F^2DR: A Distributed Hash Table Algorithm

by

Dana Rea

A Thesis
presented to
The University of Guelph

In partial fulfilment of requirements
for the degree of
Master of Science
in
Computer Science

Guelph, Ontario, Canada

ABSTRACT

F^{2}DR: A Fast and Fault-Tolerant Routing Protocol for Distributed Hash Tables

Dana Rea	Adviser:
University of Guelph	Professor David A. Calvert

Considerable research has been directed toward the study of consistent hashing and distributed hash tables. While many useful research results have emerged from this work, existing solutions can be improved in the areas of time efficiency, system growth and change to input distributions. For resolution, systems such as Chord [69][70], Memcached [23] and others use a binary search over the set of intervals to determine the node. Also, relying on a pseudo-random designation of partitions on the continuum can result in poor worst-case time performance due to load imbalance. The work proposes F^{2}DR, a system that maps an arithmetic distribution of intervals on a continuum to a fluid set of nodes. Any point on the continuum can be resolved to a node in O(1) time, and O(n) space. The system also contains flexible mechanisms for adapting to load patterns through dynamic restructuring. In all, F^{2}DR provides a fresh formulation of consistent hashing that offers several advantages over previous work.

ACKNOWLEDGEMENTS

I would like to thank my adviser, Dr David A. Calvert for his resolute patience during the writing of this thesis and completion of my degree. He gave me wide lattiude in my work, but knew when to provide motivation to meet goals.

From the School of Computer Science, I would like to thank my committee for their important advice, especially Mieso Denko, who was only able to make a brief, but essential contribution to this work. I would also like to thank Dr. Blair Nonnecke for his mentorship and friendship.

Finally, a thank you to my family, and to Katharine Tuerke, this work would have been impossible without your encouragement and example of dedication to your work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: Introduction

1.1 Topic

Algorithms for information partitioning have long been an area of interest in computer science, and before that, any discipline concerned with the arrangement or storage of physical objects. The work is concerned with the partitioning of an online stream of data among a fluid set of computing resources. The resulting function for assigning partitions to resources is monotonic (strictly increasing, or decreasing), requires minimal space (all 'buckets' are used), has no collisions, can be computed in constant time and allows for graceful restructuring if the number of computing resources change. The resulting algorithm is applicable in the areas of distributed computing, caching, network routing, message queues and others.

1.2 Research Problem

Distributed storage is an essential requirement for computer systems that need to be both scalable and performant. A number of studies and commentators have linked website performance with usability and customer satisfaction [6][45][52][54]. In particular, it has been found that users consider abandoning the use of websites when load times are just a few seconds [45][52]. Using A/B testing Google and Amazon correlated slower page loads with a significant decrease in their conversion rates [52]. Another important concern is predictability of performance across the range of request and user types. For example, pages in the Amazon e-commerce application rely on scores of service requests to gather data [16], and poor performance by any of these can result in lost sales.

Many large distributed systems exist where subsets of a large quantity of data are retrieved by clients. In some of these systems, the number of clients is large and each client may make requests for data at a high rate. Some common examples of such systems are the websites Facebook [62], Live Journal [25], Amazon [16], Yahoo [68] and Slashdot [51].

The system developed in this work implements a distributed look-up protocol based on Distributed Hash Table (DHT) systems like Consistent Hashing [35][36] and Chord [69][70] and others. Also introduced is a novel interval division and constant time node look-up technique that draws inspiration from Bonwick's slab allocation algorithm [8]. Like Chord [69][70], data entries are looked up via keys. Nodes may join and depart the system at any time with a minimal disruption to the key to node mapping. This is due to the monotonic behaviour of consistent hashing functions [35][43].

1.3 Thesis Objective

The thesis objective is to show that a consistent hashing system with hundreds of intervals assigned to many clients can be constructed such that keys resolved to nodes in constant time and load among nodes can be dynamically managed. This will be accomplished though a consistent hashing look-up function that runs in constant time with a small scalar. Also proposed is a means of maintaining a uniform distribution of keys among nodes where there is flexibility in how uniformity of load is defined. If these goals are achieved then node look-up latency can be described with stricter bounds due to the time complexity of the look-up algorithm and the tighter bounds on load achieved by the restructuring system.

