired by Fritchie's idea to replace the Consistent Hashing algorithm with RS we also presented how RS works. Seeing that Consistent Hashing directly allows for a simple and quick RPS we also presented a choice of simple RPSs. We then implemented a prototype version of RCL with RS as its data partitioning algorithm and gave an overview of how the structure and some workflows of RCL had to be changed. To compare both versions of RCL we designed configurations to illustrate the strengths and weaknesses of both systems. With those configuration we ran benchmarks to retrieve data. From that data we created a first impression of how RS works with RCL and what the big problems are with the current implementation.

This whole process has shown that the choice of the partitioning algorithm in RCL is rather an architectural choice than a design one as large parts of the system depend on the inner workings of this algorithm. In turn, simply replacing the partitioning component and redesigning components directly connected to it is not enough. One has to rather go back to the architectural phase and reevaluate the design decisions depending on the chosen partitioning algorithm and its requirements, guarantees and other attributes. Considering RS was adopted for RCL in a prototype manner without a full-scope design process the evaluation results look promising. For the tested aspects it produced metrics that were at most slightly worse than RCL with Consistent Hashing. Especially problems during handoff periods and with load balancing can be clearly overcome by a more refined development process in the future. Overall, we showed that RS works as a partitioning algorithm, though it is reliant on its actual gap collection algorithm and replica placement strategy. We could not show all of the theoretical improvements listed by Fritchie, especially the improvements of the handoff overhead. However, this was mostly caused by the prototype implementation and missing metrics in the evaluation. We strongly believe that with a full redesign of affected components and considering our proposed future work RS can improve RCL's overall performance and especially its scalability and adaptability in more dynamic clusters.

Bibliography

[1] Basho Technologies. *Riak*. Jan. 19, 2021. URL: `https://github.com/basho/riak` (visited on 01/19/2021).

[2] Basho Technologies. *Riak Core*. Jan. 19, 2021. URL: `https://github.com/basho/riak_core` (visited on 01/19/2021).

[3] David Blackman and Sebastiano Vigna. "Scrambled linear pseudorandom number generators". In: *arXiv preprint arXiv:1805.01407* (2018).

[4] A. Brinkmann, S. Effert, and F. Meyer auf der Heide. "Dynamic and Redundant Data Placement". In: *27th International Conference on Distributed Computing Systems (ICDCS '07)*. IEEE, 2007. DOI: `10.1109/icdcs.2007.103`.

[5] Brian F Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.

[6] Giuseppe DeCandia et al. "Dynamo". In: *ACM SIGOPS Operating Systems Review* 41.6 (Oct. 2007), pp. 205–220. DOI: `10.1145/1323293.1294281`.

[7] D Eastlake 3rd and Paul Jones. *Rfc3174: Us secure hash algorithm 1 (sha1)*. 2001.

[8] Scott Lystig Fritchie. *A Critique of Resizable Hash Tables: Riak Core & Random Slicing*. Aug. 26, 2018. URL: `https://www.infoq.com/articles/dynamo-riak-random-slicing/` (visited on 06/03/2020).

[9] Scott Lystig Fritchie. "Chain replication in theory and in practice". In: *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*. 2010, pp. 33–44.

[10] Hibari Developers. *Hibari DB*. 2015. URL: `https://hibari.readthedocs.io/en/latest/index.html` (visited on 07/06/2020).

[11] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 9. 2010.

[12] Nidhi Jain Kansal and Inderveer Chana. "Cloud load balancing techniques: A step towards green computing". In: *IJCSI International Journal of Computer Science Issues* 9.1 (2012), pp. 238–246.

[13] David Karger et al. "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*. 1997, pp. 654–663.

[14] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[15] Najme Mansouri. "Adaptive data replication strategy in cloud computing for performance improvement". In: *Frontiers of Computer Science* 10.5 (June 2016), pp. 925–935. DOI: 10.1007/s11704-016-5182-6.

[16] Alberto Miranda et al. "Random Slicing". In: *ACM Transactions on Storage* 10.3 (July 2014), pp. 1–35. DOI: 10.1145/2632230.

[17] Riak. *Introducing Riak Core*. July 30, 2011. URL: https://riak.com/posts/business/introducing-riak-core/index.html?p=6033.html (visited on 01/19/2021).

[18] Riak Core Lite Team. *Riak Core Lite*. 2020. URL: https://riak-core-lite.github.io/index.html (visited on 11/18/2020).

[19] I. Stoica et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications". In: *IEEE/ACM Transactions on Networking* 11.1 (Feb. 2003), pp. 17–32. DOI: 10.1109/tnet.2002.808407.

[20] The Apache Software Foundation. *Apache Cassandra*. 2016. URL: https://cassandra.apache.org/ (visited on 07/16/2020).

[21] Robbert Van Renesse and Fred B Schneider. "Chain Replication for Supporting High Throughput and Availability." In: *OSDI*. Vol. 4. 91–104. 2004.

[22] Lizhe Wang et al. "Cloud computing: a perspective study". In: *New Generation Computing* 28.2 (2010), pp. 137–146.

List of Figures

3.1. Consistent Hashing Example 11

4.1. Random Slicing example 20
4.2. Visualization of the ring rotation replication 25
4.3. Ring Jumping 27

5.1. Side by Side Comparison of Consistent Hashing and Random Slicing . . . 30
5.2. Handoff Ring 32

6.1. Throughput Anomaly 38
6.2. Handoff Anomaly 39
6.3. Consistent Hashing Throughput Comparison 40
6.4. Dynamic Runs with Consistent Hashing 41
6.5. RS with Ring Jumping Throughput Comparison 42
6.6. Dynamic Run with RS and Ring Jumping 43
6.7. RS with Ramdom Replication Anomaly 43
6.8. RS with Ramdom Replication Throughput 44
6.9. RS with Ramdom Replication Latency 45
6.10. Low Throughput with Ring Rotation 46
6.11. Throughput Comparison 48
6.12. Handoff Comparison 49

A.1. Send Ring 59
A.2. Transform Ring (alt = new_ring | reconciled_ring | ignore) 59
A.3. Check Tainted 60
A.4. Ring Changed 61
A.5. Refresh Ring 61
A.6. Reconcile 62
A.7. Set Ring 63
A.8. Do Claimant 63
A.9. Compute Resize 64
A.10. Plan 66
A.11. Apply Changes 67
A.12. Update Ring Resizing 69
A.13. Commit Staged 70
A.14. Start Cluster 71
A.15. Handle Joining 72
A.16. Commit 72

A.17.Start System Processes . . . 73
A.18.Initialize Handoff Components . . . 75
A.19.Initialize VNode Manager . . . 76
A.20.Join Cluster . . . 78
A.21.Set My Ring . . . 80
A.22.Leave Cluster . . . 80
A.23.Remove Node . . . 81
A.24.Find Index for Key . . . 82
A.25.Find Indices for Node . . . 83
A.26.Find Node for Index . . . 83
A.27.Preference List . . . 84
A.28.Find Index for Key . . . 85
A.29.Find Partition Index for Key . . . 85
A.30.Find Ring Index for Key . . . 86
A.31.Preference List . . . 87
A.32.Get APL Locally . . . 88
A.33.Get APL from Ring . . . 88
A.34.Get Annotated APL from Ring . . . 88
A.35.Get Annotated APL Locally . . . 89
A.36.Get Primary APL Locally . . . 89
A.37.Get Primary APL from Ring . . . 90
A.38.Get Primary APL from Binary . . . 90
A.39.Get APL from Binary . . . 91
A.40.Get Annotated APL from Binary . . . 91
A.41.Resize Ring . . . 92
A.42.Handoff . . . 92
A.43.Send Handoff . . . 93

B.1. Evaluation Data for ConsistentHashing_C0 . . . 95
B.2. Evaluation Data for ConsistentHashing_C1 . . . 96
B.3. Evaluation Data for ConsistentHashing_C2 . . . 97
B.4. Evaluation Data for ConsistentHashing_dynamic . . . 98
B.5. Evaluation Data for RandomSlicing_Jumping_C0 . . . 99
B.6. Evaluation Data for RandomSlicing_Jumping_C1 . . . 100
B.7. Evaluation Data for RandomSlicing_Jumping_C2 . . . 101
B.8. Evaluation Data for RandomSlicing_Jumping_dynamic . . . 102
B.9. Evaluation Data for RandomSlicing_Random_C0 . . . 103
B.10.Evaluation Data for RandomSlicing_Random_C1 . . . 104
B.11.Evaluation Data for RandomSlicing_Random_C2 . . . 105
B.12.Evaluation Data for RandomSlicing_Random_dynamic . . . 106
B.13.Evaluation Data for RandomSlicing_Rotation_C0 . . . 107
B.14.Evaluation Data for RandomSlicing_Rotation_C1 . . . 108
B.15.Evaluation Data for RandomSlicing_Rotation_C2 . . . 109
B.16.Evaluation Data for RandomSlicing_Rotation_dynamic . . . 110

A. Actions

A.1. General Actions

1 If the sending node owns gossip tokens it sends its ring to another node where both rings are reconciled.

1.1 Retrieve the raw ring that is to be sent out.

1.2 *<<action>>* See Action A.3.

1.3 *<<rpc>>* Reconcile the received ring with the local ring.

1.3.1 *<<action>>* Apply the reconcile callback to the local ring. See Action A.2.

1.3.1.1 *<<callback>><<action>>* See Action A.6.

Action A.1.: Send Ring, see Figure A.1

1 Transforms the current ring with the given callback. Depending on the callback the new state is gossiped randomly or deterministic or only set locally.

1.1 *<<callback>>* The transformation callback that is applied to the ring. The result tuple contains the alt-value.

1.2 *<<action>>* See Action A.3.

1.3 *<<action>>* Sets the new ring in the ETS table. See Action A.7.

1.4 Updates the meta entry and its timestamp if the value differs.

1.5 [ignored for alt=set_only] Applies the update callback to the current ring.

1.6 [alt=new_ring → random; alt=set_only → ignore; alt=reconciled_ring → not random] Gossips the state to the children of a given node. For random recursive gossip the gossiping node is chosen at random.

1.6.1 Retrieves all members whose status is joining, valid, leaving, or exiting.

1.6.2 [for each active member]*<<action>>* See Action A.1.

Action A.2.: Transform Ring (alt = new_ring | reconciled_ring | ignore), see Figure A.2

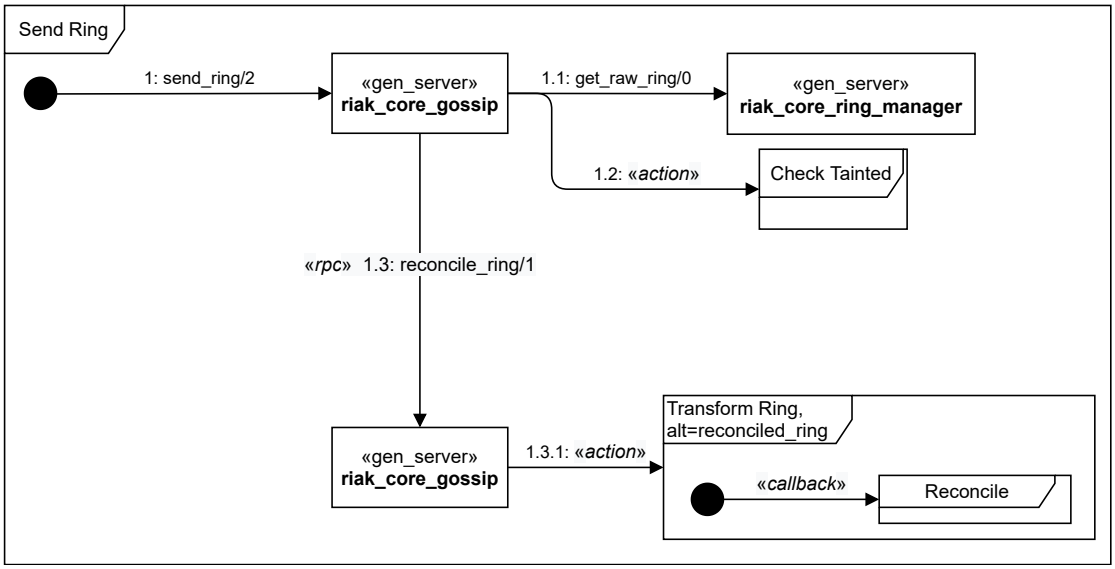


Figure A.1.: Send Ring, see Action A.1

.

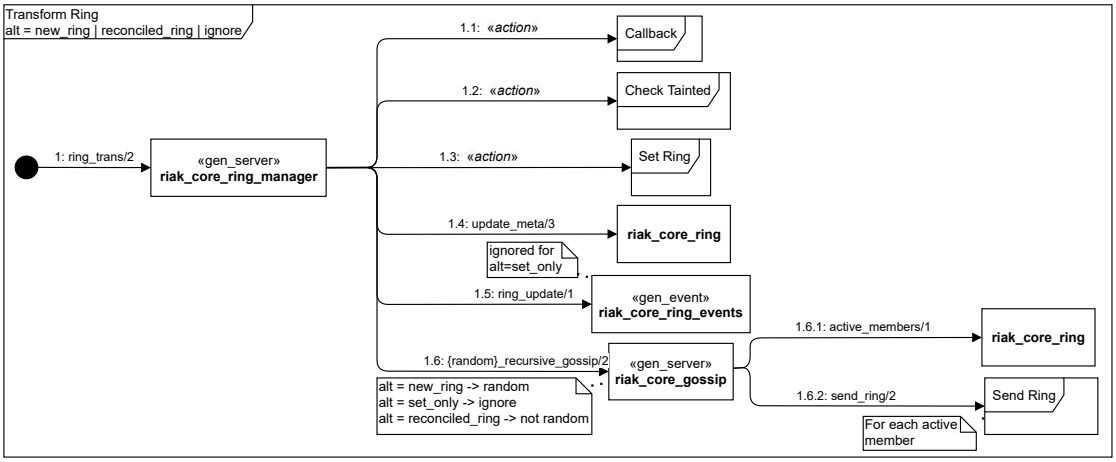


Figure A.2.: Transform Ring (alt = new_ring | reconciled_ring | ignore), see Action A.2

.

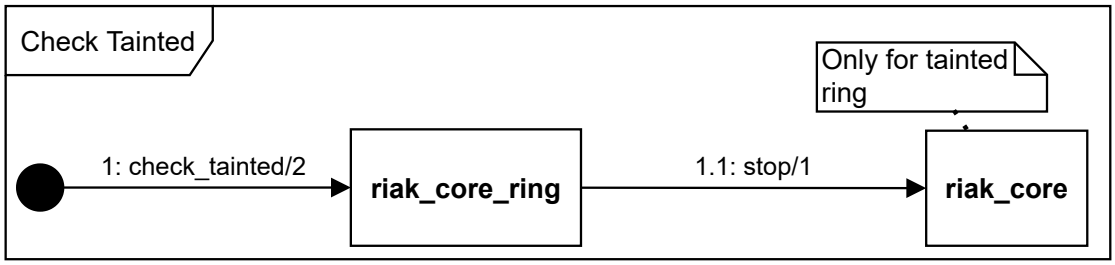


Figure A.3.: Check Tainted, see Action A.3

.

1 Checks if the tainted-flag is set in the meta data. Either stops the application with a given error message or just logs the error message when the ring is tainted.

1.1 [If ring is tainted] Stops the application and logs the given reason.

Action A.3.: Check Tainted, see Figure A.3

1 Applies changes to the ring and informs other nodes.

1.1 Check for joining members.

1.2 Check for valid and leaving members.

1.3 Retrieve claimant to check if it is a valid member.

1.4 If the claimant is missing set the next claimant.

1.5 Update the ring version if the claimant was changed.

1.6 Find joining members.

1.7 Retrieve meta to filter for auto joining nodes.

1.8 Retrieve current claimant.

1.9 Set all joining nodes as valid members.

1.10 Retrieve current cluster name.

1.11 Retrieve exiting members.

1.12 Retrieve invalid members.

1.13 <<*action*>> Inform each exiting node and refresh their ring to a fresh ring only containing themselves. See Action A.5

1.14 Retrieve current claimant to check if the local node is the claimant.

1.15 Retrieve pending changes from old state.

1.16 Retrieve pending changes from changed state.

1.17 Find out if the cluster name is set.

1.18 <<*rpc*>> If the cluster name is not set, set the cluster name for every member.

Action A.4.: Ring Changed, see Figure A.4

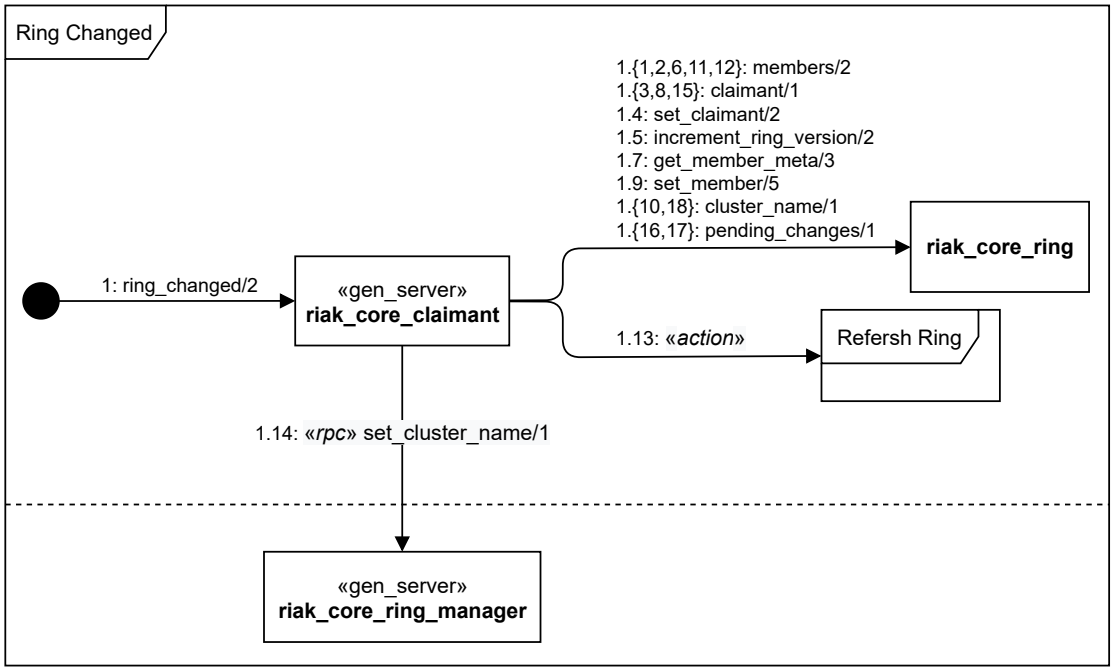


Figure A.4.: Ring Changed, see Action A.4

.

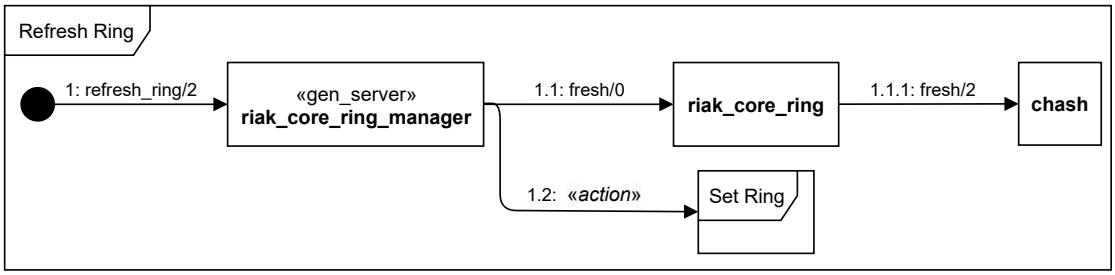


Figure A.5.: Refresh Ring, see Action A.5

.

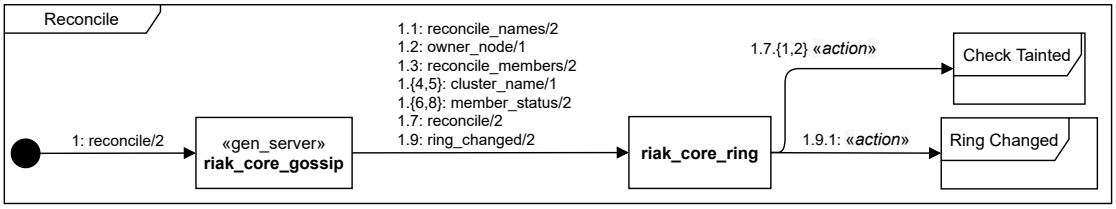


Figure A.6.: Reconcile, see Action A.6

.