Chapter 2: Background

2.1 Overview

Distributed hash tables have become a mainstay for large applications, due to their ability to cope with high server load and failures. These systems can be decentralized by applying a common routing mechanism to the key. The key is then used to assign the value to a server [19][23][24][28][33][35][36][70][80]. This avoids the need for a central load balancing component, which can be a single point of failure and performance bottleneck. This works very well for systems like web applications such as web servers are often also distributed. This works well in situations where many servers send requests for the same data. Adding additional servers for load balancing the cache adds more infrastructure complexity and additional points of failure.

A simple way to implement a distributed caching system is to perform a separate hash on the key used in the request. This hash will determine the caching server that will be contacted [19][23][24][28][33][35][36][70]. This is done by dividing the key domain into equal portions to be assigned to each server in the cluster. Fitzpatrick describes this approach as using the cache servers in the cluster as 'virtual buckets', which mimic the hash table structure that is used to actually store the data [10][18][23][24][50][70].

Since it is important that the entries be distributed evenly [18][19][23][24], the statistical distribution of values over the key domain is of great importance [16]. If not, the result could be undesirable, where too many entries are be assigned to one cache server, potentially increasing the response time for those entries and leaving other servers underutilized. The work of Karger et al, which introduces *consistent hashing*, provides an excellent solution in which data is distributed among many nodes and failure is handled gracefully [35][36][43]. This approach has been incorporated into a number of well-known distributed storage systems, such as Chord [69][70], Memcached [33], Dynamo [16], Redis [64], which power major web applications such as Live Journal [48], Stack Exchange [7], LastFM [33], Yahoo [68] Slashdot [67], and Facebook [22].

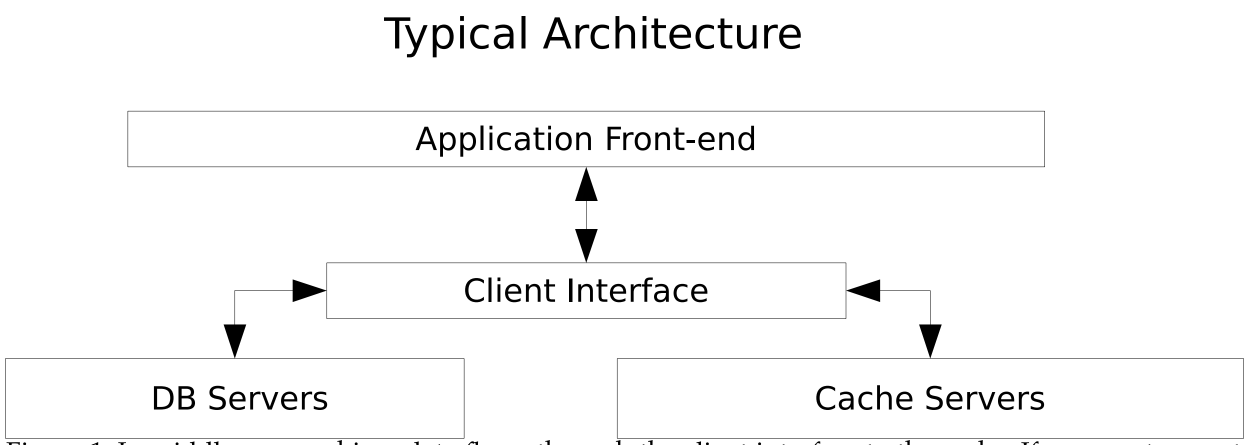

Figure 1. In middleware caching, data flows through the client interface to the cache. If a request cannot be served by the cache, it can be redirected to the database.

Relational Database Management Systems (RDBMS) systems remain a tool of choice for maintaining consistent, large scale stores of data. The consistency guarantees afforded by such systems are computationally expensive [23][24][58][66]. This focus on persistent storage relies heavily on slower hard disk access. Although hard disk storage does provide reliable storage [46][79]. A common approach to allow this solution to scale well has been to use database clusters to provide more servers to store and retrieve data, thus lightening the server load on each. However, this partitioning of the work can have the drawback that the same work is being done by separate machines, and that maintaining concurrency in a distributed environment is expensive. However, a number of approaches are being taken to resolve these and other problems with traditional systems [34][63][72][74] as in 1. As an alternative, distributed caching provides a more effective use of this hardware by maintaining a quickly accessible copy of the resulting data. With a suitable implementation, developers can exert fine control over when data is stored in memory, for how long, and in what format. This takes advantage of the fast access times provided by memory, and greatly reduces the computational time required for look-ups.