1 Refreshes the ring on the given node to a ring containing only that node and puts the result into the ETS table and ring file.

1.1 Create a new ring state.

1.1.1 Create a new chash structure with the given node as the owner.

1.2 *<<action>>* Sets the ring in the ETS table.

Action A.5.: Refresh Ring, see Figure A.5

1 Reconciles two diverged rings into one.

1.1 If one name is undefined the result is undefined, else a term consisting of both names is returned.

1.2 Retrieve the owner of the other ring.

1.3 Reconciles members and their status of both rings using vector clocks.

1.4 Get first ring's name for comparison.

1.5 Get other ring's name for comparison.

1.6 Get status of other ring's owner before reconciliation.

1.7 Merges all ring attributes.

1.7.1 *<<action>>* Check other ring tainted. See Action A.3.

1.7.2 *<<action>>* Check my ring tainted. See Action A.3.

1.8 Get status of other ring's owner in reconciled ring.

1.9 Apply changes to the ring.

1.9.1 *<<action>>* See Action A.3.

1.9.2 *<<action>>* Apply changes to the ring. See Action A.4.

Action A.6.: Reconcile, see Figure A.6

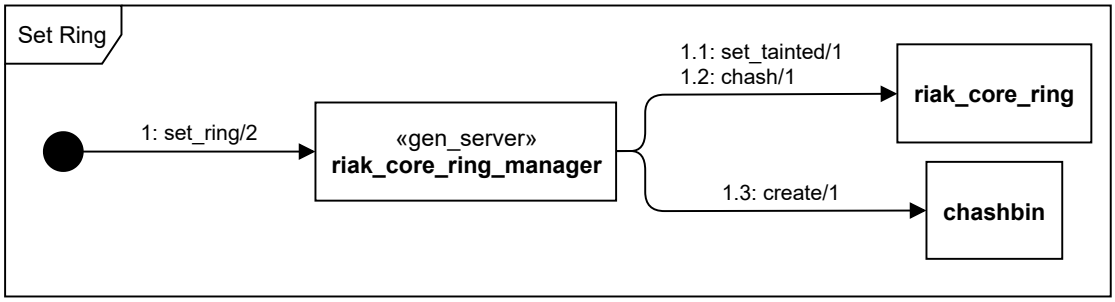


Figure A.7.: Set Ring, see Action A.7

.

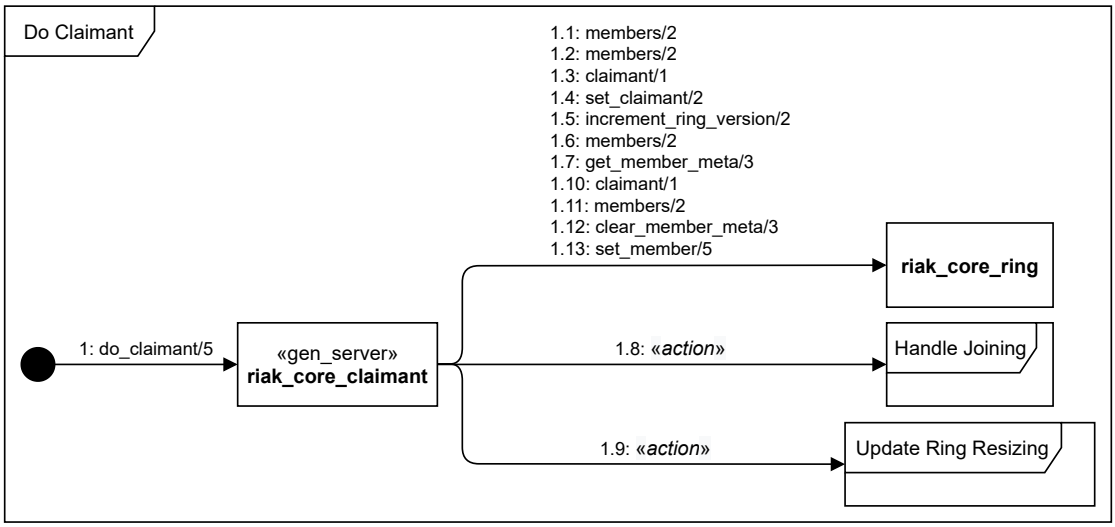


Figure A.8.: Do Claimant, see Action A.8

.

1 Set the ring in the ETS table.
1.1 Set the ring as tainted to check for leaks over gossip.
1.2 Get the chash from the ring.
1.3 Compute the chashbin from the ring.

Action A.7.: Set Ring, see Figure A.7

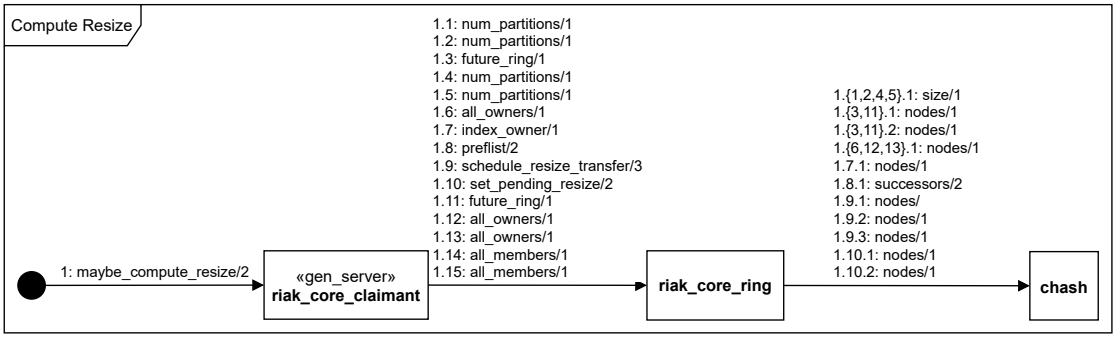


Figure A.9.: Compute Resize, see Action A.9

.

1 Handle joining nodes and add them to the ring. If there are no joining nodes apply transfers to the ring. Finally handle nodes marked as exiting and remove them from the ring.

- 1.1 Check if there are any members joining the ring.
- 1.2 Get all active members on the ring.
- 1.3 Retrieve the current claimant.
- 1.4 Set the next claimant in the list if it is missing.
- 1.5 Increment the ring version after the claimant update.
- 1.6 Get all members of the ring.
- 1.7 Filter members for auto-joining nodes.
- 1.8 *<<action>>* Handle all of the auto-joining nodes. See Action A.15.
- 1.9 *<<action>>* Update the ring with the changes planned by the claimant. See Action A.12.
- 1.10 Retrieve current claimant and check if the local node is the claimant.
- 1.11 Retrieve all members marked as exiting.
- 1.12 Remove meta for exiting nodes.
- 1.13 Set exiting nodes as invalid members.

Action A.8.: Do Claimant, see Figure A.8

1 Compute the resized ring and schedule transfers. After this validate the future ring by comparing owners and members.
  1.1 Retrieve size of the original ring.
    1.1.1 Retrieve number of partitions.
  1.2 Retrieve size of the resized ring.
    1.2.1 Retrieve number of partitions.
  1.3 Compute ring resulting from applying pending changes to the resized ring.
    1.3.1 Retrieve all member nodes.
    1.3.2 Retrieve all member nodes.
  1.4 Check size of the original ring for comparison.
    1.4.1 Retrieve number of partitions.
  1.5 Check size of the resized ring for comparison.
    1.5.1 Retrieve number of partitions.
  1.6 Retrieve all owners of the original ring.
  1.7 Retrieve the owner of an index on the resized ring to schedule a transfer from the original to the future owner.
    1.7.1 Retrieve all owners.
  1.8 Compute a list of responsible nodes for a given index.
    1.8.1 Retrieve all successors wrapping around the ring.
  1.9 Determine the first resize transfer a partition should perform with the goal of ensuring the transfer will actually have data to send to the target.
    1.9.1 Retrieve all owners to compute an index owner of the future ring.
    1.9.2 Retrieve all owners to compute an index owner.
    1.9.3 Retrieve all owners to schedule a resize transfer.
  1.10 Schedule a resize operation from the original ring to the ring after resizing and transferring nodes.
    1.10.1 Retrieve members of the original ring.
    1.10.2 Retrieve members of the next ring.
  1.11 Retrieve new future ring for validation.
    1.11.1 Retrieve all owners of the next ring.
    1.11.2 Retrieve all owners to get a list of indices.
  1.12 Retrieve all original owners.
    1.12.1 Retrieve all owners.
  1.13 Retrieve all future owners.
    1.13.1 Retrieve all owners.
  1.14 Retrieve all original members.
  1.15 Retrieve all future members.

Action A.9.: Compute Resize, see Figure A.9

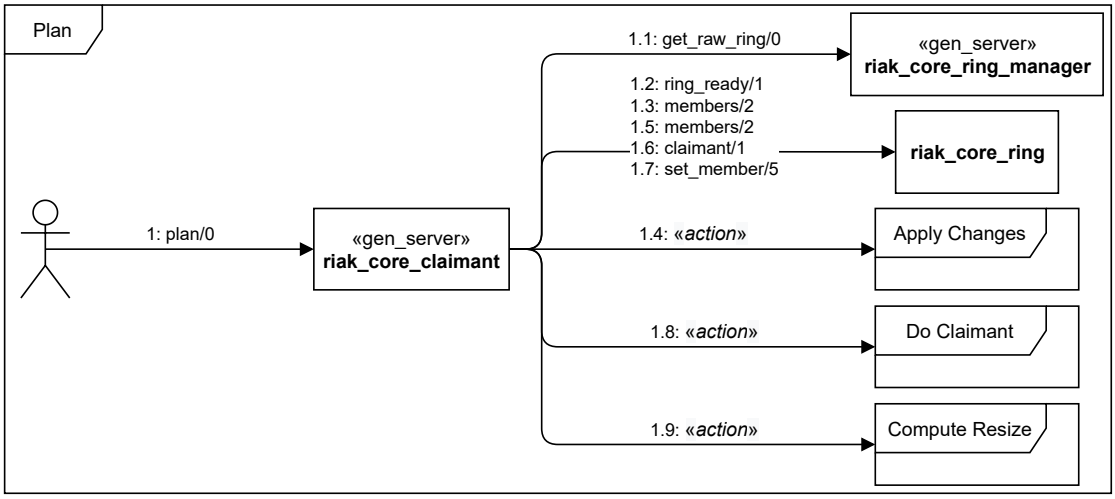


Figure A.10.: Plan, see Action A.10

.

1 Compute the ring resulting when applying the pending changes and set it as the provisional ring to check against when actually committing changes.

1.1 Retrieve the currently installed ring.

1.2 Check if the ring is ready for changes.

1.3 Retrieve joining members.

1.4 *<<action>>* Apply the pending changes to the ring. See Action A.11.

1.5 Retrieve joining nodes.

1.6 Retrieve current claimant id.

1.7 Set the joining members to valid.

1.8 *<<action>>* Update the ring according to the pending changes and set transfer markers. See Action A.8.

1.9 *<<action>>* Compute the ring ownership after a resize and schedule transfers. See Action A.9.

Action A.10.: Plan, see Figure A.10

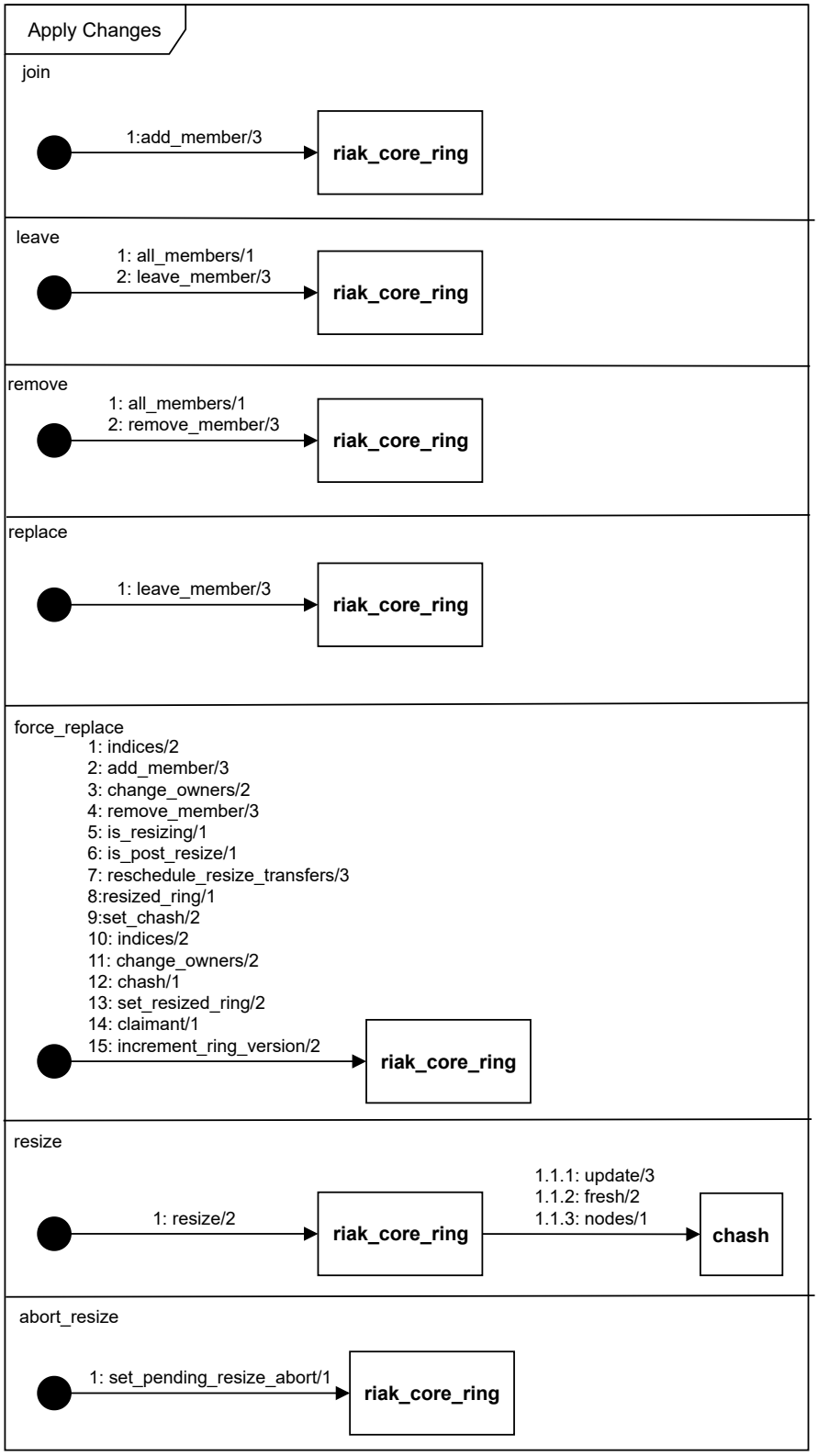


Figure A.11.: Apply Changes, see Action A.11

join

1 Add the node to the member list as joining.

leave

1 Retrieve the member list to check if the leaving node is a member.

2 Mark the node as leaving.

remove

1 Retrieve the member list to check if the node to be removed is a member.

2 Mark the node as removed.

replace

1 Mark the replaced member as leaving. Replacement happens when the claimant updates the ring the next time.

force_replace

1 Get all indices on the ring.

2 Add the replacing node.

3 Transfer ownership from the replaced to the replacing node.

4 Remove the replaced member.

5 Check if there is currently a resize operation ongoing.

6 Check if the resize operation has finished.

7 Abort the ongoing resize transfers to enable the replacement.

8 Compute the ring after the resize.

9 Assign the chash structure of the resized ring to the current ring.

10 Get a list of indices in the resized ring.

11 Replace nodes in the resized ring.

12 Retrieve the chash structure after reassigning indices.

13 Set the chash structure in the original ring as resized.

14 Retrieve the claimant of the cluster.

15 Update the ring version of the ring after the resize.

resize

1 Start the resize operation on the ring.

1.1 Reassign a section.

1.2 Create a new chash structure with the new size.

1.3 Retrieve all members and check for consistency with the cluster state.

abort_resize

1 Abort an ongoing resize operation.

Action A.11.: Apply Changes, see Figure A.11

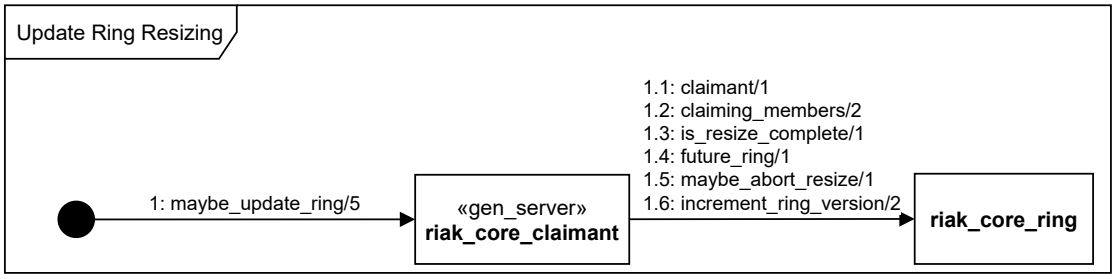


Figure A.12.: Update Ring Resizing, see Action A.12

.

1 Install the future ring after an resizing operation is completed.
- 1.1 Check if the local node is the claimant.
- 1.2 Check if there are any nodes claiming partitions.
- 1.3 Check if an ongoing resize operation is completed.
- 1.4 Retrieve the ring resulting from applying all pending changes.
  - 1.4.1 Retrieve all owners of the next ring.
  - 1.4.2 Retrieve all owners to get a list of indices.
- 1.5 Abort an ongoing resize operation if it is scheduled to be aborted.
- 1.6 Increment the version number of the ring.

Action A.12.: Update Ring Resizing, see Figure A.12

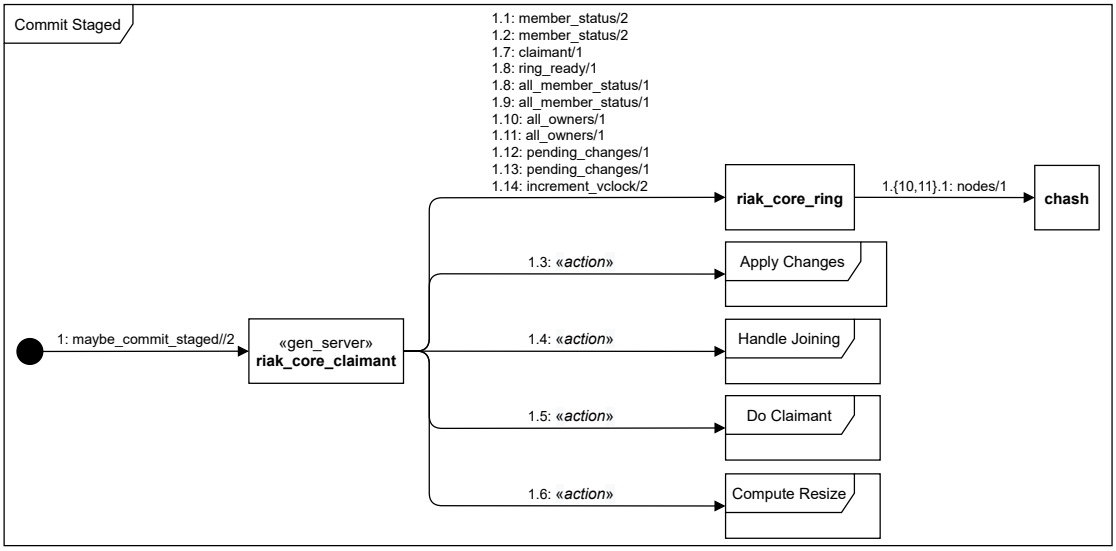


Figure A.13.: Commit Staged, see Action A.13

.

1 Commit all staged changes.

1.1 Check that a node contained in a change is not invalid.

1.2 In case of a replacement change check that the replacing node is joining.

1.3 ≪*action*≫ Apply all filtered changes to the current ring. See Action A.11.