Some commonly supported operations in a caching layer, are 'create', 'retrieve', 'update' and 'delete' [4][23][24]. This paradigm is also referred to as CRUD. There are several different strategies for allowing these operations. The first is for the front end to directly access the hash table using a single layer of depth, so that the hash key used corresponds directly to the area in memory where the cached entry resides. While this method arguably provides very low data structure overhead, there are several drawbacks to this approach. As John Hamer points out in *Hashing Revisited* [31], the numerical domain of the keys can get very large, even in a simple scenario. This means that the interface to the cache will need to manage compression of the key domain, which will need to be re-implemented in each application using the caching system. Alternatively, the cache could dynamically allocate space to the hash table as the load factor increases, however this approach entails much higher overhead for memory management due to allocation resizing, object creation or deletion and external memory fragmentation [8][23][24][44].

Strings of 10 characters are at the low end of what should be used as a hash key. For example, this would not be long enough to contain a person's first and last name concatenated together. But the number of entries required is many orders of magnitude greater than can be stored in a server with gigabytes of memory available. This example is somewhat far-fetched because it is unlikely this hash function would be used, but the example does emphasizes the need for compression of the hash key domain, and a more suitable hash table implementation.

A technique that is more common, is to have a second layer of hashing that transparently manages the final assignment of hash keys [23][38][46]. The problem of hash key domain size is often dealt with using the consistent hashing algorithm first described by Karger et al [35][36] or by wrapping the key back into a smaller domain size using a technique such as modular arithmetic [31]. This is only a trade-off however, using a smaller domain, even one that is still larger than the number of cache entries expected, will result in some key collisions [31].

2.2 Shared Nothing

The name was coined by Michael Stonebraker to describe systems that divide the problem space among nodes in such a way that state is maintained independent of other nodes [71]. In this type of architecture neither CPU, memory or storage need be shared among nodes, and system communication is performed on an interconnect [55], such as Ethernet or fibre channel. Instead, nodes in the system work in parallel using a common specification, not unlike a factory where independent workers produce identical widgets. However, in common practice this may not be strictly true for several reasons:

- The system resides on a central network storage area.
- Several instances of the system run concurrently within a single operating system.
- Where instances of the system reside within a virtualized computing environment where hardware is shared.
- Where explicit co-operation between clients is required in order to maintain system state.

Stonebraker also provides a rough evaluation of shared nothing against shared memory and shared disk systems for criteria such as concurrency control, high availability and load balancing. The shared-nothing approached faced some initial criticisms about the scalability of relying on an interconnect [55], but this has long since been muted by the progress on interconnect performance and focus on "embarrassingly parallel" computing problems.

2.3 Fault Tolerance and Data Consistency

In distributed systems, fault tolerance is the capacity of a system to continue to produce correct results in the face of programming error, component failure, or malicious intent. A common problem to solve when designing systems that must co-operate is illustrated by the "Byzantine Generals Problem" [11][40][41]. In short, when nodes must communicate with each other to maintain system state, there must be a mechanism to evaluate whether messages are both trustworthy and accurate. This mechanism should be resilient against data faults from the benign to the deranged.

In relational database management systems (RDBM's), this problem is dealt with by conforming to the ACID properties, which are atomicity, consistency, isolation and durability [29]. There is contention in both research and industry whether ACID should be strictly adhered to [71][72] or weakened in order to gain performance [17][26][56]. Where careful locking and defensive algorithm design is required to maintain ACID, this safety can sacrifice performance for data accuracy. Where some data inaccuracy is tolerable or certain ACID principles are not exercised, these guarantees can be weakened or discarded entirely. At present, it appears that no consensus in this matter is forthcoming, and both approaches remain firmly entrenched.

Nevertheless, whether the guarantees provided by ACID are necessary, it is still necessary to protect properties such as consistency. In a distributed system such as a distributed hash table (DHT), large systems will likely experience a 'network split' where some nodes become unavailable. In the event that these nodes rejoin the system, it can be more efficient if the joining nodes can preserve stored data, rather than requiring each entry to be entered again. The issue is, that while these nodes were absent, updates to this data could have been made on other nodes, so upon rejoining the system, these nodes may contain inconsistent entries. To overcome this problem, versioning schemes have been implemented that allow events to be ordered, where more recent events can be given priority [16][40][41]. An alternative approach attempts to retain ACID, without the performance penalties associated with applying locking across distributed nodes [74].