1.4 ≪*action*≫ Mark joining nodes as valid. See Action A.15.

1.5 ≪*action*≫ Compute necessary transfers between current and next ring. See Action A.8.

1.6 ≪*action*≫ Compute the resized ring. See Action A.9.

1.7 Retrieve the current claimant to check if it is the local node.

1.8 Check if the ring is ready for incoming operations.

1.9 Get member status of the planned ring.

1.10 Get member status of the next ring and compare it to the planned ring.

1.11 Get all owners of the planned ring.

1.11.1 Retrieve all owners.

1.12 Get all owners of the next ring and compare it to the planned ring.

1.12.1 Retrieve all owners.

1.13 Get the pending changes of the planned ring.

1.14 Get the pending changes of the next ring and compare it to the planned ring.

1.15 If the changes can be committed increment the vector clock.

Action A.13.: Commit Staged, see Figure A.13

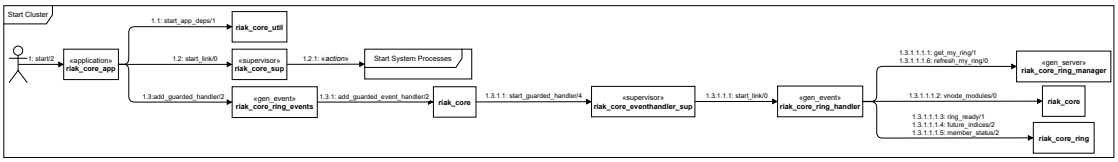


Figure A.14.: Start Cluster, see Action A.14

.

A.2. Start Cluster

1 is the entry point to start the application. It ensures all applications it depends on are started, reads or creates the ring state directory, starts the main supervisor and cascading all other supervisors, and adds the event handler to the riak core ring events.

- 1.1 Ensures that all applications listed in the dependencies are started and starts them if they are not.
- 1.2 Starts the riak core supervisor. Subsequently starts other supervisor processes.
  - 1.2.1 *≪action≫* Starts all processes whose supervisor tree roots in the riak core supervisor. See Action A.17.
- 1.3 Registers the riak core ring handler.
  - 1.3.1 Adds the event handler to the supervisor.
    - 1.3.1.1 Starts the supervised event handler as a child.
      - 1.3.1.1.1 Initializes the ring handler and starts vnodes if they are not yet started.
        - 1.3.1.1.1.1 Retrieves the current ring.
        - 1.3.1.1.1.2 Retrieves the modules with the vnode behavior.
        - 1.3.1.1.1.3 Checks if the ring is ready for operatins.
        - 1.3.1.1.1.4 Retrieves indices belonging to the local node in the future ring.
        - 1.3.1.1.1.5 Retrieves the local member status.
        - 1.3.1.1.1.6 If the local node is not on the ring, refresh it.

Action A.14.: Start Cluster, see Figure A.14

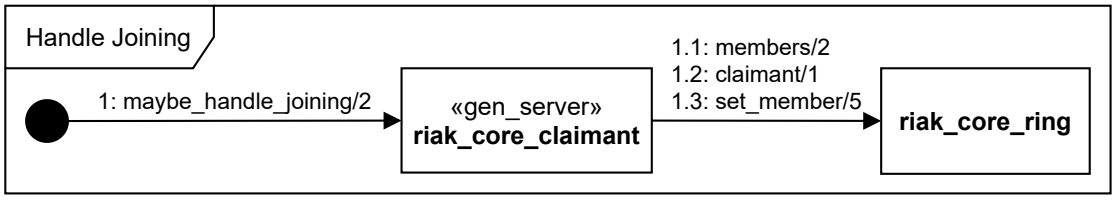


Figure A.15.: Handle Joining, see Action A.15

.

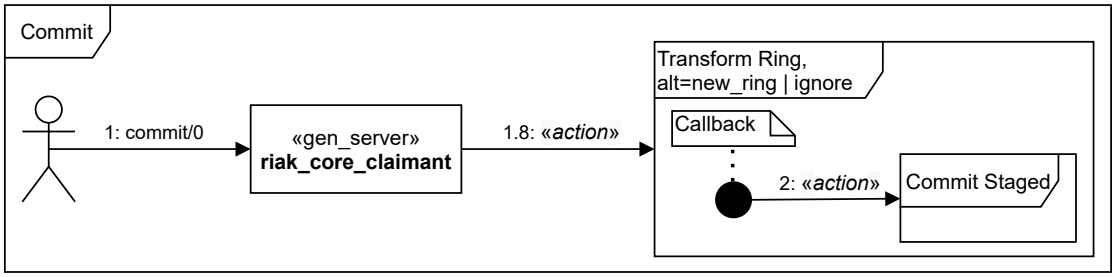


Figure A.16.: Commit, see Action A.16

.

1 Handle all nodes marked as joining and set them as a valid member.
1.1 If no list of joining nodes to handle is given, retrieve all joining members.
1.2 Retrieve claimant to check if the local node is the current claimant.
1.3 Set a joining node as valid member.

Action A.15.: Handle Joining, see Figure A.15

1 Process all scheduled changes, apply them to the currently installed ring and install the updated ring.
1.1 *<<action>>* Transform the ring with the given callback and gossip the transformed ring. See Action A.2.

Action A.16.: Commit, see Figure A.16

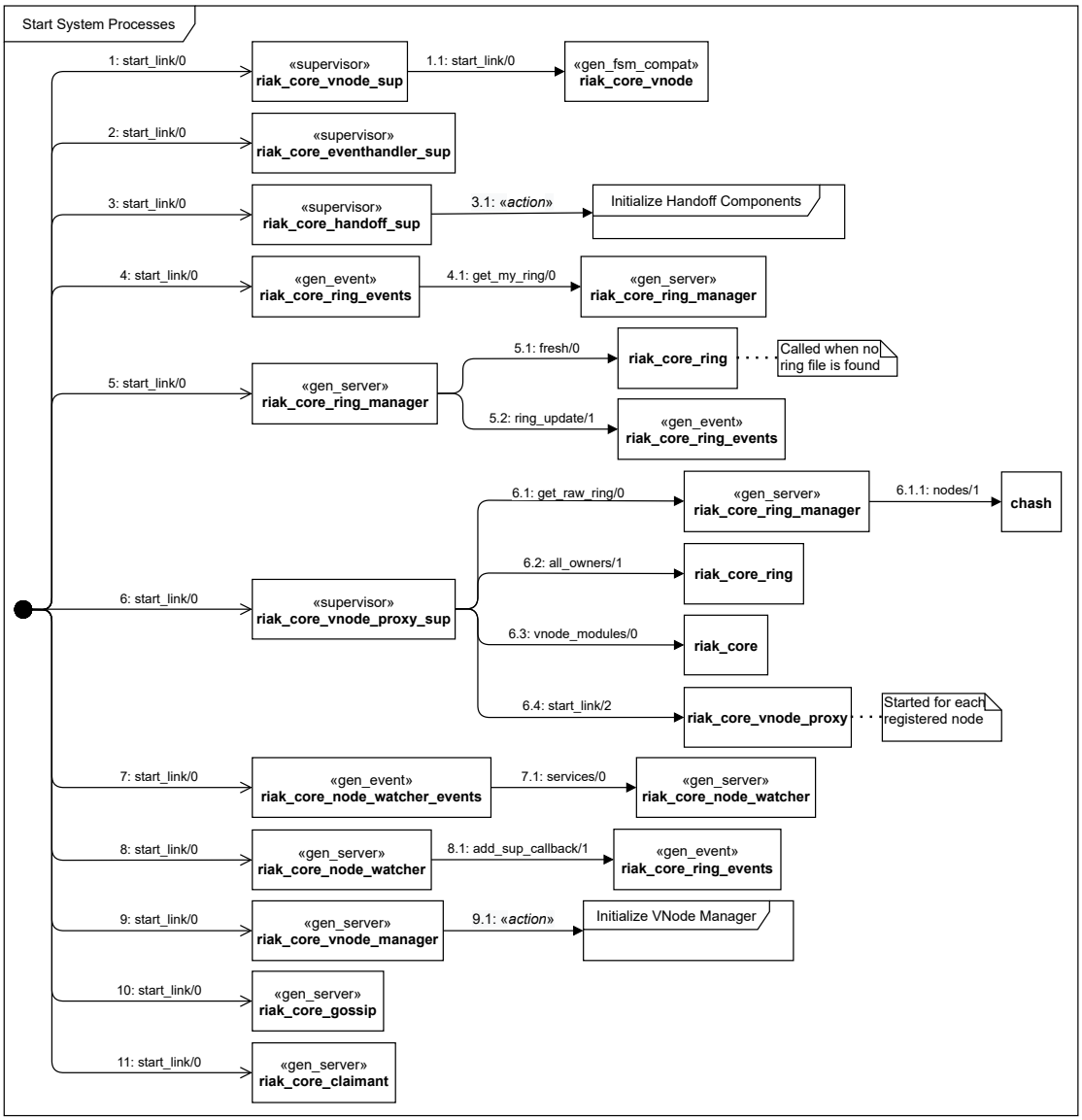


Figure A.17.: Start System Processes, see Action A.17

.

1 Start the supervisor for the local vnode.

1.1 Initialize the vnode with index, module, forwarding, and timeout settings.

2 Starts the eventhandler supervisor.

3 Starts the handoff supervisor and subsequently the supervised handoff processes.

3.1 <<*action*>> Start all supervised handoff processes. See Action A.18.

4 Start the eventhandler with the event handling callback function.

4.1 Retrieve the ring to check if the callback is applicable to it.

5 Starts the ETS, loads the ring file or creates a ring and sets the ring.

5.1 [If no ring file is found] Create a fresh ring if no ring file was found.

6 Start the vnode proxy supervisor. Subsequentially starts all proxies for the vnode modules for each index.

6.1 Retrieve the ring to compute indices.

6.1.1 Retrieve all nodes.

6.2 Retrieve every node owning at least one index.

6.3 Retrieve registered vnode modules.

6.4 [For each registered node] Start the vnode proxy for each node.

7 Start the node watcher events with an event callback function.

7.1 Retrieve the node watcher services to test if the callback function is applicable.

8 Starts the node watcher and adds an update callback to the ring events.

8.1 Adds a callback to the ring events that is executed on events.

9 Initializes the vnode manager and subsequently updates all vnode settings.

9.1 <<*action*>> Setup all existing vnodes to conform with management. See Action A.19.

10 Initializes the gossip process with the default number of tokens.

11 Initialize the claimant with a random seed.

Action A.17.: Start System Processes, see Figure A.17

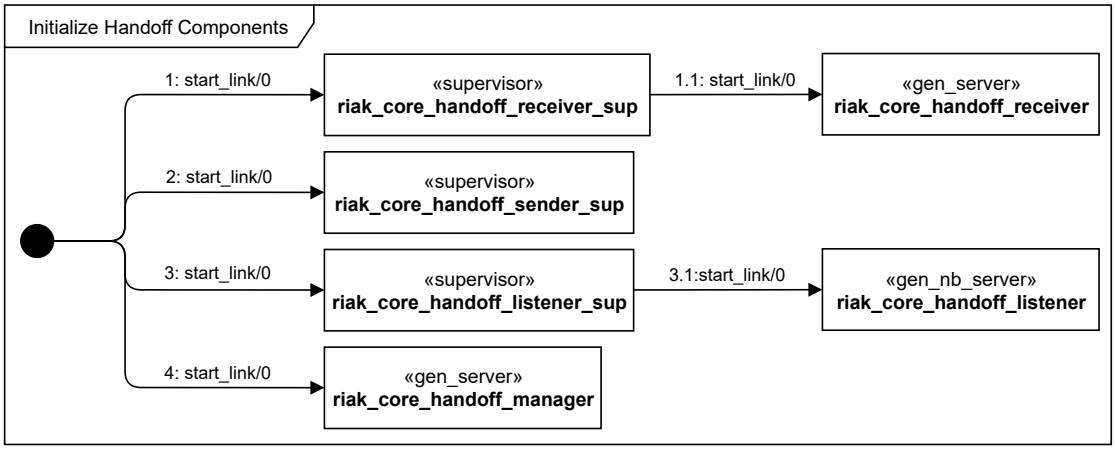


Figure A.18.: Initialize Handoff Components, see Action A.18

.

1 Starts the handoff receiver supervisor and subsequently the supervised child.

1.1 Starts the handoff receiver process with timeout settings.

2 Starts the handoff sender supervisor and subsequently the supervised child.

3 Starts the handoff listener supervisor and subsequently the supervised child.

3.1 Registers the process with the module name and sets the IP address and port number.

4 Initializes the handoff manager with an empty handoff list.

Action A.18.: Initialize Handoff Components, see Figure A.18

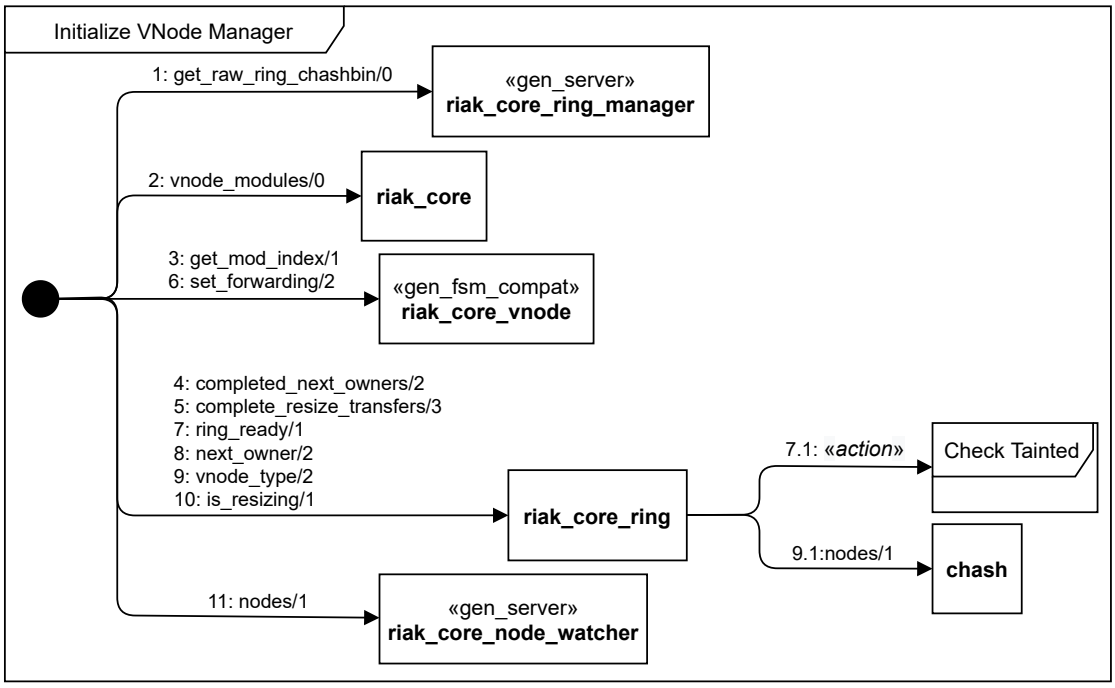


Figure A.19.: Initialize VNode Manager, see Action A.19

.

1 Retrieves the ring and chashbin structure of the current ring.

2 Retrieves registered vnode modules from the application settings.

3 Retrieves the index of the vnode.

4 Checks if the owner transfer is completed.

5 Checks if the resize transfer is completed.

6 Set vnode to forward requests.

7 Checks if the ring is ready for operations.

7.1 *<<action>>* Check if the ring is marked tainted. See Action A.3.

8 Retrieves ownership change for an index.

9 Determines if a vnode is fallback, future primary or resized primary.

9.1 Retrieve all indices and owners.

10 Check if the ring is currently resizing.

11 Retrieve all active nodes.

Action A.19.: Initialize VNode Manager, see Figure A.19

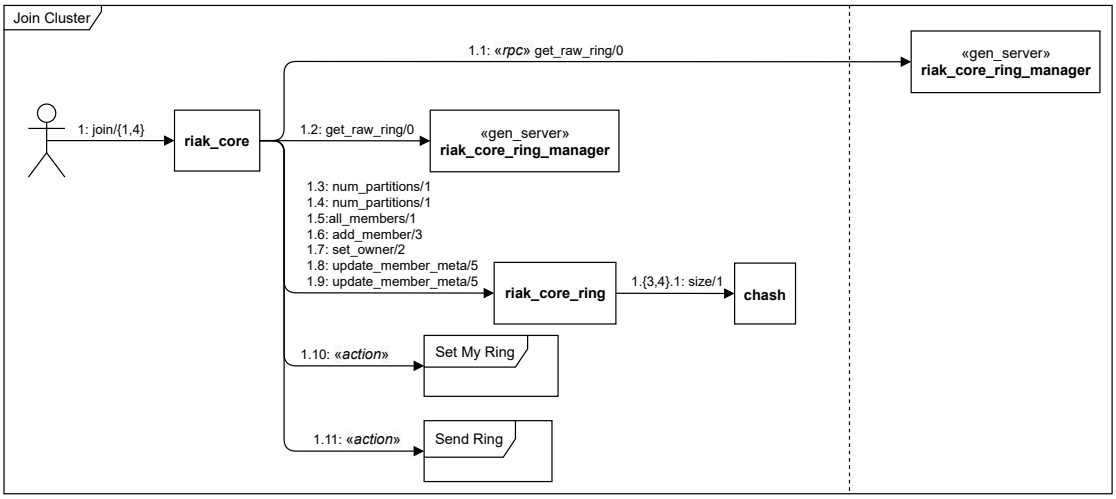


Figure A.20.: Join Cluster, see Action A.20

.

A.3. Join Cluster

1 is the entry point to join a cluster. One can either specify just a remote node hosting the cluster the local node should join or specify the joining node and the cluster node together with join modes. The joining node requests the remote node's ring and reconciles the ring locally. Subsequently the ring is sent to remote node where it is again checked and reconciled. After this has finished without errors, the updated ring is distributed via the gossip protocol to the other cluster members.

- 1.1 <<*rpc*>> Retrieves the ring structure of the cluster the node is about to join.
- 1.2 Retrieves the ring of the joining node.
- 1.3 Retrieves the number of partitions on the remote ring.
  - 1.3.1 Retrieves the number of entries in the chash structure.
- 1.4 Retrieves the number of partitions in the local ring.
  - 1.4.1 Retrieves the number of entries in the chash structure.
- 1.5 Checks if the local node is the only node in the ring.
- 1.6 Add the local node to the remote ring.
- 1.7 Set the local node as the ring owner.
- 1.8 Sets the gossip version.
- 1.9 Sets the auto join setting.
- 1.10 <<*action*>> Set the new ring as the local ring. See Action A.21
- 1.11 <<*action*>> Send the new ring to the cluster node. See Action A.1.

Action A.20.: Join Cluster, see Figure A.20

1 Set the local ring and notify the cluster.

- 1.1 <<*action*>> Checks if the ring is marked as tainted. See Action A.3.
- 1.2 <<*action*>> Set the ring in the ETS table. See Action A.7.
- 1.3 Notify listeners of the updated ring.

Action A.21.: Set My Ring, see Figure A.21

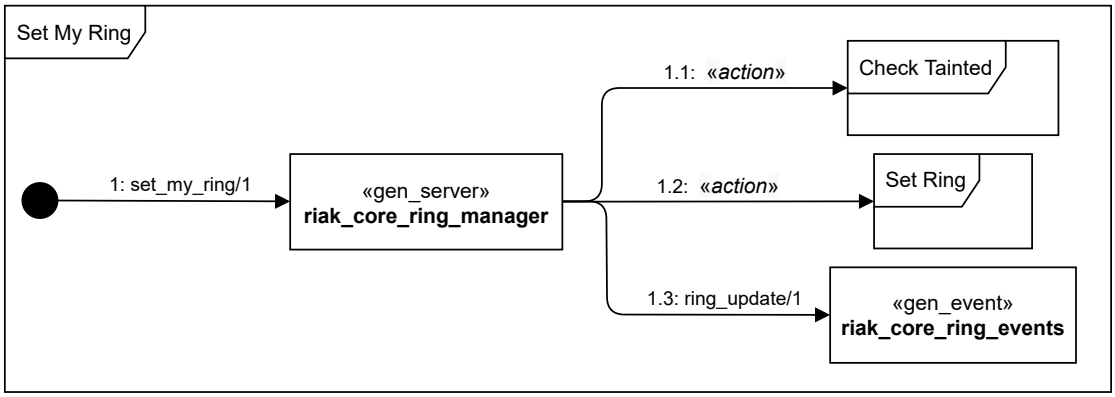


Figure A.21.: Set My Ring, see Action A.21

.

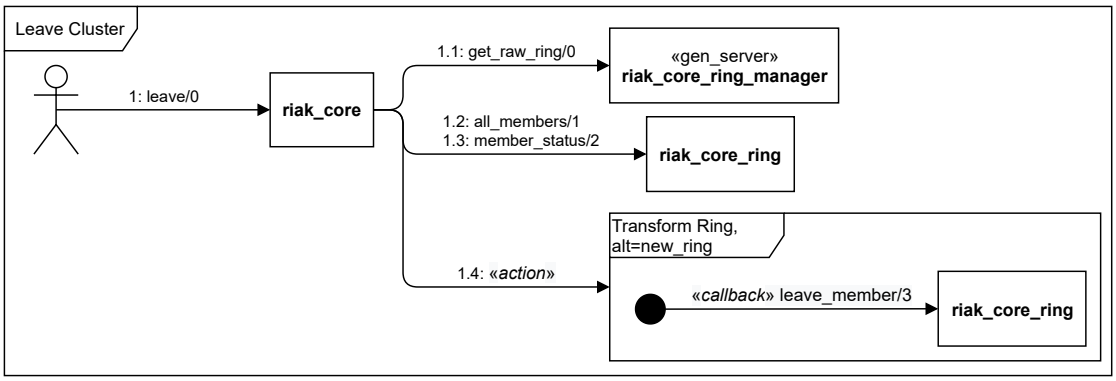


Figure A.22.: Leave Cluster, see Action A.22

.

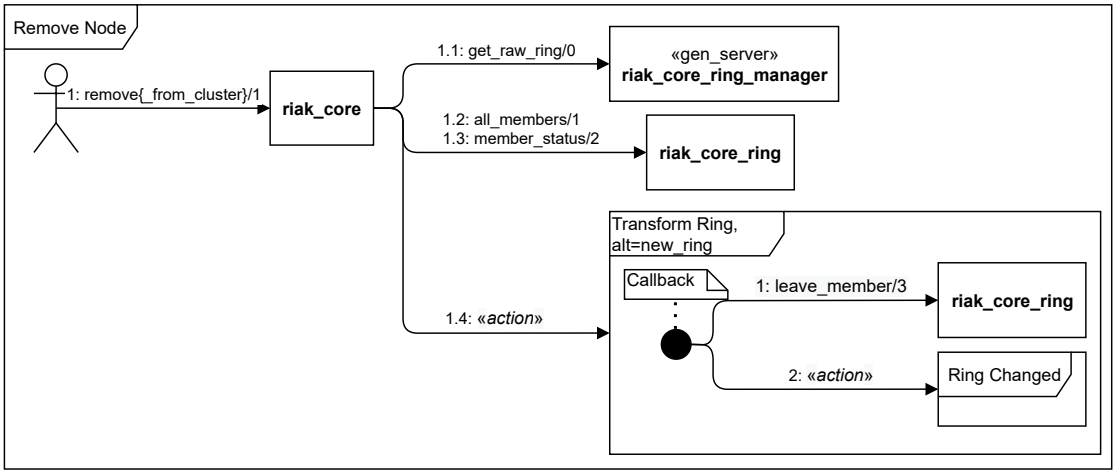


Figure A.23.: Remove Node, see Action A.23

.

A.4. Leave Cluster

1 Removes the local node from the ring if it is a valid node and not the only node.

1.1 Retrieve the ring to check the local nodes status.

1.2 Retrieve nodes on the ring to check if there are other nodes.

1.3 Check if the node is a valid member.

1.4 *<<action>>* Transform the ring with the given callback and gossip the transformed ring.

1.4.1 *<<callback>>* Set the state of the node to leaving.

Action A.22.: Leave Cluster, see Figure A.22

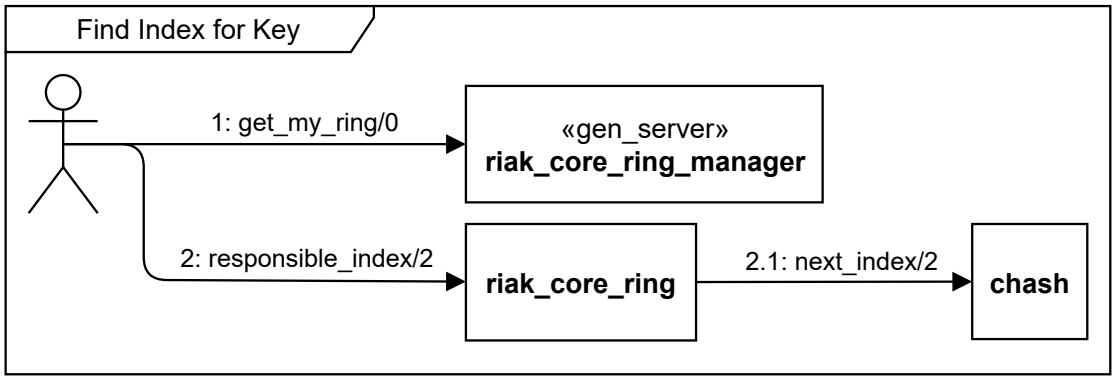


Figure A.24.: Find Index for Key, see Action A.24

.

A.5. Remove Node

1 Removes a node from the cluster. The difference to remove_from_cluster is a check if the node name is an atom.

- 1.1 Retrieve the ring to check the nodes status.
- 1.2 Retrieve nodes on the ring to check if there are other nodes.
- 1.3 Check if the node is a valid member.
- 1.4 <<*action*>> Transform the ring with the given callback and gossip the transformed ring. See Action A.2.
  - 1.4.1 <<*callback*>> Set the state of the node as invalid.
  - 1.4.2 <<*callback*>><<*action*>> Applies marked state changes to the ring and informs cluster members. See Action A.4.

Action A.23.: Remove Node, see Figure A.23

A.6. Navigate via riak_core_ring

1 Retrieve the current ring used as input to the library functions.

2 Retrieve the index of the node responsible for a key.

- 2.1 Retrieve the index of the next node.

Action A.24.: Find Index for Key, see Figure A.24

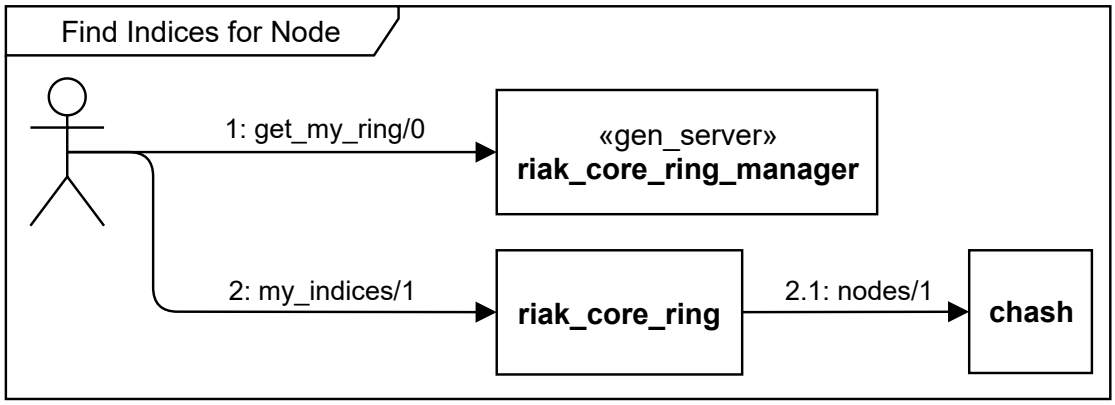


Figure A.25.: Find Indices for Node, see Action A.25

.

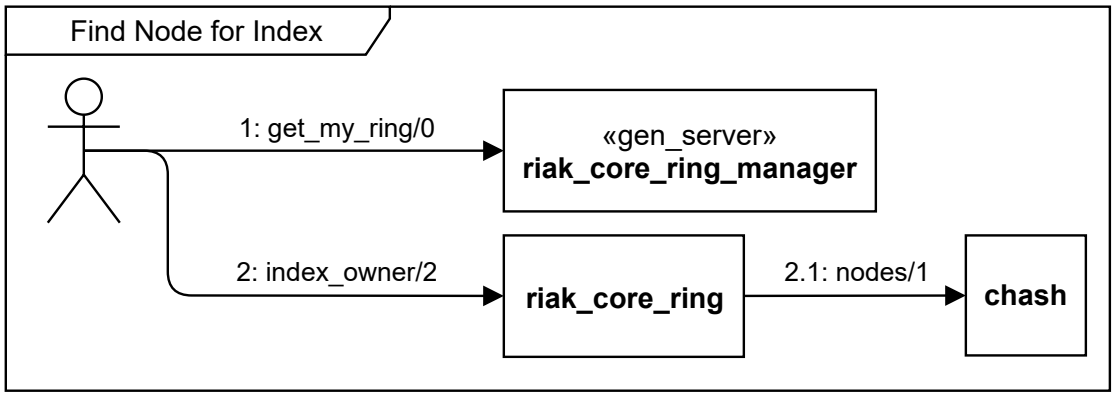


Figure A.26.: Find Node for Index, see Action A.26

.

1 Retrieve the current ring used as input to the library functions.

2 Retrieve all indices of the local node on the ring.

2.1 Retrieve all indices with the respective owners.

Action A.25.: Find Indices for Node, see Figure A.25

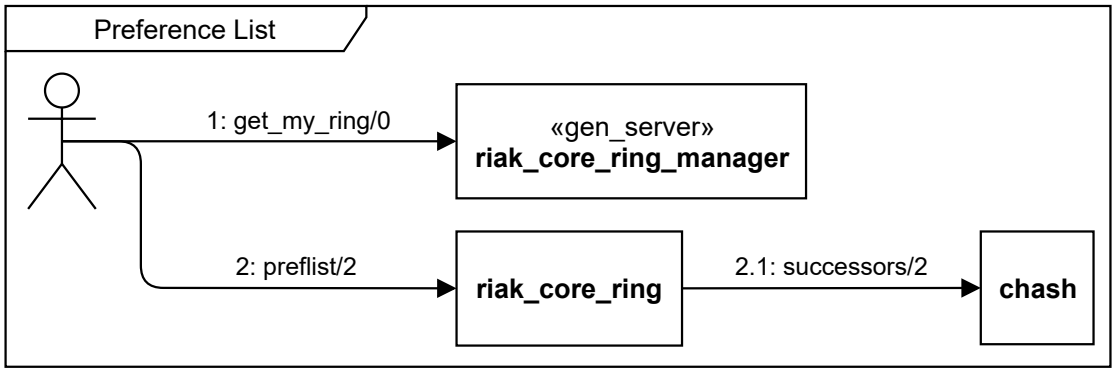


Figure A.27.: Preference List, see Action A.27

.

1 Retrieve the current ring used as input to the library functions.

2 Retrieve the owning node for a given index.

  2.1 Retrieve all indices and their owner node

Action A.26.: Find Node for Index, see Figure A.26

1 Retrieve the current ring used as input to the library functions.

2 Retrieve the preference list for a given key.

  2.1 Retrieve all indices and their owning nodes starting from a given index and wrapping around the ring.

Action A.27.: Preference List, see Figure A.27

A.7. Navigate via chashbin

1 Retrieve the local chash binary to use as input for the library functions.

2 Compute the index of the section responsible for a given key.

  2.1 Retrieve the offset from one section to the next.

Action A.28.: Find Index for Key, see Figure A.28

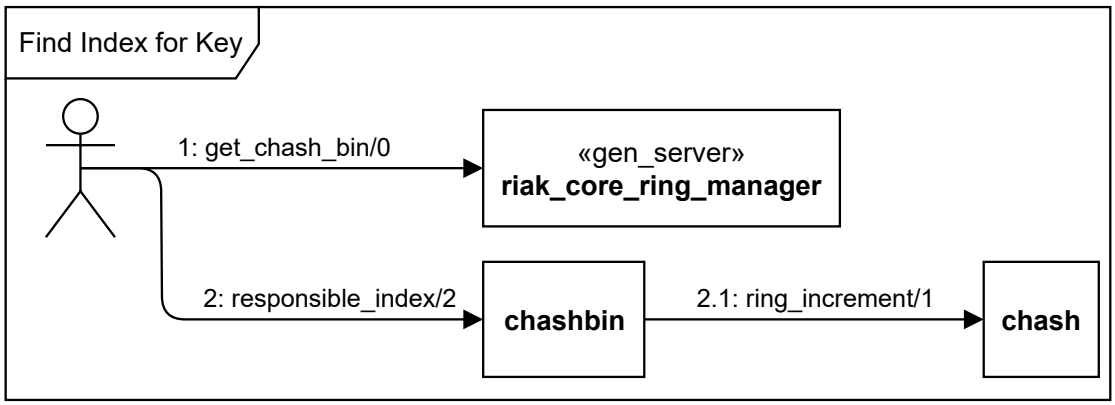


Figure A.28.: Find Index for Key, see Action A.28

.

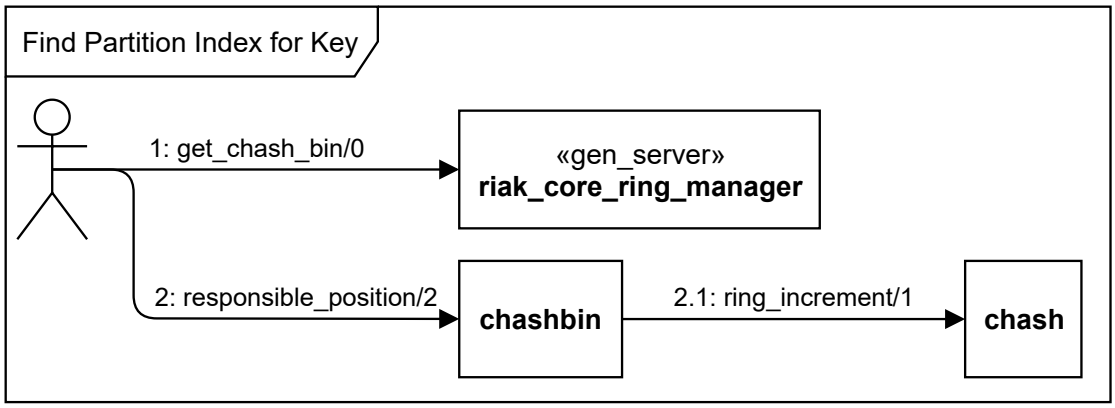


Figure A.29.: Find Partition Index for Key, see Action A.29

.

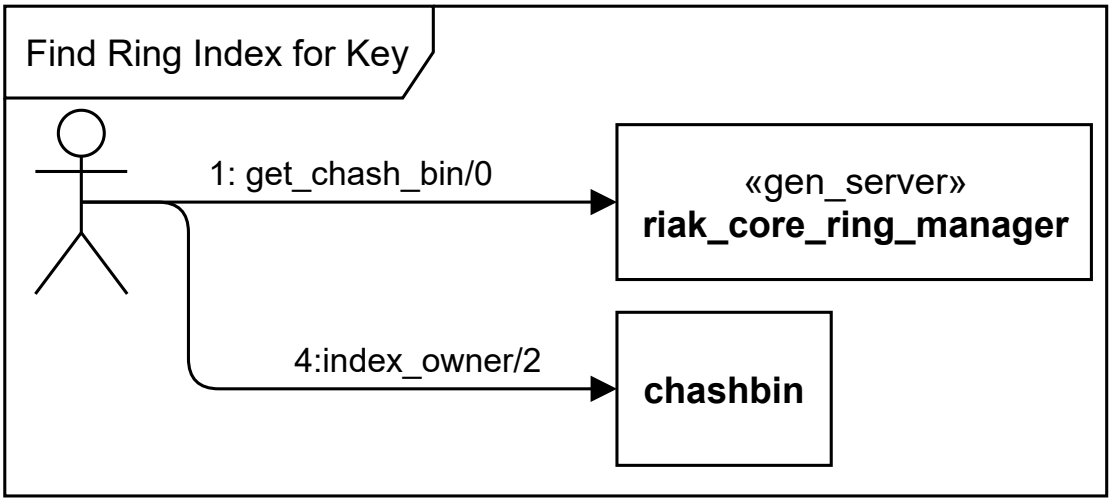


Figure A.30.: Find Ring Index for Key, see Action A.30

.

1 Retrieve the local chash binary to use as input for the library functions.

2 Compute the position of the section responsible for a given key. The position of a section describes the index of the position on the ring starting from as opposed to the index resulting from the hash function.

  2.1 Retrieve the offset from one section to the next.

Action A.29.: Find Partition Index for Key, see Figure A.29

1 Retrieve the local chash binary to use as input for the library functions.

2 Retrieve the node owning the given index.

Action A.30.: Find Ring Index for Key, see Figure A.30

1 Retrieve the local chash binary to use as input for the library functions.

2 Retrieve an iterator starting at the given position.

  2.1 Retrieve the offset from one section to the next.

3 Iterate over all nodes and receive the list in order of iteration to build the preference list.

Action A.31.: Preference List, see Figure A.31

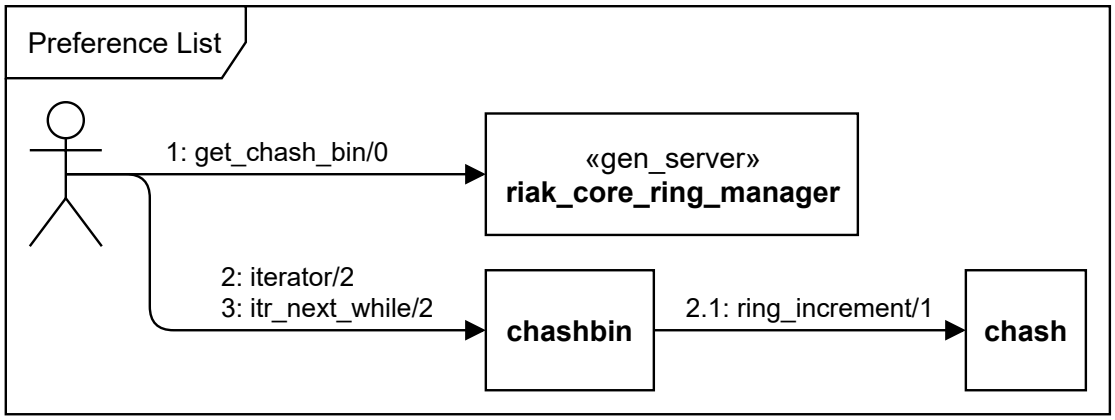


Figure A.31.: Preference List, see Action A.31

.

A.8. Navigate via riak_core_apl

1 retrieves the active preference list containing nodes that are up. This function does not need a structure as an argument and uses the local binary chash internally.

1.1 Retrieve the binary chash structure.

1.2 Retrieve all nodes that are up at the moment.

1.3 *<<action>>* Compute the active preference list from up nodes and binary chash. See Action A.39.

Action A.32.: Get APL Locally, see Figure A.32

1 Compute the active preference list from a ring and up nodes.

1.1 *<<action>>* Compute the active preference list from a ring containing annotations of primary and fallback nodes. See Action A.34.