In more complex situations where there are disjunct views of the network, other strategies can be employed to reach quorum, even when malicious adversaries attempt to undermine this process. This case is illustrated by the Byzantine Generals Problem. A number of generals in the byzantine army must agree upon a sound battle strategy via messenger but some generals could be traitors. The traitors will act to either, prevent the group of generals from coming to any agreement, or cause an unsound plan of attack to be chosen. The authors show that the traitors will be successful if, for `m` traitors, there are less than `3m + 1` total generals [57]. However, this approach could be expensive for large system because communication between many neighbour nodes is required for consensus to be achieved on each message. As many scenarios require nodes increase in degree and number, this type of solution is intractable. A later extension of this approach, called Practical Byzantine Fault Tolerance (BZFT) [11] was created to handle this problem and provides similar performance to system NFSv2 [11] which is not byzantine fault tolerant. The BZFT algorithms are implemented by [1][16] and others to resolve consistency issues with data when an item is mirrored between multiple nodes.

2.4 Consistent Hashing

The Consistent Hashing algorithm, proposed by Karger *et al.* [35][36], was designed to cache internet content in a way that was fast, fault tolerant and resistant to swamping. This algorithm has several properties that are useful for distributed caches or DHT's [35][36][23]. In the situation where a server is added or removed from a distributed cache, only a small number of entries must move between buckets. In comparison, many entries will move between nodes when using a modular arithmetic hash function [35][43]. This property makes the cache resilient against performance degradation when the number of buckets or nodes change, due to scheduled maintenance, increased application load or unplanned outages [18][19][23][24][33][35][36]. This is because the function for mapping keys to caches is monotonic [33][35][36]. Another helpful property is that keys can be mapped to caches quickly, in logarithmic time for standard algorithm, and an alternate approach running in constant time. In the analysis of this algorithm and others in this work, we use the well known 'Big-O' asymptotic limit notation.

As a companion for consistent hashing, Karger et al introduce *random trees*, which can provide support for a DHT for data replication [35]. This builds upon the work of Chankhunthod et al [12], and Plaxton & Rajaraman [59]. The work of these two groups is expanded upon to provide a better distribution of cache requests, reducing the load in individual cache servers. Where the work of Karger et al differs is that their primary goal is to create a distributed tree hierarchy of cache nodes [35][36]. In this scheme, one cache server in the cluster is considered to be the primary source or 'root' for a particular cache entry, as defined by the hash key [35][36]. The assignment of each entry to a particular server is done at random, the hope being that one primary source will not contacted for many different pages at once. When a primary source receives a request, and it does not possess the requested entry, then the next node in the tree is contacted, and the process is repeated.

The monotonic mapping function in consistent hashing maps a key to a cache node, independent of the number of cache nodes in the system. In other words, a mapping, or assignment, of a key `k` to cache node `n`, will remain mapped to `n` until that node departs the system. To do this, the authors define a circular continuum from the set of integers `[0, x)`, which is divided into 'intervals. These intervals are variable length chords on the continuum 'circle', each of which are assigned to a cache node. This is similar to the concept of hash table 'buckets', however intervals usually contain multiple values in the range while buckets are usually defined as containers for single values.

Using the approach in Plaxton & Rajaraman [59], the start points of these intervals are dispersed at random over the continuum. To store an item in the cache, an integral key for the item must be defined on the continuum, and the item assigned to the node with the interval ending 'closest' to the key. In other words, points on the continuum correspond to separate nodes. Items are assigned to the node with the first point that follows their key. There is also an optional step to select a separate hash function for each key. This increases the independence of each possible key to better balance between nodes [43]. This secondary hash process is important for load balancing but not central to DHT functionality, so this will omitted from the analysis for simplicity.

To provide key look-ups in the system, the authors use a balanced binary search tree to store the mapping between each key and node. Therefore, the time complexity of adding, removing and retrieving a key are $O(log_2kn)$, where k is the number of keys and n the number of nodes [43].Both of these variables must be considered since either may dominates, depending on the composition of the system. In practice, it is more common to have systems with many more keys than nodes. Creation of the tree requires $O(knlog_2kn)$ thus adding and removing nodes requires $O(log_2kn)$ [43].

In the alternate consistent hashing algorithm, the authors achieve constant time look-up, rather than logarithmic. To do this for a single hash function, they divide the continuum into intervals of equal length, then the node to which a key belongs can then be determined in constant time via interpolation. The length of each interval is defined by (1). As additional nodes are added, new intervals are created by bisecting existing ones. For the constant time algorithm, the authors do not provide a complexity analysis of the other operations.

$$\left(\frac{1}{2^x}\right) \leq \left(\frac{1}{v}\right), \text{ where } x \in \mathbb{N} \wedge v \text{ is the number of buckets. } [43] \tag{1}$$

Where k represents the number of keys and n intervals on the continuum:

Operation	Complexity	
	Standard	Improved
Storage Overhead	$O(kn)$	$O(kn)$
Construction	$O(kn\log_2 kn)$	$O(kn\log_2 kn)$
Resolve	$O(\log_2 kn)$	$O(1)$
Join	$O(\log_2 kn)$	$O(\log_2 kn)$
Depart	$O(\log_2 kn)$	$O(\log_2 kn)$

Table 1. Summary of Consistent Hashing time complexities summarized from section 2.4 Consistent Hashing.

2.5 Chord

Chord is a straight forward and efficient routing system for distributed key-value stores [69][70]. This system is a successor to [35], that retains the hashing characteristics, but uses a tree search where each node is only aware of a roughly logarithmic proportion of assignments. Chord assigns intervals on a continuum to nodes, such that there at most 2^m nodes and a range of 2^m-1, (where $m >= n$). In other words, there must be at least one interval assigned to each node [69][70].

Locating the node for an arbitrary key k requires that $O(log(n))$ nodes be contacted [69][70]. For network models it is necessary to consider the latency (RTT), required to receive a response from each node. There will also be some look-up overhead to locate the entry corresponding to k. If nodes contain roughly the same number of entries, the total time complexity is $O(RTT*log(n) + log(k/n))$

(assuming a binary search tree) [69][70]. This approach is practical for systems with many thousands of nodes, where frequent joins and departures occur. Others have attempted to reduce this look-up cost to a constant number of hops but replication of entries across many nodes is needed [60]. For node joins and departures, since each node only need make a logarithmic number of interval assignments, the time complexity for the node to make its interval assignments is $log(I) * log(I)$ [69]. It takes and unknown number of RTT's to use the system's gossip operation to communicate the assignments to other nodes [69], so only a lower bound can be defined.

The authors' solution is geared toward clients that participate as nodes, or can request that nodes perform look-ups. In the former case, by design, each node has a disjunct view of the network, so the number of queries needed to resolve a request is non-uniform between nodes. Depending on the round trip time (RTT) between nodes, the response time for a request will take longer on some nodes than others. This approach is unsuitable for situations where nodes are dispersed over a network and fast response times are needed. In the later case, the issues arise with the complexity of balancing clients between nodes, and increasing the number of nodes so the system does not become overloaded by too many clients. Reducing load by providing more nodes is common, but the addition of each introduces a small performance penalty, from increasing the maximum number of queries needed to resolve a request. These are acceptable trade-offs for Peer-to-Peer (P2P) systems where scalability is crucial, but impractical for systems requiring consistent performance, low latency and a high request rate.

Where I represents the number of intervals on the continuum, N represents the number of nodes, n is the number of intervals assigned to each node and RTT represents the time complexity of communication between nodes:

Operation	Complexity
Construction	*Not available*
Resolve	$O(RTT * log_2(n) + log_2(I/N))$
Join	$o(log_2^2(I) * RTT)$
Depart	$o(log_2^2(I) * RTT)$

Table 2. Summary of Chord time complexities, summarized from section 2.5 Chord.

2.6 Memcached

Memcached is a DHT system originally developed at Live Journal [33] for caching RDBMS results. Memcached has since be used for storage in many applications, even for industrial applications [58]. Support for consistent hashing was later added by LastFM [33] in order to provide better scalability and fault tolerance. Unlike Chord, Memcached [24] is geared toward a system where clients almost always have a consistent view of the entire system. This allows queries to be processed more quickly as only a single node must be contacted. However, when there are differing views in the system, there are no mechanisms in place to verify data consistency.

There are hash implementation strategies of interest, modulo and consistent hashing. In the former, Memcached uses the naive strategy of adding nodes to a numbered list and resolving keys to the node number that is the modulus of the key by the number of nodes. In other words, a key k is assigned to one of N nodes n_i where:

$$n_i \equiv N \bmod k \tag{2}$$

This construction step requires $O(intervals)$ time, as it is necessary to iterate through each interval and the modulo operation to select the node requires constant time.

This approach provides $O(1)$ resolution of a key to a node [23][24], however if there is a node join or departure then the majority of key-node assignments change [43]. Repopulating all of the cache entries for a large system will be prohibitively expensive.

Adding and removing nodes is also fast since a node has only to be appended or removed from a list and intervals to be removed from the continuum. This requires $O(1)$ time proportional to the fixed interval count [23][24].