Action A.33.: Get APL from Ring, see Figure A.33

1 Compute the active preference list from a ring and up nodes where nodes are annotated with primary or fallback.

1.1 Retrieve the preference list containing all nodes.

1.1.1 Retrieve all node entries beginning at the given key wrapping around the ring.

Action A.34.: Get Annotated APL from Ring, see Figure A.34

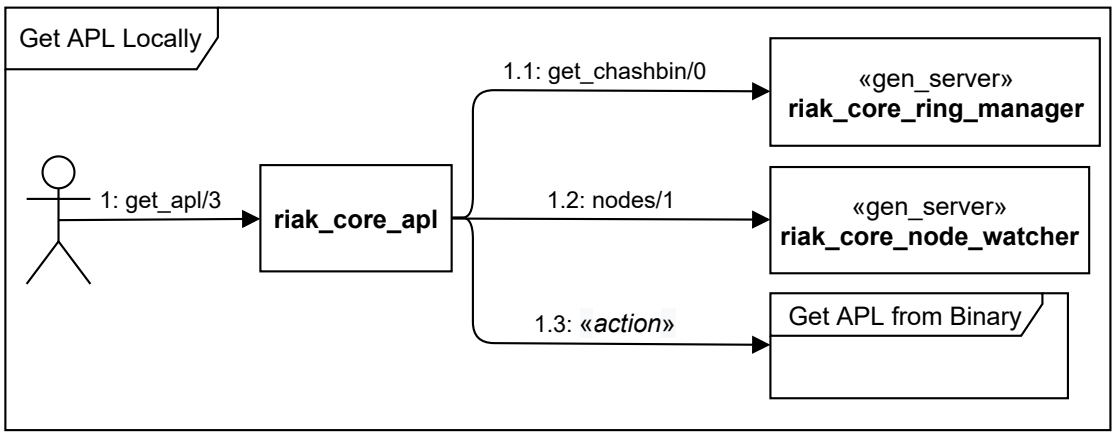


Figure A.32.: Get APL Locally, see Action A.32

.

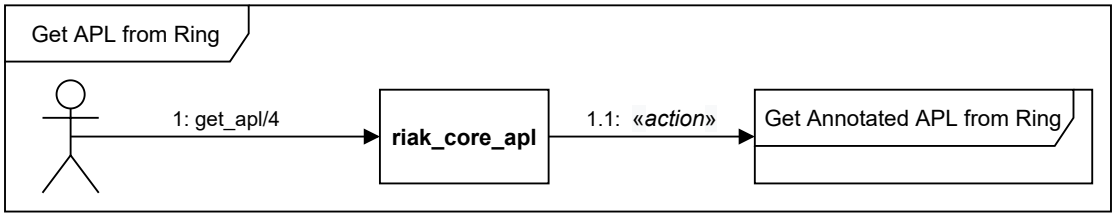


Figure A.33.: Get APL from Ring, see Action A.33

.

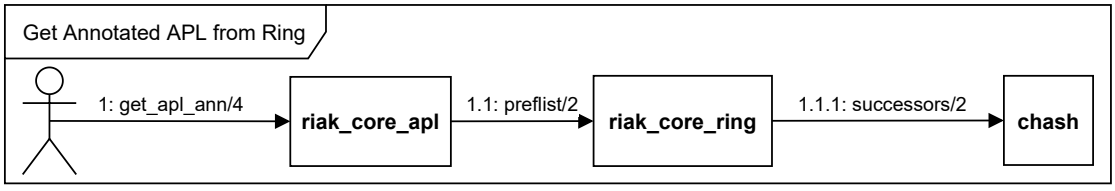


Figure A.34.: Get Annotated APL from Ring, see Action A.34

.

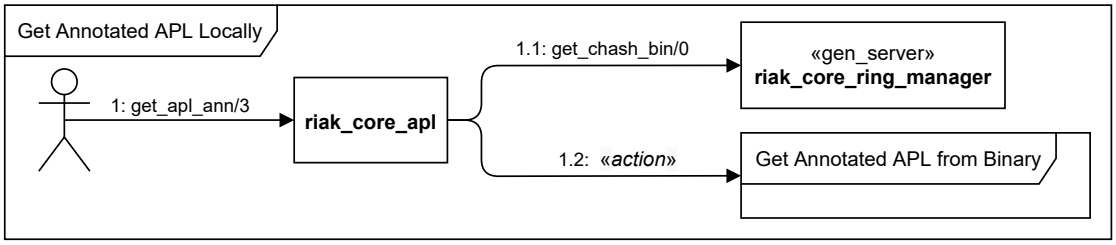


Figure A.35.: Get Annotated APL Locally, see Action A.35

.

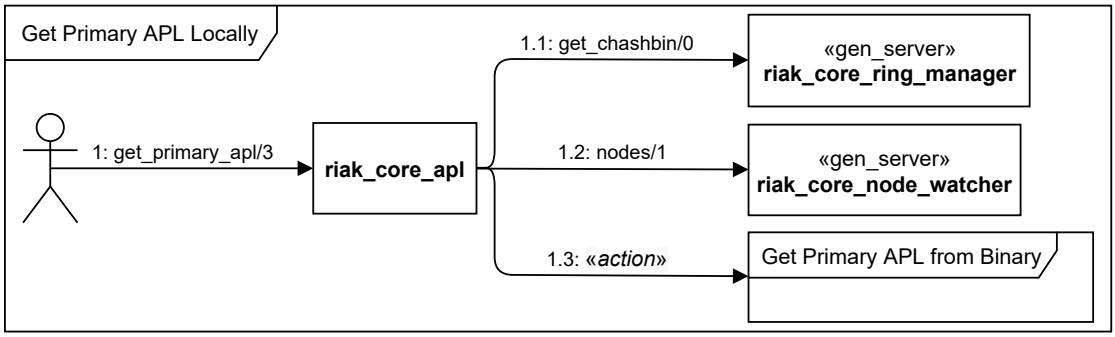


Figure A.36.: Get Primary APL Locally, see Action A.36

.

1 Compute the active preference list without needing a structure as the input. Internally the local binary chash is used.

1.1 Retrieve the local binary chash structure.

1.2 *<<action>>* Compute the active preference list which contains primary and fallback annotations.

Action A.35.: Get Annotated APL Locally, see Figure A.35

1 Compute the active preference list only containing primary nodes.

1.1 Receive the binary chash structure.

1.2 Receive nodes that are currently up.

1.3 *<<action>>* Compute the active preference list from a binary chash. See Action A.38.

Action A.36.: Get Primary APL Locally, see Figure A.36

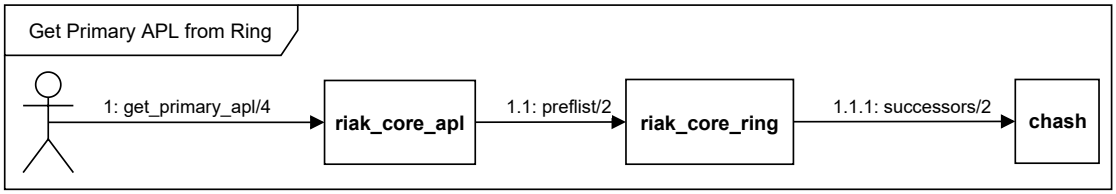


Figure A.37.: Get Primary APL from Ring, see Action A.37

.

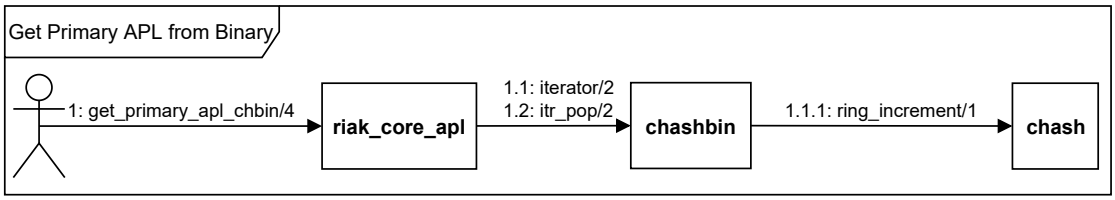


Figure A.38.: Get Primary APL from Binary, see Action A.38

.

1 Compute the active preference list only containing primary nodes from the ring and up nodes.

1.1 Retrieve the preference list containing all nodes.

1.1.1 Retrieve all node entries beginning at the given key wrapping around the ring.

Action A.37.: Get Primary APL from Ring, see Figure A.37

1 Compute the active preference list only containing primary nodes from the binary chash and upnodes.

1.1 Create an iterator starting at a given position.

1.1.1 Receive the offset from one section to another.

1.2 Iterate over all nodes and receive the list in order of iteration.

Action A.38.: Get Primary APL from Binary, see Figure A.38

1 Compute the active preference list from a binary chash and up nodes.

1.1 *<<action>>* Compute the active preference list which contains nodes annotated with primary or fallback. See Action A.40.

Action A.39.: Get APL from Binary, see Figure A.39

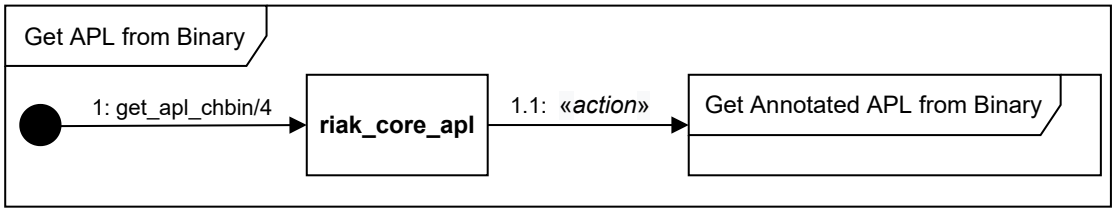


Figure A.39.: Get APL from Binary, see Action A.39

.

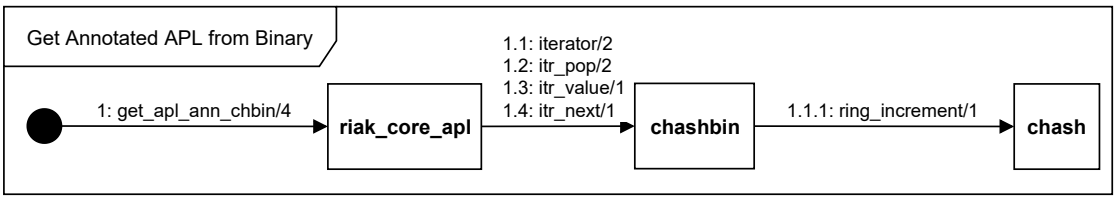


Figure A.40.: Get Annotated APL from Binary, see Action A.40

.

1 Compute the preference list from a binary chash with nodes marked as primary or fallback.

1.1 Create an iterator starting at a given position.

1.1.1 Receive the offset from one section to another.

1.2 Iterate over all nodes and receive the list in order of iteration.

1.3 Receive the value at the current iterator position.

1.4 Advance the iterator by one position.

Action A.40.: Get Annotated APL from Binary, see Figure A.40

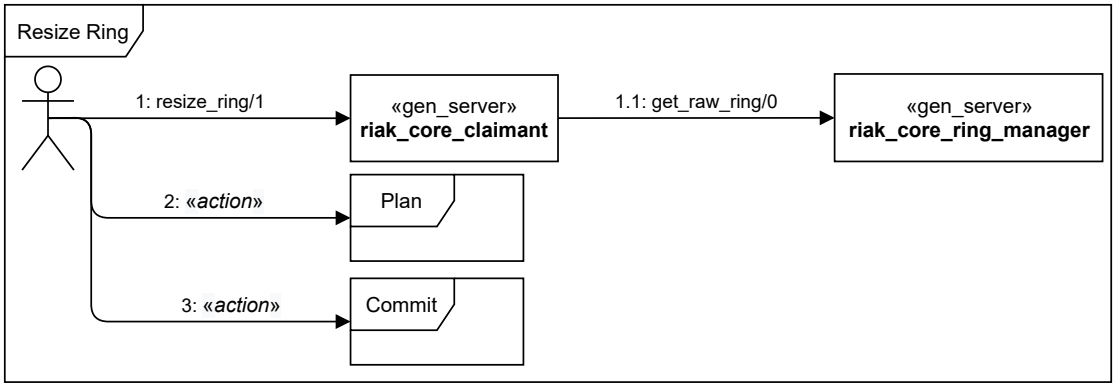


Figure A.41.: Resize Ring, see Action A.41

.

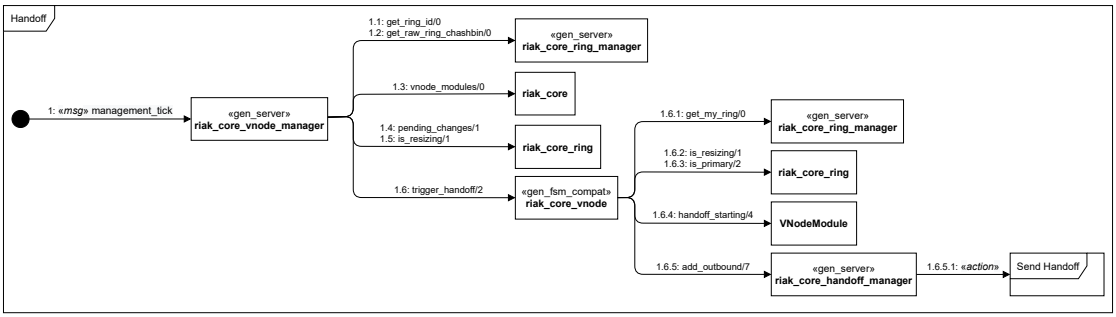


Figure A.42.: Handoff, see Action A.42

.

A.9. Resize Ring

1 Stage a resize for the current ring.

1.1 Retrieve the current ring.

2 <<*action*>> Generate the provisional next ring after the resize operation. See Action A.10.

3 <<*action*>> Commit all staged changes and install the resized ring. See Action A.16.

Action A.41.: Resize Ring, see Figure A.41

Figure A.43.: Send Handoff, see Action A.43

.

A.10. Handoff

1 <<*msg*>> When a management tick message is received, the vnode manager executes its periodical tasks. In the following only tasks relevant to the handoff are considered.

- 1.1 Retrieve the identifier of the currently installed ring.
- 1.2 Retrieve the binary chash structure of the currently installed ring.
- 1.3 Retrieve all modules registered as vnodes.
- 1.4 Retrieve all owner changes scheduled on the ring.
- 1.5 Check if the ring is undergoing a resizing operation.
- 1.6 Start the handoff process for vnodes having outgoing handoffs scheduled.
  - 1.6.1 Retrieve the locally installed ring.
  - 1.6.2 Check if the ring is resizing.
  - 1.6.3 Check if the handed off index is a primary or redundancy storage.
  - 1.6.4 Check in the implementation of the vnode behavior if the handoff can start.
  - 1.6.5 Add an outbound handoff to the list of ongoing handoffs.
    - 1.6.5.1 <<*action*>> Send data owned by the vnode on the handed off index to the new owner. See Action A.43.

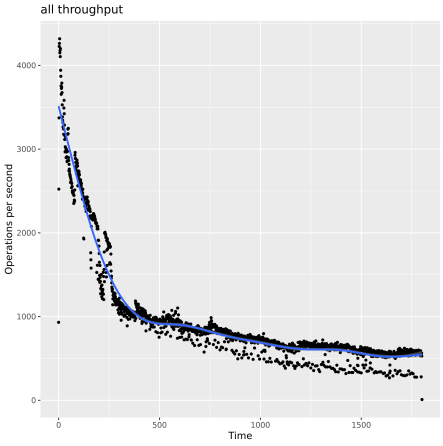
Action A.42.: Handoff, see Figure A.42

1 Prepare and send data from one node to another.

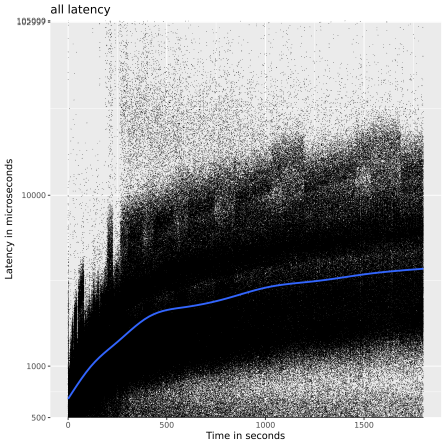
- 1.1 Start a process to send the handoff data.
  - 1.1.1 Starts the handoff sender. On start, all ports and timeouts are set and tested.
    - 1.1.1.1 Sets up the synchronized call to a vnode with the fold request.

Action A.43.: Send Handoff, see Figure A.43

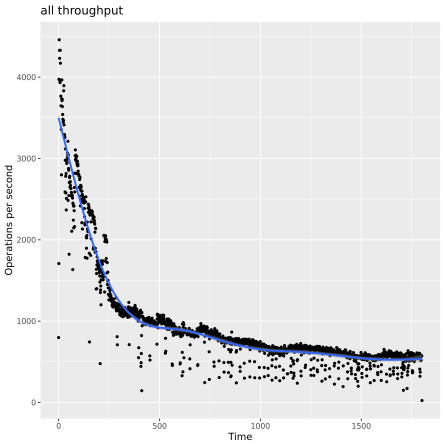
B. Evaluation Data



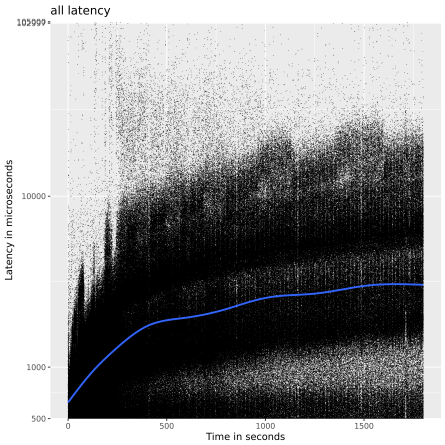
(a) Throughput for ConsistentHashing_C0_read_heavy



(b) Latency for ConsistentHashing_C0_read_heavy



(c) Throughput for ConsistentHashing_C0_write_heavy



(d) Latency for ConsistentHashing_C0_write_heavy

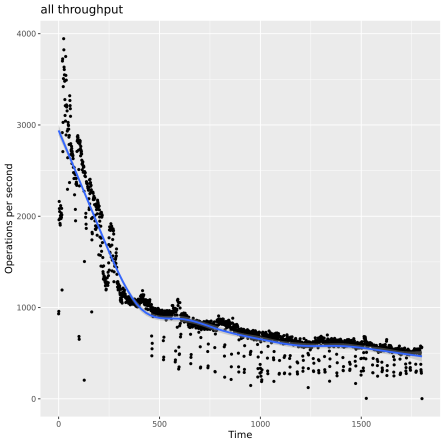
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	103093	859967	951426	0.343715513591187	0.343802934190795	0.343849103736302
n_1	98379	820221	907435	0.327998879764751	0.327913032110427	0.327950583071044
n_2	98465	821149	908126	0.328285606644062	0.328284033698778	0.328200313192653

(e) Stats for ConsistentHashing_C0_read_heavy

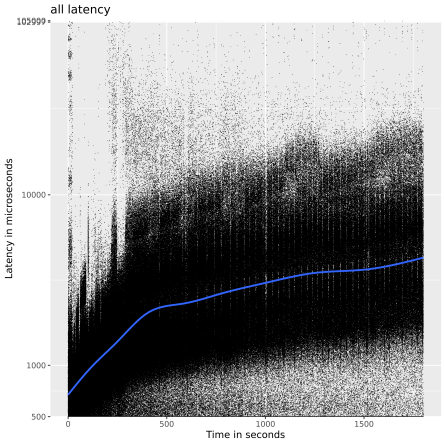
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	103103	890468	891069	0.343711037770444	0.343677212011883	0.343680358047862
n_1	98390	849929	850468	0.327999466613328	0.328031135456914	0.328020778130817
n_2	98477	850604	851189	0.328289495616228	0.328291652531203	0.328298863821322

(f) Stats for ConsistentHashing_C0_write_heavy

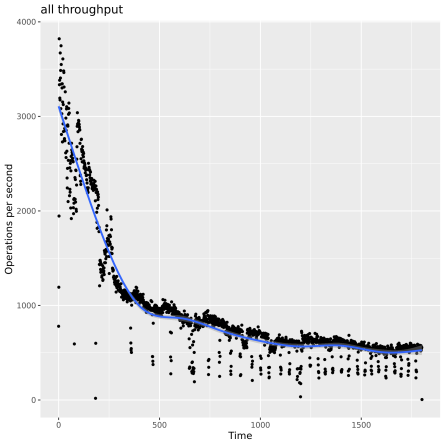
Figure B.1.: Evaluation Data for ConsistentHashing_C0



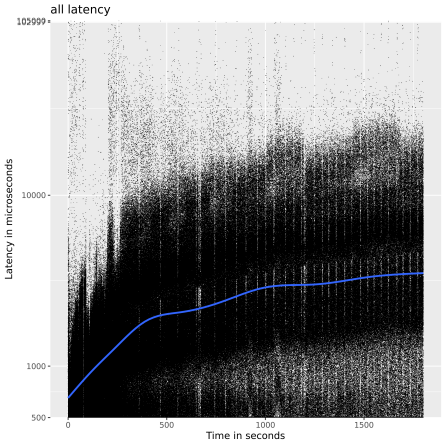
(a) Throughput for ConsistentHashing_C1_read_heavy