For the consistent hashing strategy each node is assigned a fixed number of intervals, i, at random points on the continuum $[0,\ 2^x)$. These allocations are arranged in order of the points defining the continuum. To resolve a key to a node, a binary search is performed on the list of assignments for the containing interval, requiring logarithmic time. Adding nodes is accomplished by defining a further i random intervals on the continuum. This results in existing intervals being subdivided when a new interval overlaps it. When removing nodes, the assigned intervals are destroyed and an adjacent interval is expanded to absorb unallocated parts of the continuum. As a binary search is needed to perform the resolve operation, a sort is needed after a node joins or departs. This is can be accomplished in polylogarithmic time based on the number of intervals assigned to each node and the number of nodes. For construction, the initial assignments can be performed sequentially, followed by a sort. The time complexity function is that of join times the total number of intervals.

Where I is the number of intervals, N is the number of nodes and n is the number of intervals assigned to each node:

Operation	Complexity	
	Modulus Hashing	With Consistent Hashing
Construction	$O(I)$	$O(Inlog(N))$
Resolve	$O(1)$	$O(log(I))$
Join	$O(1)$	$O(nlog(N))$
Depart	$O(1)$	$O(nlog(N))$

Table 3. Summary of Memcached time complexities, summarized from section 2.6 Memcached.

2.7 Dynamo

Dynamo [16] is a key-value store developed as part of Amazon Web Services (AWS) [2], which uses consistent hashing to provide route storage and retrieval requests. Amazon's initial requirements for Dynamo were a system providing scalable, reliable service, that degrades gracefully and adheres to a strict service level agreement for latency and availability [16]. In production, this system has serviced millions of requests per day and achieved an uptime of '5 nines' (i.e. downtime of several hours per year, or available for 99.999% over a calendar year) [16].

Dynamo also includes several algorithms to aid in data consistency and fault tolerance. Instances exist in DHT systems where multiple versions of the same entry can exist. Perhaps a network split has occurred where clients are unable to reach all nodes in the network, or multiple clients may be attempting to update the same entry. This is dealt with in Dynamo with *Vector Clocks* [16][40][41], a mechanism for using data revision numbers and revision predecessors to maintain consistency. This is the same process used in revision control systems that provide revision history and branching. In Dynamo, each entry has an associated list of node-revision pairs. Two instances of the same entry can be reconciled by comparing their revision lists. If instances have overlapping sets of revisions then conflict resolution must be performed. Otherwise, the instance with the most recent revision number is saved while the other is discarded (or overwritten). The same process can also be applied when there are 3 or more instances of an entry. This approach makes Dynamo tolerant to Byzantine Faults [11], as discussed in section 2.3.

Dynamo's design follows the principles laid down in Karger [35][36] and Stoica [70] for consistent hashing. Partitions of equal size are defined on the key continuum and assigned among available nodes so each node is responsible for many separate intervals on the continuum. As Dynamo is a proprietary system the authors often do not provide specific implementation details, so a granular algorithmic evaluation cannot be presented. The work of the authors provides a thorough overview of the design considerations for such systems intended to operate in a large-scale, distributed environment.

2.8 Hashing

2.8.1 Overview

A fundamental component of this type of caching system is the hash table. A well implemented hash table will guarantee `O(1)` inserts and deletes [14][24][31] can provide consistent memory allocation response times over long periods [8][44][75] and provide flexible look-up methods if effective hashing functions are used [31][38]. This section will present the background theory upon which these concepts rely, to better inform the reader of the performance concerns involved with DHT's.

2.8.2 Selecting a Hash Function

We will not provide an overview of the behaviour of specific hash functions, factors for evaluating some of these functions will be presented. Karger et al outline a set of 4 useful criteria for evaluating the suitability of a hash function for storing a particular set of data in a hash table [35]. They are:

- Balance
- Monotonicity
- Spread
- Load

Balance is described as the property of entries being consistently distributed among the set of allocated hash table 'buckets' [35][36]. In other words, in a balanced hash function, no one bucket is more likely to receive an entry than any other, thereby reducing collisions. The property of monotonicity implies that the relative ordering or positioning of entries does not change as new entries are added to the structure [35]. This property is also pertinent to the task of mapping entries to a specific server if the cache is distributed. Spread is the level of agreement on the key of an object, between successive hash operations [35]. That is to say that a function has low spread if an object is consistently given the same key. Last, the load property defines how many entries are allowed per 'bucket' [35]. Note that this is separate from the concept of *load factor*, which describes the proportion of populated buckets to empty ones, or how 'full' the hash table is. This is a useful consideration however, as a hash function without a large enough domain will tend to have a high load factor (near or above 1), resulting in many collisions since there are not enough buckets.