(b) Latency for ConsistentHashing_C1_read_heavy



(c) Throughput for ConsistentHashing_C1_write_heavy



(d) Latency for ConsistentHashing_C1_write_heavy

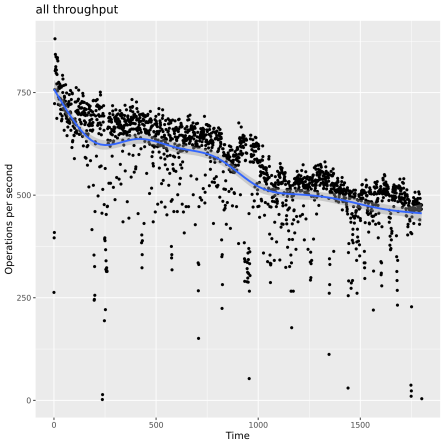
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	74924	603572	668253	0.249743261446538	0.249724251277549	0.25003732101347
n_1	75039	604267	667574	0.249816615318956	0.249807441511264	0.250023851775526
n_2	75051	604939	668641	0.250200056015684	0.250095089334308	0.249769806982079
n_3	74902	603371	668289	0.249743261446538	0.249724251277549	0.25003732101347

(e) Stats for ConsistentHashing_C1_read_heavy

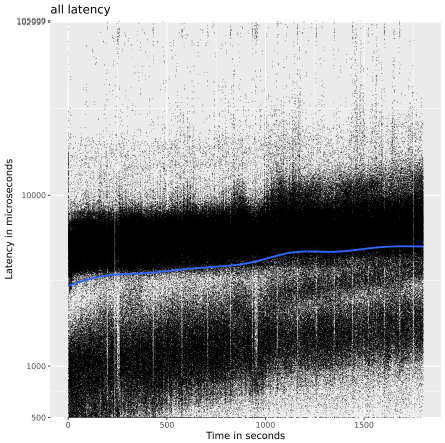
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	74935	628817	629194	0.249815809388554	0.249844050352069	0.249771444246092
n_1	75047	628879	629263	0.250189191261531	0.249868684436583	0.249798835209217
n_2	75066	630243	630756	0.250252532829268	0.25041063429589	0.250391512135983
n_3	74913	628899	629866	0.249742466520648	0.249876630915458	0.250038208408708

(f) Stats for ConsistentHashing_C1_write_heavy

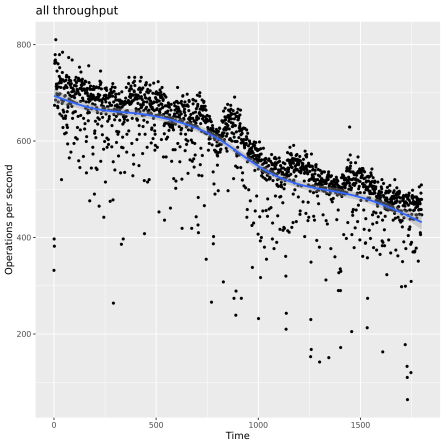
Figure B.2.: Evaluation Data for ConsistentHashing_C1



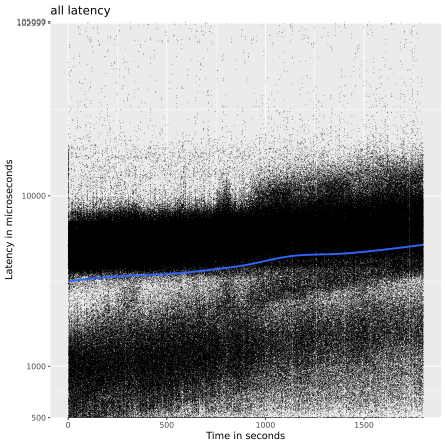
(a) Throughput for ConsistentHashing_C2_read_heavy



(b) Latency for ConsistentHashing_C2_read_heavy



(c) Throughput for ConsistentHashing_C2_write_heavy



(d) Latency for ConsistentHashing_C2_write_heavy

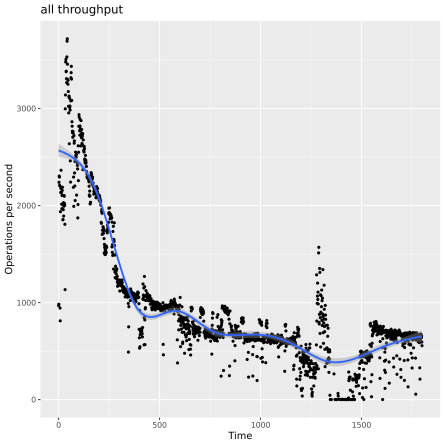
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	41806	201465	224062	0.140502644969181	0.140264508933234	0.14077579929418
n_1	41733	201505	223726	0.140257304752879	0.140292357841765	0.140564694026161
n_2	41947	202294	223361	0.140976521277382	0.140841677562552	0.140335368362985
n_3	46540	224657	249346	0.156412789955167	0.156411306099886	0.1566614707126
n_4	41848	202290	223914	0.140643799614177	0.140838892671699	0.140682812449933
n_5	41771	201781	223160	0.140385016098351	0.140484515310634	0.140209082175867
n_6	41901	202330	224054	0.140821923332863	0.140866741580231	0.140770772978274

(e) Stats for ConsistentHashing_C2_read_heavy

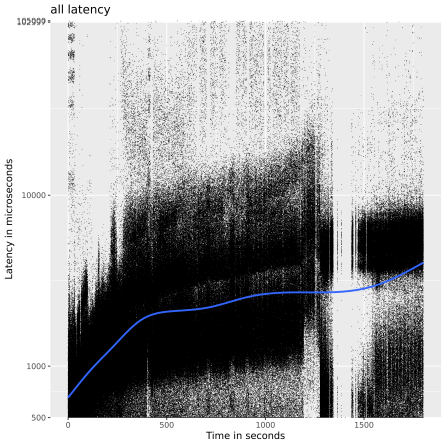
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	41942	217090	217913	0.140608133024909	0.14045514480292	0.140657818468645
n_1	41849	216852	217435	0.14074558315733	0.140926153810321	0.140676537300176
n_2	42026	217789	218256	0.14088973817426	0.140907391088872	0.140879217062279
n_3	46662	241915	243014	0.156431660464648	0.156516681353349	0.156859935374848
n_4	41983	217818	217942	0.14074558315733	0.140926153810321	0.140676537300176
n_5	41844	216698	216531	0.140279593683999	0.140201524568166	0.139765769324612
n_6	41984	217456	218151	0.140748935599584	0.14069194328741	0.140811441982595

(f) Stats for ConsistentHashing_C2_write_heavy

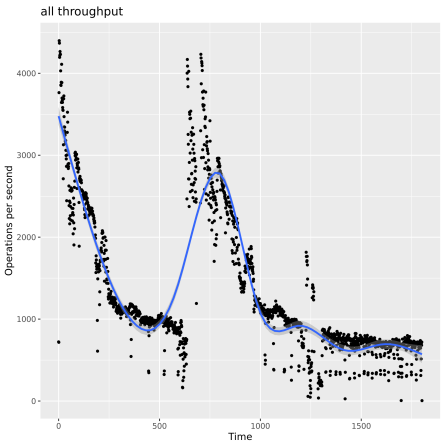
Figure B.3.: Evaluation Data for ConsistentHashing_C2



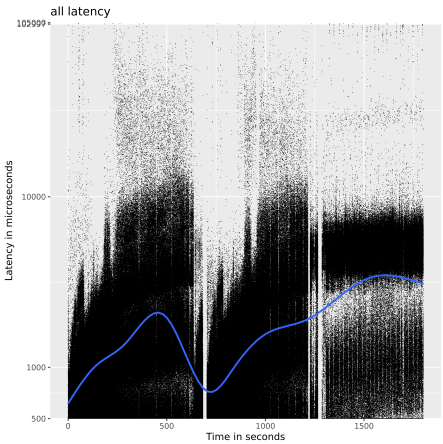
(a) Throughput for ConsistentHashing_dynamic_read_heavy



(b) Latency for ConsistentHashing_dynamic_read_heavy



(c) Throughput for ConsistentHashing_dynamic_write_heavy



(d) Latency for ConsistentHashing_dynamic_write_heavy

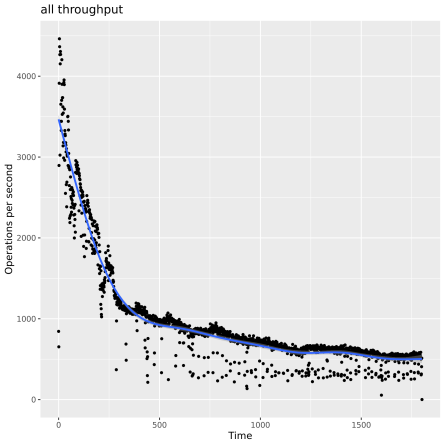
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	42106	690488	765645	0.142841150029684	0.299403524047225	0.29955421628296
n_1	41846	664416	735951	0.141959121363752	0.288098405523863	0.287936609038995
n_2	37573	669967	741840	0.12746331948096	0.290505382852921	0.290240646523325
n_3	46868	149656	166229	0.158995844288016	0.064892559747326	0.065036143145322
n_4	42163	44453	49749	0.143034517852599	0.019275331149088	0.019464011004919
n_5	42039	43642	48168	0.142613858027309	0.018923672238285	0.018845453819874
n_6	42180	43590	48366	0.14309218895768	0.018901124441292	0.018922920184605

(e) Stats for ConsistentHashing_dynamic_read_heavy

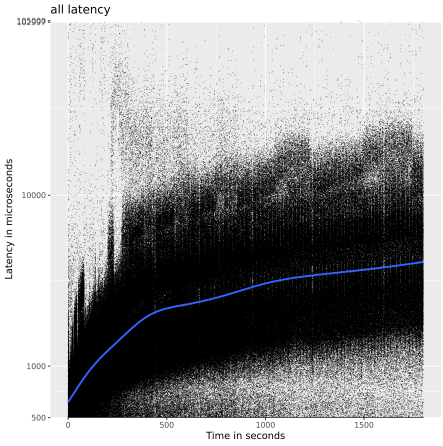
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	41892	973293	976226	0.140622010372434	0.274271806357583	0.274316180739257
n_1	41799	950963	953079	0.140309830315033	0.267979261937799	0.267811952583511
n_2	41971	946705	949072	0.140887195582484	0.266779366992012	0.266685999232317
n_3	46595	453983	456508	0.156408922307447	0.127931401402902	0.128277193023866
n_4	41928	75001	75194	0.140742854265622	0.0211351152722	0.021129257871136
n_5	41796	74463	74214	0.140299759990601	0.020983508066743	0.020853881209252
n_6	41924	74236	74469	0.140729427166379	0.020919539970761	0.020925535340661

(f) Stats for ConsistentHashing_dynamic_write_heavy

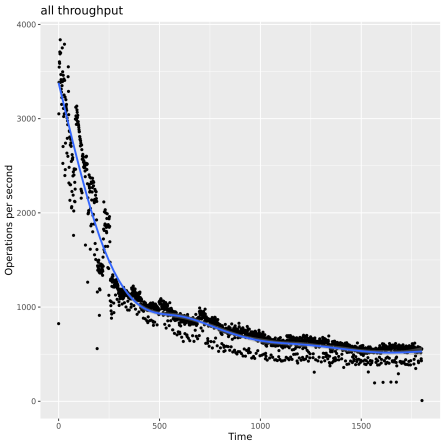
Figure B.4.: Evaluation Data for ConsistentHashing_dynamic



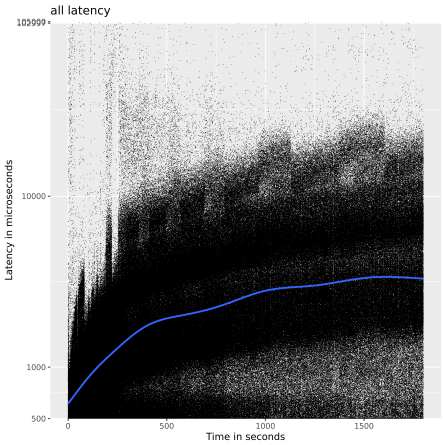
(a) Throughput for RandomSlicing_Jumping_C0_read_heavy



(b) Latency for RandomSlicing_Jumping_C0_read_heavy



(c) Throughput for RandomSlicing_Jumping_C0_write_heavy



(d) Latency for RandomSlicing_Jumping_C0_write_heavy

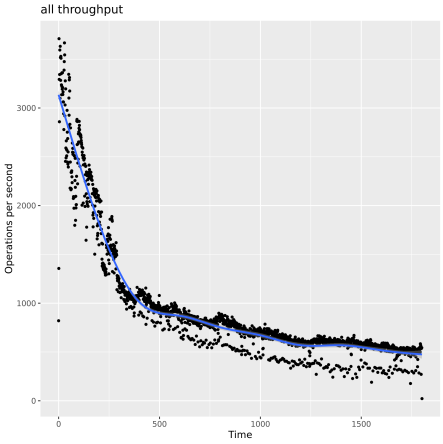
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	99976	819294	906303	0.333333333333333	0.333333333333333	0.333333333333333
n_1	99976	819294	906303	0.333333333333333	0.333333333333333	0.333333333333333
n_2	99976	819294	906303	0.333333333333333	0.333333333333333	0.333333333333333

(e) Stats for RandomSlicing_Jumping_C0_read_heavy

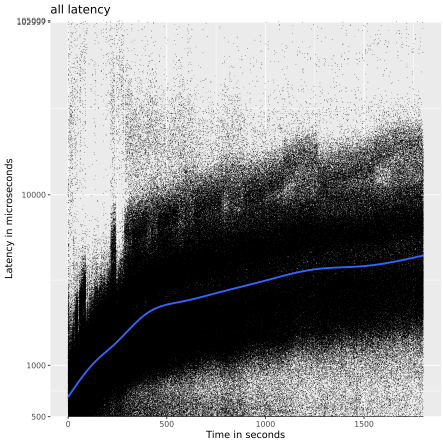
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	99990	858732	859417	0.333333333333333	0.333333333333333	0.333333333333333
n_1	99990	858732	859417	0.333333333333333	0.333333333333333	0.333333333333333
n_2	99990	858732	859417	0.333333333333333	0.333333333333333	0.333333333333333

(f) Stats for RandomSlicing_Jumping_C0_write_heavy

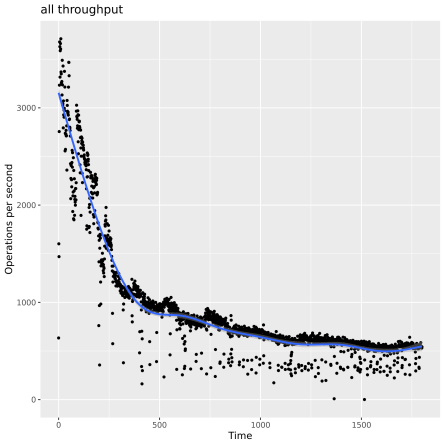
Figure B.5.: Evaluation Data for RandomSlicing_Jumping_C0



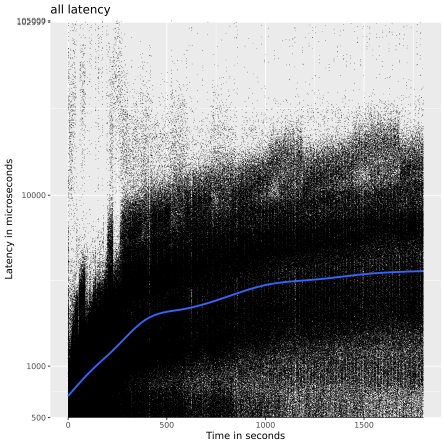
(a) Throughput for RandomSlicing_Jumping_C1_read_heavy



(b) Latency for RandomSlicing_Jumping_C1_read_heavy



(c) Throughput for RandomSlicing_Jumping_C1_write_heavy



(d) Latency for RandomSlicing_Jumping_C1_write_heavy

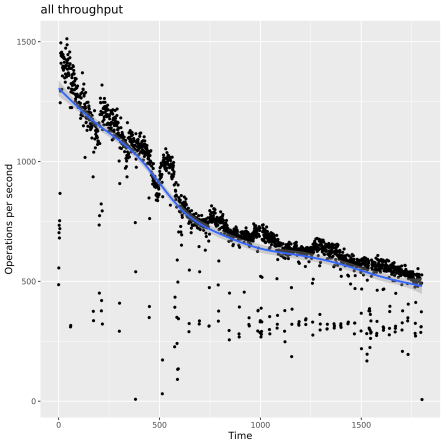
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	75090	604862	668618	0.250377617061289	0.250294110953176	0.250124198322275
n_1	75155	605479	670129	0.25059435091545	0.250549427812986	0.25068945032516
n_2	74879	603436	667799	0.249674065627011	0.249704026930342	0.249817817521241
n_3	74783	602828	666598	0.249353966396249	0.249452434303496	0.249368533831324

(e) Stats for RandomSlicing_Jumping_C1_read_heavy

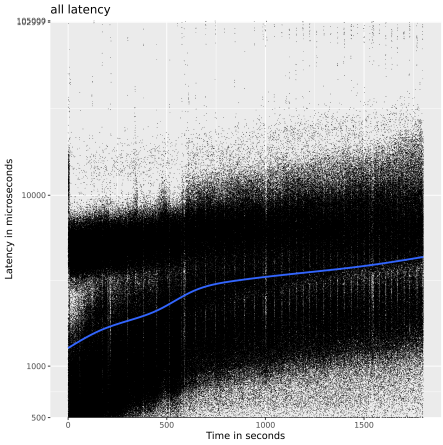
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	75106	629064	629715	0.250388387707612	0.250394061846216	0.25041665523375
n_1	75168	629444	629960	0.250595083311664	0.250545317908399	0.250514083563284
n_2	74891	627105	627841	0.249671620693564	0.249614297041431	0.249671427929481
n_3	74793	626683	627153	0.24934490828716	0.249446323203954	0.249397833273484

(f) Stats for RandomSlicing_Jumping_C1_write_heavy

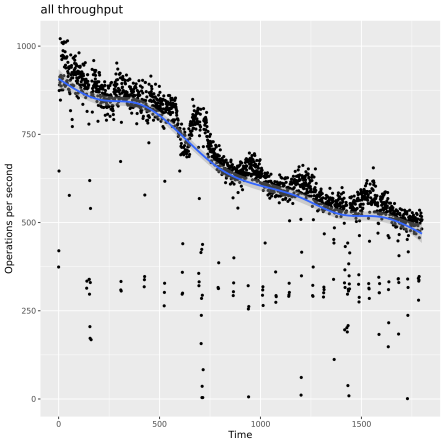
Figure B.6.: Evaluation Data for RandomSlicing_Jumping_C1



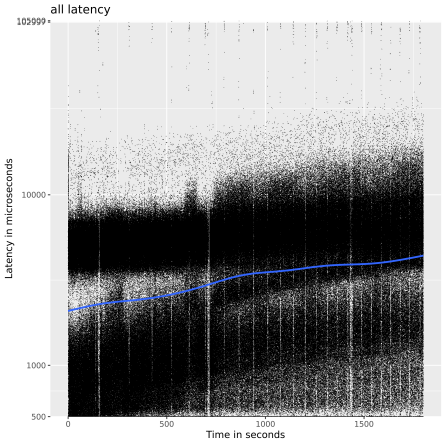
(a) Throughput for RandomSlicing_Jumping_C2_read_heavy



(b) Latency for RandomSlicing_Jumping_C2_read_heavy



(c) Throughput for RandomSlicing_Jumping_C2_write_heavy



(d) Latency for RandomSlicing_Jumping_C2_write_heavy

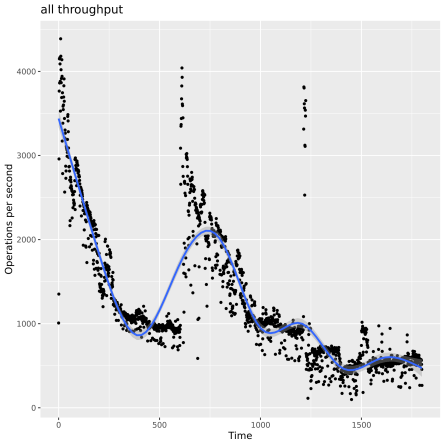
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	57238	374972	415598	0.191047426410459	0.190970641291649	0.191168664379623
n_1	50167	328547	363113	0.167446036561961	0.167326710486242	0.167026374594869
n_2	59500	390119	431981	0.198597467965728	0.198684903432941	0.19870459147391
n_3	24699	161975	179615	0.082439644727488	0.082492745120209	0.082620127268529
n_4	20511	134370	149037	0.068461053200757	0.06843370990463	0.06855471930362
n_5	53425	349990	387724	0.178320499597798	0.178247481800412	0.178347054672845
n_6	34061	223533	246918	0.113687871535809	0.113843807963918	0.113578468306604

(e) Stats for RandomSlicing_Jumping_C2_read_heavy

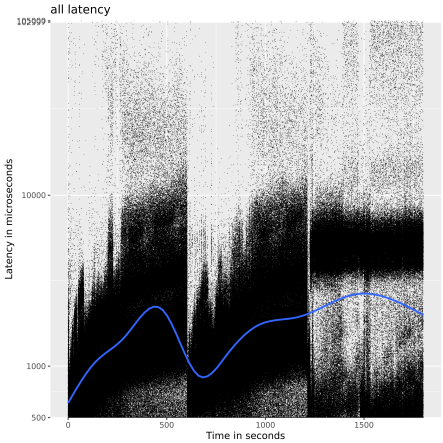
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	42613	255496	255398	0.142393621642574	0.142552346264233	0.142323287036108
n_1	48927	293229	293960	0.163492190789342	0.163605230386052	0.163812376984684
n_2	54471	326411	326704	0.182017763698699	0.18211891339377	0.182059323752906
n_3	26069	155974	156194	0.08711095962735	0.087024687886376	0.087040789259579
n_4	46353	277739	278407	0.154891031938569	0.154962684735111	0.155145300174088
n_5	40470	241972	243054	0.13523267237404	0.13500671764039	0.135444460047746
n_6	40359	241475	240775	0.134861759929426	0.134729419694068	0.134174462744888

(f) Stats for RandomSlicing_Jumping_C2_write_heavy

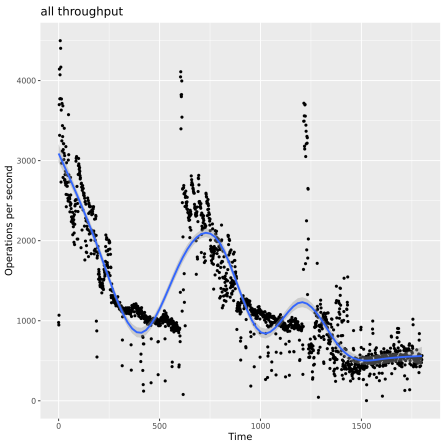
Figure B.7.: Evaluation Data for RandomSlicing_Jumping_C2



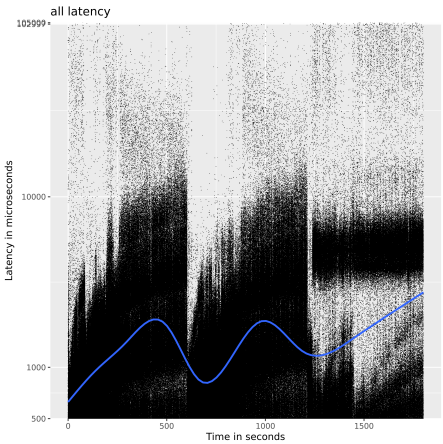
(a) Throughput for RandomSlicing_Jumping_dynamic_read_heavy



(b) Latency for RandomSlicing_Jumping_dynamic_read_heavy



(c) Throughput for RandomSlicing_Jumping_dynamic_write_heavy



(d) Latency for RandomSlicing_Jumping_dynamic_write_heavy

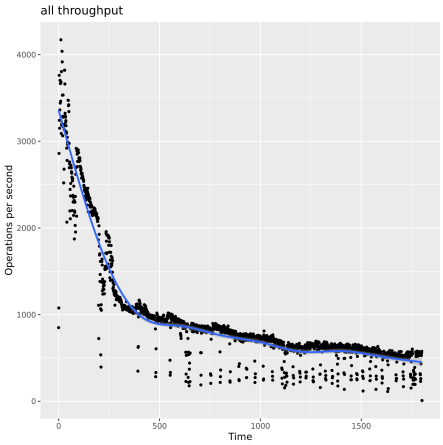
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	24643	869304	964743	0.037105016698236	0.27123479752024	0.271272898438739
n_1	28079	872526	967008	0.042278609092634	0.272240105810102	0.271909786309357
n_2	28920	875986	972730	0.043544904553544	0.27331967337153	0.273518736594424
n_3	112465	370913	410605	0.169338785982516	0.11572995459888	0.115456664068502
n_4	107697	65813	73525	0.162159598399138	0.020534560670605	0.020674251959028
n_5	104878	68097	75679	0.157915024196633	0.021247200066646	0.021279928106185
n_6	257460	82348	92066	0.3876580610773	0.025693707961998	0.025887734523765

(e) Stats for RandomSlicing_Jumping_dynamic_read_heavy

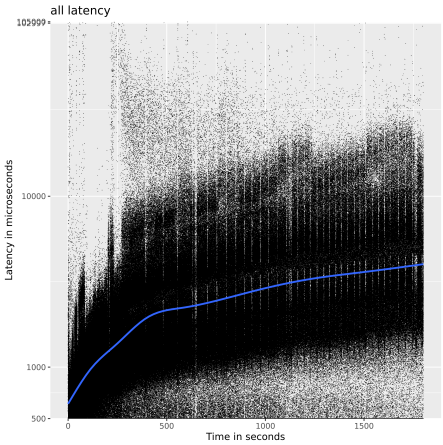
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	27345	925863	928801	0.040918599744418	0.271297356598424	0.271520569093403
n_1	24819	941721	944317	0.037138735675872	0.275944084549468	0.27605643108112
n_2	31224	947721	949339	0.046723070338991	0.277702210902493	0.277524534903131
n_3	117793	386757	387660	0.176263471190133	0.113327945652798	0.113326389414685
n_4	106602	49340	49215	0.159517446332215	0.01445765904304	0.014387242055006
n_5	104528	70878	71029	0.156413947488919	0.020768746608281	0.020764226677335
n_6	255967	90444	90378	0.383024729229452	0.026501996645495	0.02642060677532

(f) Stats for RandomSlicing_Jumping_dynamic_write_heavy

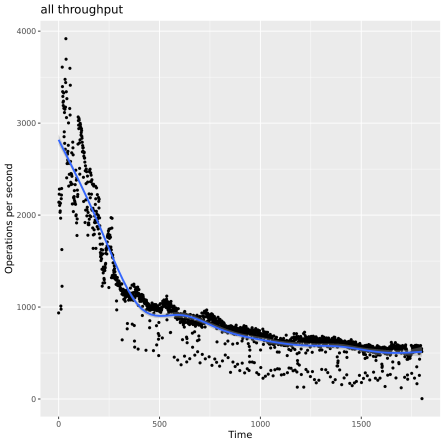
Figure B.8.: Evaluation Data for RandomSlicing_Jumping_dynamic



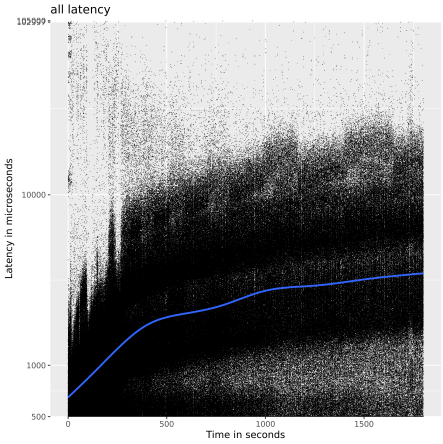
(a) Throughput for RandomSlicing_Random_C0_read_heavy



(b) Latency for RandomSlicing_Random_C0_read_heavy



(c) Throughput for RandomSlicing_Random_C0_write_heavy



(d) Latency for RandomSlicing_Random_C0_write_heavy

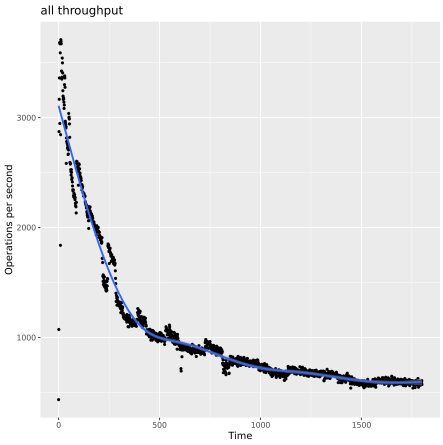
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	99974	814142	900522	0.333333333333333	0.333333333333333	0.333333333333333
n_1	99974	814142	900522	0.333333333333333	0.333333333333333	0.333333333333333
n_2	99974	814142	900522	0.333333333333333	0.333333333333333	0.333333333333333

(e) Stats for RandomSlicing_Random_C0_read_heavy

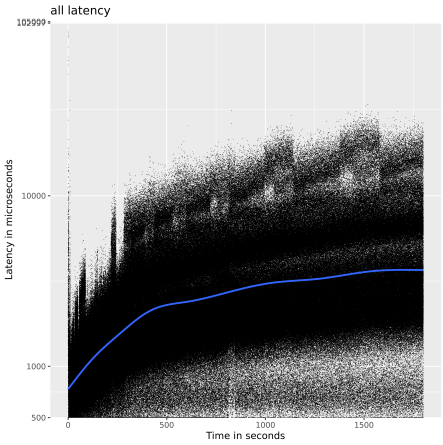
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	99989	847928	848633	0.333333333333333	0.333333333333333	0.333333333333333
n_1	99989	847928	848633	0.333333333333333	0.333333333333333	0.333333333333333
n_2	99989	847928	848633	0.333333333333333	0.333333333333333	0.333333333333333

(f) Stats for RandomSlicing_Random_C0_write_heavy

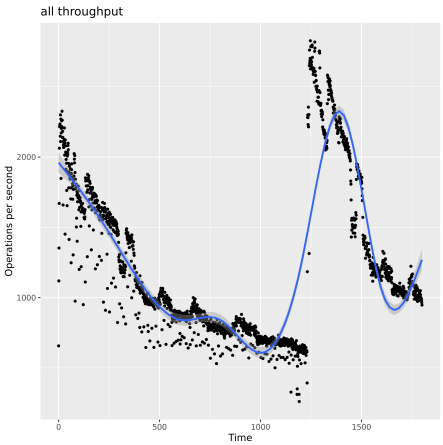
Figure B.9.: Evaluation Data for RandomSlicing_Random_C0



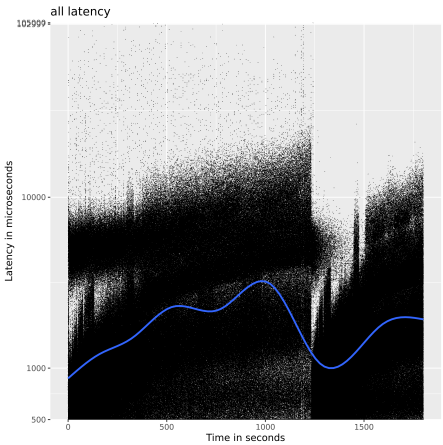
(a) Throughput for RandomSlicing_Random_C1_read_heavy



(b) Latency for RandomSlicing_Random_C1_read_heavy



(c) Throughput for RandomSlicing_Random_C1_write_heavy



(d) Latency for RandomSlicing_Random_C1_write_heavy

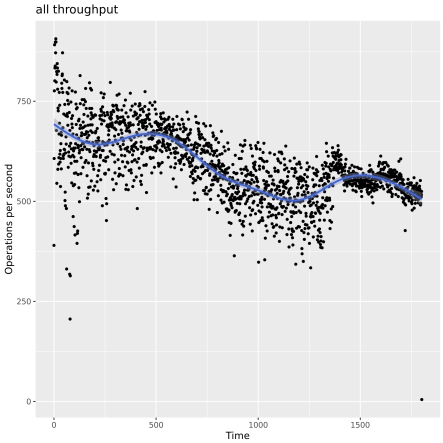
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	74959	650164	718092	0.249879980796927	0.25000278661487	0.25022258827945
n_1	75033	651151	719981	0.249903317864192	0.249897471790809	0.24955325958902
n_2	75008	649970	719419	0.250150024003841	0.250276835773831	0.250209729940122
n_3	74952	650438	720018	0.249879980796927	0.25000278661487	0.25022258827945

(e) Stats for RandomSlicing_Random_C1_read_heavy

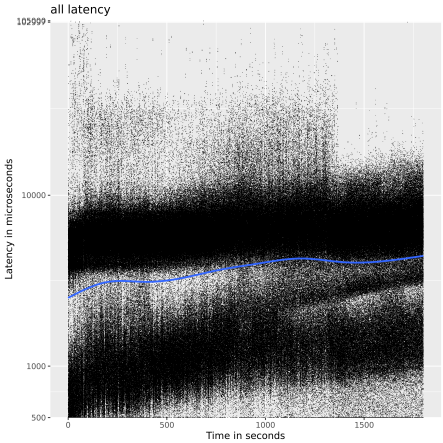
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	82088	834553	836678	0.242489409846332	0.250677190935704	0.250762843204514
n_1	82294	829793	831172	0.2430979375048	0.249247415440494	0.249112626257631
n_2	82486	830792	833040	0.243665108914635	0.249547488070686	0.249672489181129
n_3	91654	834056	835641	0.270747543734233	0.250527905553116	0.250452041356726

(f) Stats for RandomSlicing_Random_C1_write_heavy

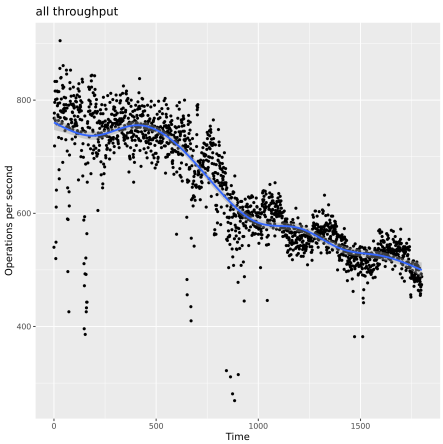
Figure B.10.: Evaluation Data for RandomSlicing_Random_C1



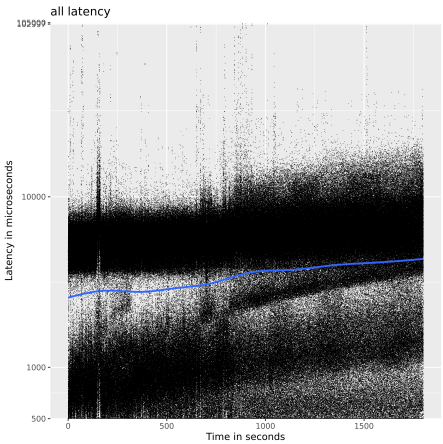
(a) Throughput for RandomSlicing_Random_C2_read_heavy



(b) Latency for RandomSlicing_Random_C2_read_heavy



(c) Throughput for RandomSlicing_Random_C2_write_heavy



(d) Latency for RandomSlicing_Random_C2_write_heavy

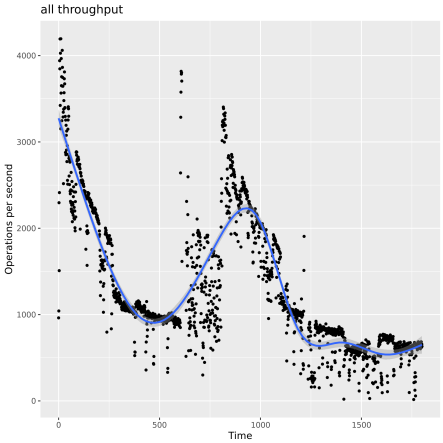
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	42747	213624	236175	0.140413748702519	0.142386953061575	0.142033067980421
n_1	43439	214490	238235	0.14268680445151	0.14296416864293	0.143271929502765
n_2	42998	214719	238144	0.141238224125925	0.143116804171949	0.14321720309571
n_3	43411	214392	237533	0.142594831097505	0.142898848634878	0.142849754362627
n_4	44702	212743	235339	0.146835459669684	0.141799739519805	0.141530306702421
n_5	43935	216377	240475	0.144316046722464	0.144221912063272	0.144619041061043
n_6	43204	213961	236916	0.141914885230393	0.14261157390559	0.142478697295012

(e) Stats for RandomSlicing_Random_C2_read_heavy

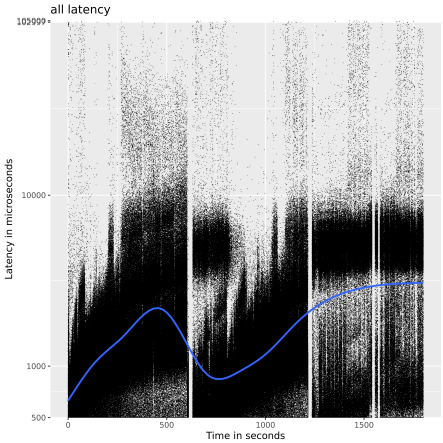
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	42702	244720	244854	0.142802680685421	0.14292756203145	0.142763187814631
n_1	42553	243089	243977	0.142304399587998	0.141974984172373	0.142251849156845
n_2	42881	245576	245814	0.143401286836015	0.143427504795012	0.143322919982788
n_3	42658	244531	245208	0.142655537274101	0.142817177472673	0.142969589051639
n_4	42546	243573	243794	0.142280990408925	0.142257662090088	0.14214515021229
n_5	43043	246280	246380	0.143943042123146	0.143838672675325	0.143652928740264
n_6	42645	244427	245079	0.142612063084393	0.142756436763081	0.142894375041543

(f) Stats for RandomSlicing_Random_C2_write_heavy

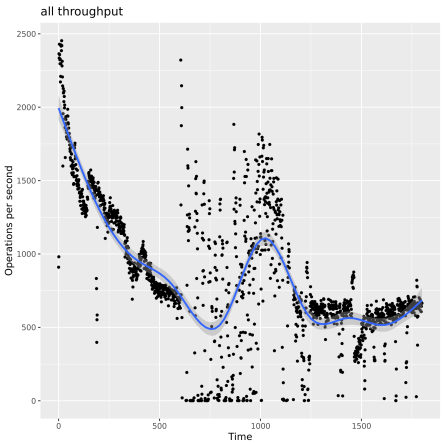
Figure B.11.: Evaluation Data for RandomSlicing_Random_C2



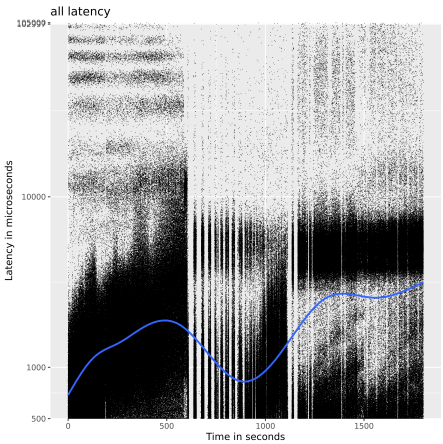
(a) Throughput for RandomSlicing_Random_dynamic_read_heavy



(b) Latency for RandomSlicing_Random_dynamic_read_heavy



(c) Throughput for RandomSlicing_Random_dynamic_write_heavy



(d) Latency for RandomSlicing_Random_dynamic_write_heavy

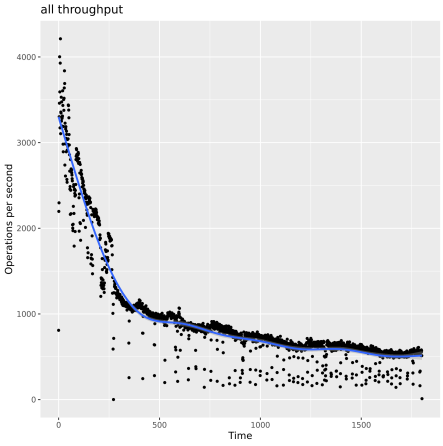
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	31746	884811	981011	0.045224090453879	0.27070369542052	0.270375649817574
n_1	31574	885058	982078	0.044979066086776	0.270779263889683	0.27066972482627
n_2	31970	886819	984810	0.045543191955223	0.271318033420844	0.271422689140943
n_3	112407	410361	456306	0.160130546703496	0.125547986130891	0.125762128320333
n_4	112711	66923	74071	0.160563613026749	0.02047477191019	0.020414648518353
n_5	120765	67784	75643	0.172037021472397	0.020738190743995	0.020847906169402
n_6	260798	66803	74407	0.37152247030148	0.020438058483876	0.020507253207126