Aside from the previous considerations, there are a number of other function properties that can be be useful in specific situations. For instance, a function with the property of grouping similar entries together with sequential keys would be helpful for search problems. There is also a need for hash functions that are able to return results on fuzzy searches [56], such as for multimedia [86] or text searches [66]. There are also hashing schemes that can adapt to perform better (in terms of collisions, *etc.*) to specific data sets [21].

2.8.3 Perfect Hashing

Several common problems in computing where hashing is applied is to determine if an element exists in a data set, and if so, where the element resides in memory. This problem is encountered in various networking applications such as routing tables [49], packet filtering [49] and name resolution. Since it is undesirable to add extra latency to perform these tasks [49], it is important to limit the worst-case time behaviour for which hash tables are known [31]. A class of hash functions that can be better suited in these cases are perfect hash functions (PHF's). A perfect hash function creates a collision free mapping between the domain and $\mathbb{N}_0$ [14][19][27][30][49] (natural numbers including 0). By this definition, any countable set may be the co-domain, for example the set of all Universally Unique Resource Identifiers or UUID's [42]. With even better performance is the minimal perfect hash function (MPHF), in which the domain of n keys is a bijection with $\{0,\ldots,n-1\}$. Since collisions are avoided in both classes, no additional steps are needed to avoid hash table collisions, so accesses can be made in worst-case `O(1)` time [19][49].

Another distinction between these functions is their utility in the static vs. dynamic case. When all of the contents of the domain are known *a priori* or following a precisely known distribution, creating a function with the desired characteristics is a more trivial task. However, in the dynamic case, the frequency and cost of performing rebuilds should be considered [19][49]. Overall, perfect hash functions are an ideal algorithm when repeated searches must be performed on a predictable or predefined set of data.

2.8.4 Defining The Identity of Hash Table Entries

In retrieving entries, a primary concern is that the necessary information for retrieving entries is available. Again, consider the example of looking up information about a person by address and birth date. In this case, the application front end is unlikely to be able to derive the key from the available information, in fact, the name contained in the key may even be the piece of information needed for the look-up.

There are several ways to approach this problem, each with some drawbacks. The first is to create multiple cache entries, using separate keys for each, catering to the available information by which the entry is expected to be looked up. This approach should provide for a good hit rate, but this comes at the expense of data duplication. To prevent this duplication, the cache may be restructured so that is it possible to have multiple keys to the same entry. A third approach to this would be to break up the entry into in smaller parts, again, each with separate keys. This approach may be successful in cases where an entire object isn't required, however if different keys are used, retrieving the entire object from the cache will not be feasible.

For the caching process to work well, it must be determined how items should be associated with a key. To do this, an algorithm must be chosen that defines the identity of the data to be cached, such that any two data objects are unlikely to have the same identity (unless these two objects are indeed considered to be equivalent). This process is analogous to selecting primary keys in database design [47][73]. Unlike the integer keys mainly used for database tables, it is important to be able to calculate a key without having access to the full entry itself. The integer sequences often used for primary keys in databases are an example of this problem, since this type of key is not derived from the data in each row. In this case, you cannot perform a direct look-up on that cache entry unless you know the integer key ahead of time. Another complication a look up may be defined by multiple properties of an entry [31][39], so providing the means for objects to be defined and referenced in multiple respects is a useful goal.

To reiterate, the goal in defining the identity of the object is to select an appropriate input for the cache to generate a hash key for. There are a few considerations that are helpful to take into account for this process. First, for an object a subset of the properties of the object can be used to define the identity of the entry. As stated previously, this should be done in such a way that the values of each property are unlikely to overlap those of a separate object. These set of properties will be referred to as the object identity properties. A complication arises when the properties of the entries are themselves objects, but as long as each object down the chain has defined object identity properties, the identity of the parent object can be determined. However, determining the object identity properties for many sub-objects will add to the computational time needed to generate an overall hash key, so reliance on sub-objects for identity should be minimized. In general, what differentiates two objects from each other will be dependent on the design decisions made, and the requirements of the situation [31][39].