(e) Stats for RandomSlicing_Random_dynamic_read_heavy

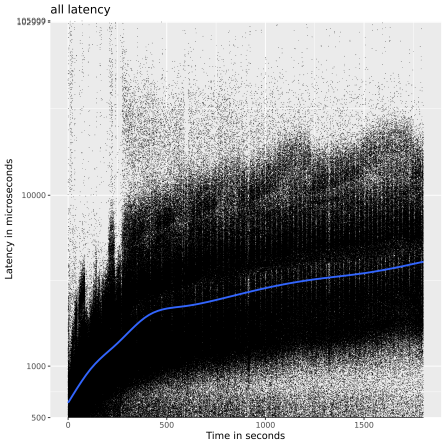
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	25814	574251	576663	0.042170505150833	0.266552325163099	0.267010819563282
n_1	25224	568472	570780	0.041206663900388	0.263869864205926	0.264286828859022
n_2	26034	569418	571349	0.042529903583202	0.264308972713537	0.264550291498954
n_3	108211	209804	209183	0.176776653477833	0.0973855405189	0.096857478750511
n_4	86457	76526	76253	0.141238683033453	0.035521371726704	0.035307234943388
n_5	115427	78115	77628	0.188564922059549	0.036258944050799	0.035943897737601
n_6	224967	77779	77843	0.367512668794741	0.036102981621035	0.036043448647242

(f) Stats for RandomSlicing_Random_dynamic_write_heavy

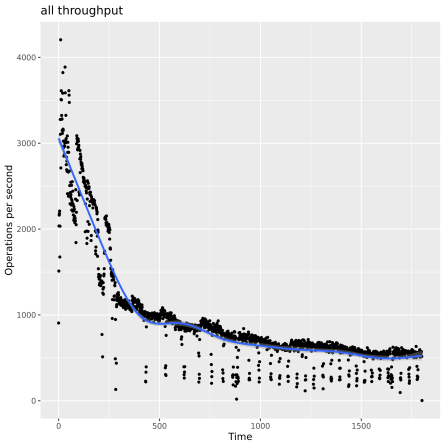
Figure B.12.: Evaluation Data for RandomSlicing_Random_dynamic



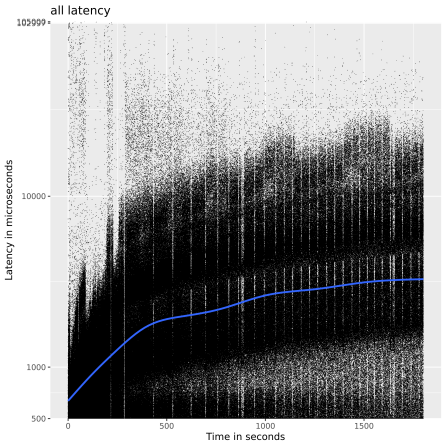
(a) Throughput for RandomSlicing_Rotation_C0_read_heavy



(b) Latency for RandomSlicing_Rotation_C0_read_heavy



(c) Throughput for RandomSlicing_Rotation_C0_write_heavy



(d) Latency for RandomSlicing_Rotation_C0_write_heavy

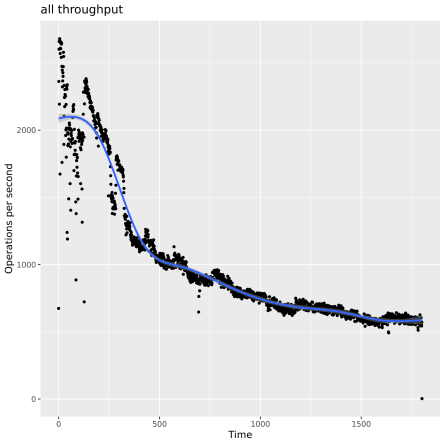
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	99975	820608	907767	0.333333333333333	0.333333333333333	0.333333333333333
n_1	99975	820608	907767	0.333333333333333	0.333333333333333	0.333333333333333
n_2	99975	820608	907767	0.333333333333333	0.333333333333333	0.333333333333333

(e) Stats for RandomSlicing_Rotation_C0_read_heavy

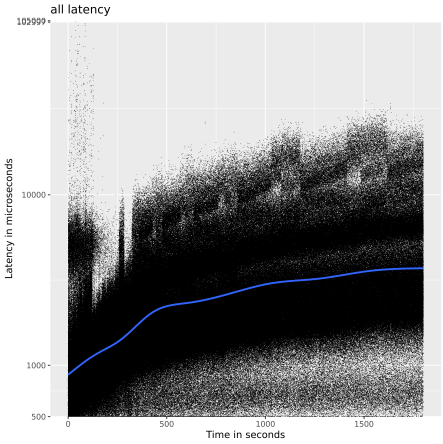
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	99989	849253	849965	0.333333333333333	0.333333333333333	0.333333333333333
n_1	99989	849253	849965	0.333333333333333	0.333333333333333	0.333333333333333
n_2	99989	849253	849965	0.333333333333333	0.333333333333333	0.333333333333333

(f) Stats for RandomSlicing_Rotation_C0_write_heavy

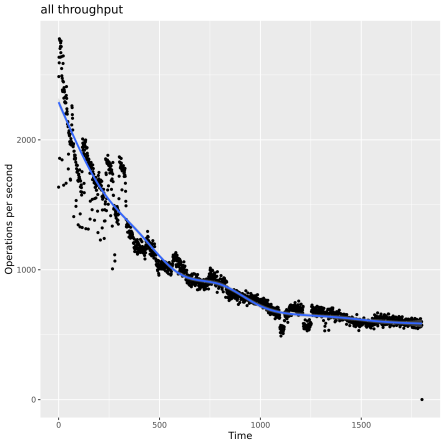
Figure B.13.: Evaluation Data for RandomSlicing_Rotation_C0



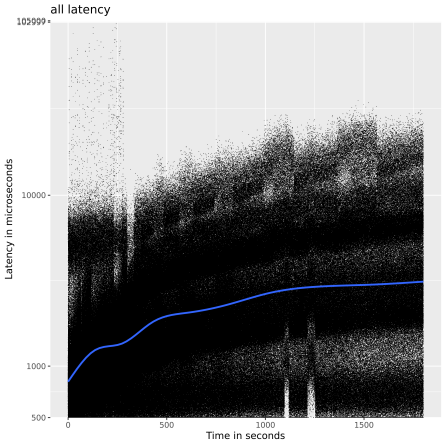
(a) Throughput for RandomSlicing_Rotation_C1_read_heavy



(b) Latency for RandomSlicing_Rotation_C1_read_heavy



(c) Throughput for RandomSlicing_Rotation_C1_write_heavy



(d) Latency for RandomSlicing_Rotation_C1_write_heavy

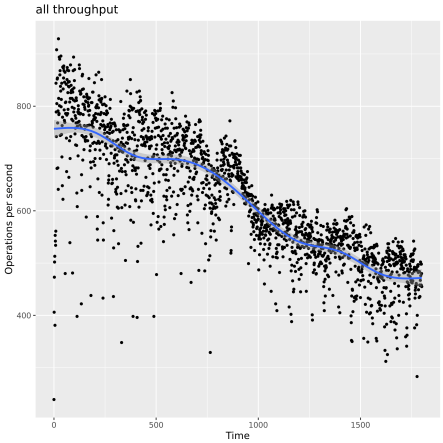
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	84704	685866	758802	0.273370103695002	0.267182593431867	0.267262475855569
n_1	81851	681733	753899	0.264162452275449	0.2655725622324	0.265535559058934
n_2	80912	678936	750790	0.261131963427583	0.264482976637212	0.264440518406122
n_3	62384	520496	575673	0.201335480601967	0.20276186769852	0.202761446679375

(e) Stats for RandomSlicing_Rotation_C1_read_heavy

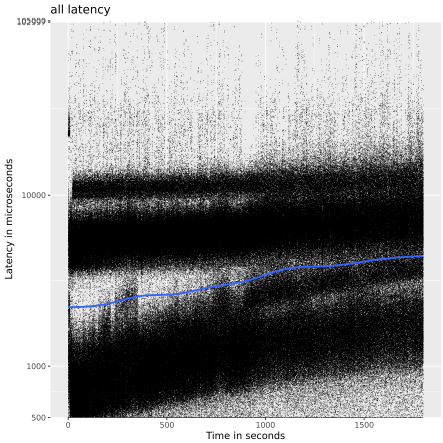
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	81209	694176	694504	0.258677641197812	0.263567533826998	0.263535675960372
n_1	86677	714841	715640	0.276095037570993	0.271413704087185	0.271555917812253
n_2	83164	707695	708135	0.264904965614339	0.268700482084799	0.268708079285646
n_3	62889	517057	517053	0.200322355616855	0.196318280001018	0.196200326941729

(f) Stats for RandomSlicing_Rotation_C1_write_heavy

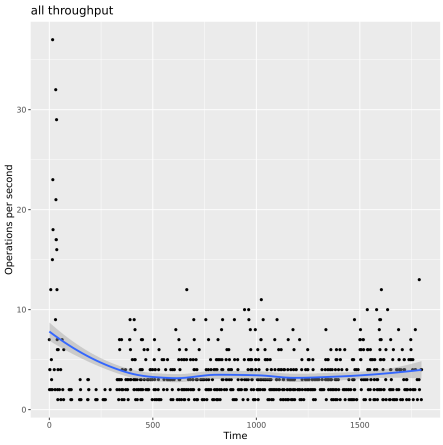
Figure B.14.: Evaluation Data for RandomSlicing_Rotation_C1



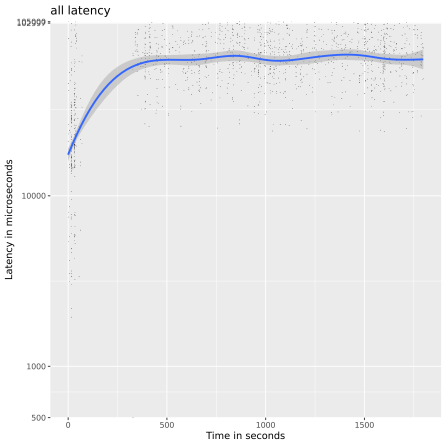
(a) Throughput for RandomSlicing_Rotation_C2_read_heavy



(b) Latency for RandomSlicing_Rotation_C2_read_heavy



(c) Throughput for RandomSlicing_Rotation_C2_write_heavy



(d) Latency for RandomSlicing_Rotation_C2_write_heavy

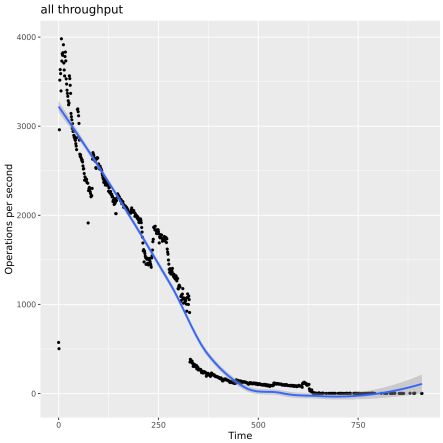
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	55653	293997	325468	0.183501272734467	0.185230314825444	0.184974501699034
n_1	42723	225899	250830	0.131480724337585	0.131388857211802	0.131738664153873
n_2	43867	231549	256360	0.144640007385817	0.145885482394435	0.145698081702547
n_3	37376	195957	218046	0.123237625459965	0.123461044848245	0.123922936194857
n_4	39876	208540	231798	0.131480724337585	0.131388857211802	0.131738664153873
n_5	46390	235751	260606	0.15295894277311	0.14853291683389	0.148111227493267
n_6	37399	195504	216421	0.123313461969639	0.123175636042659	0.122999393587716

(e) Stats for RandomSlicing_Rotation_C2_read_heavy

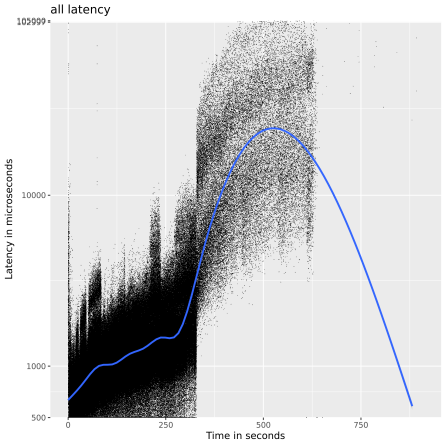
Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	722	739	734	0.187825182101977	0.188616641143441	0.184375784978649
n_1	562	579	599	0.150624349635796	0.151097498723839	0.144184878171314
n_2	584	609	636	0.151925078043704	0.155436447166922	0.159758854559156
n_3	492	497	496	0.127991675338189	0.126850433894844	0.124591811102738
n_4	579	592	574	0.150624349635796	0.151097498723839	0.144184878171314
n_5	498	504	508	0.129552549427679	0.128637059724349	0.127606129113288
n_6	407	398	434	0.105879292403746	0.101582440020419	0.109017834714896

(f) Stats for RandomSlicing_Rotation_C2_write_heavy

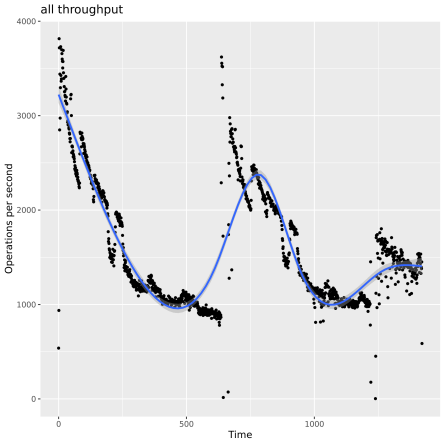
Figure B.15.: Evaluation Data for RandomSlicing_Rotation_C2



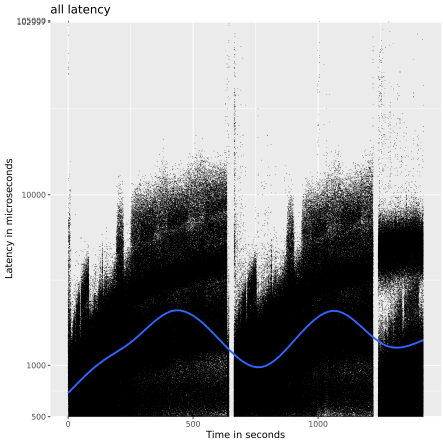
(a) Throughput for RandomSlicing_Rotation_dynamic_read_heavy



(b) Latency for RandomSlicing_Rotation_dynamic_read_heavy



(c) Throughput for RandomSlicing_Rotation_dynamic_write_heavy



(d) Latency for RandomSlicing_Rotation_dynamic_write_heavy

Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	41	351581	390333	0.000139504656393	0.333224337685577	0.333221785040179
n_1	17	351623	390377	5.78433941142645E-05	0.333264144791714	0.333259347220527
n_2	16	351654	390426	5.44408415193078E-05	0.333293526227196	0.33330117783046
n_3	97014	212	241	0.330095237447133	0.000200931107168	0.000205738305997
n_4	1839	2	0	0.006257294222125	1.89557648271992E-06	0
n_5	965	8	6	0.003283463254133	7.5823059308797E-06	5.1221155019972E-06
n_6	194005	8	8	0.660112216184582	7.5823059308797E-06	6.82948733599626E-06

(e) Stats for RandomSlicing_Rotation_dynamic_read_heavy

Node	Keys	Puts	Gets	Key Load	Put Load	Get Load
n_0	0	0	0	0	0	0
n_1	0	0	0	0	0	0
n_2	0	0	0	0	0	0
n_3	115939	347491	348280	0.201817662766287	0.756821919001623	0.756727395191691
n_4	91227	42302	42325	0.201817662766287	0.756821919001623	0.756727395191691
n_5	108757	38074	38109	0.158800920494226	0.09213211512703	0.091961889863008
n_6	258551	31278	31531	0.189315791489258	0.08292369512899	0.082801551347652

(f) Stats for RandomSlicing_Rotation_dynamic_write_heavy

Figure B.16.: Evaluation Data for RandomSlicing_Rotation_dynamic

Configuration	Load Divergence Keys	Load Diversion Puts	Load Diversion Gets
ConsistentHashing_C0_read_heavy	0.006921453505236	0.006979733904974	0.007010513601979
ConsistentHashing_C0_write_heavy	0.00691846962474	0.006895919119033	0.006898016476352
ConsistentHashing_C1_read_heavy	0.000224229450913	0.000209786316987	8.21717050967369E-05
ConsistentHashing_C1_write_heavy	0.000220862045399	0.000205317147945	0.000214860272345
ConsistentHashing_C2_read_heavy	0.003873042028007	0.003872618069355	0.003944093672988
ConsistentHashing_C2_write_heavy	0.003814258278993	0.00381344061015	0.003954046888868
ConsistentHashing_dynamic_read_heavy	0.004728892150533	0.128410252529309	0.128331440935386
ConsistentHashing_dynamic_write_heavy	0.003871936985801	0.108702573347419	0.108640772566759
RandomSlicing_Jumping_C0_read_heavy	0	0	0
RandomSlicing_Jumping_C0_write_heavy	0	0	0
RandomSlicing_Jumping_C1_read_heavy	0.00048598398837	0.000421769383081	0.000406824323717
RandomSlicing_Jumping_C1_write_heavy	0.000491735509638	0.000469689877308	0.000465369398517
RandomSlicing_Jumping_C2_read_heavy	0.04685224545925	0.046800333023621	0.046805175340765
RandomSlicing_Jumping_C2_write_heavy	0.020522730815766	0.020604399983858	0.020698734954357
RandomSlicing_Jumping_dynamic_read_heavy	0.087326542350576	0.110920899465841	0.110894283648883
RandomSlicing_Jumping_dynamic_write_heavy	0.086797435089185	0.113249206708273	0.113294316144636
RandomSlicing_Random_C0_read_heavy	0	0	0
RandomSlicing_Random_C0_write_heavy	0	0	0
RandomSlicing_Random_C1_read_heavy	0.000121686136449	9.62343031905011E-05	0.000275411727501
RandomSlicing_Random_C1_write_heavy	0.010373771867117	0.00060254824441	0.00060744228062
RandomSlicing_Random_C2_read_heavy	0.001553491622246	0.000506617738417	0.000724784310883
RandomSlicing_Random_C2_write_heavy	0.000465726641393	0.000463517408674	0.000403211825046
RandomSlicing_Random_dynamic_read_heavy	0.083664308593015	0.109779875474177	0.109684752918102
RandomSlicing_Random_dynamic_write_heavy	0.086937947360199	0.104617066717467	0.104936146099951
RandomSlicing_Rotation_C0_read_heavy	0	0	0
RandomSlicing_Rotation_C0_write_heavy	0	0	0
RandomSlicing_Rotation_C1_read_heavy	0.024332259699017	0.02361906615074	0.023619276660313
RandomSlicing_Rotation_C1_write_heavy	0.024838822191572	0.026840859999491	0.026899836529136
RandomSlicing_Rotation_C2_read_heavy	0.016349261377966	0.016155923085201	0.015891613665953
RandomSlicing_Rotation_C2_write_heavy	0.019245471341502	0.020903001323041	0.018347354013196
RandomSlicing_Rotation_dynamic_read_heavy	0.201283762262123	0.16320302289516	0.163203109005639
RandomSlicing_Rotation_dynamic_write_heavy	UNDEF	UNDEF	UNDEF
Overall_ConsistentHashing	0.003821643008703	0.032386205130647	0.032384489514972
Overall_RandomSlicing_Jumping	0.030309584151598	0.036558287305248	0.03657058797636
Overall_RandomSlicing_Random	0.022889616527552	0.027008232485792	0.027078968645263
Overall_RandomSlicing_Rotation	0.040864225267454	0.035817410493376	0.035423027124891

Table B.1.: Load Balance