Consider the earlier example of a record defined by a first and last name. Such entries are often identified by a record id such as a customer or employee number. Although hash tables have many useful benefits, they do not provide as well for loose searches, so you must be able to produce the key in advance to locate the entry you want. Using the defined scheme guarantees that an entry, with a key created using 'Jane Smith' as an input, can be looked up directly in `O(1)` [27][31][35]. But if other information must be used for the look-up, such as 'born on June 2nd, 1978' and 'lives at 156 Anystreet, Anytown Ontario', then you will need to get access to the entry by some other means. To compensate, a greater possibility of key collisions must be allowed for by creating keys in a way that differentiates the majority of cache entries but allows for two entries to have the same key, this need not be a problem. In the preceding example, it is possible for two people at the same address to have the same birthday (*e.g.* twins). However, the probability of this is far less likely because considering more criteria should result in fewer collisions. Also, if there is a collision, as long as it is possible to select the appropriate entry using the full contents of all entries matching that key, then the collision can be resolved. If not, the properties used to define the identity of the entry may need to be changed. In the cases of storing records by name, collisions can be more common, but this need not be a problem for other situations. Many types of data such as home addresses are structured so that entries with the same key aren't allowed, so if restrictions can be put in place, there will be a hash function that maps the data without collision.

A potential solution is to store the entry so that it is reference by multiple keys, depending on the search needs of the application. However this approach has several pitfalls. Referencing entries comprising many individual results provides a coarse level of access [18] and a secondary search of the data will be required if only a subset is needed. The opposite case is to request many individual entries stored under distinct keys. In order to be effective, the hashing system must provide an efficient means of requesting multiple hash keys, as in [23]. In either case it is important to be wary of IO performance because requesting large or numerous entries at once may saturate the communication channel (such as a memory bus or Ethernet link). One approach to this problem is to 'chunk' data in related blocks [18]. Where there is high overlap between a set of data and a subset of available chunks, an acceptable trade off in granularity is reached.

2.8.5 Entry Serialization

Once a strategy for hashing and entry identity has been chosen, the next step is to consider the storage representation of entries. The process for storing atomic data types is well-understood but composite types can be problematic. For example, language specific structures and sets such as arrays, dictionaries and tuples require an agnostic representation. Common representations in use are XML [84], JSON [15] and YAML [81]. Several language specific representations with differing degrees of agnosticism are Python Object Serialization/Pickling [82], the PHP serialize format [85] and Java Object Serialization [83]. There are pros and cons to be considered for each of these approaches, such as the required storage space and human readability. Trade offs between using a text or binary representation should also be considered. In any case, using a predefined and well-formed representation is preferred as this removes ambiguity and portability concerns such as endianness, character encoding, floating point accuracy and others.

A primary concern with serialization is that all objects stored in the cache can be restored to an equivalent state when retrieved. There are some operational complications with this, such as whether or not the retrieved object would be considered 'equal' to the object instance from which the entry was derived. Care must be taken to ensure that the de-serialized entry meets appropriate requirement for equality and identity. If not, data consistency problems are likely to be encountered in practice.

2.9 Balls & Bins

Balls and bins describes a discrete modelling problem where n balls are successively placed within one of n bins. Bins may be selected uniformly, independently or neither. The dynamic case with an unknown stream of balls arriving is ideal for modelling load balancing scenarios, with balls as work units and bins as computing resources. However, the static case is covered most heavily in the literature.

Of particular concern in this area is the load carried by each bin. The birthday problem [14][65] states that the expected size of a group for two people to have the same birthday is far fewer than the number of days in the year (28, given the naive assumption that birthdays are uniformly distributed [14]). Following from this result, there is a likelihood that placing balls into bins at random will result in multiple balls sharing a bin, even when there are many more bins than balls. However, there are expected limits on the load on each bin, when a single bin is selected at random. A classic result is that the maximum expected load when n balls are placed at random in one of n bins:

$$\theta\left(\frac{\ln n}{\ln\ln(n)}\right), n>0 \quad [37] \tag{3}$$

This upper bound holds with a probability of $1-n^{-\alpha}$, where $\alpha>0$ and $n\rightarrow\infty$ [3] [53] , which the authors refer to as 'with high probability' (*w.h.p*). Later work introduced the “Power of Two Choices” [3] technique in which a uniform and independent selection of at least two bins and placement the ball in the least loaded, reduces the upper bound to, *w.h.p.*:

$$\left(\frac{\ln(\ln(n))}{\ln(d)}\right)+O(1), d\in[2,\infty] \quad [3][76] \tag{4}$$

There is some benefit of $d>2$ but returns diminish quickly and storage and entry look-up overhead may be problematic.

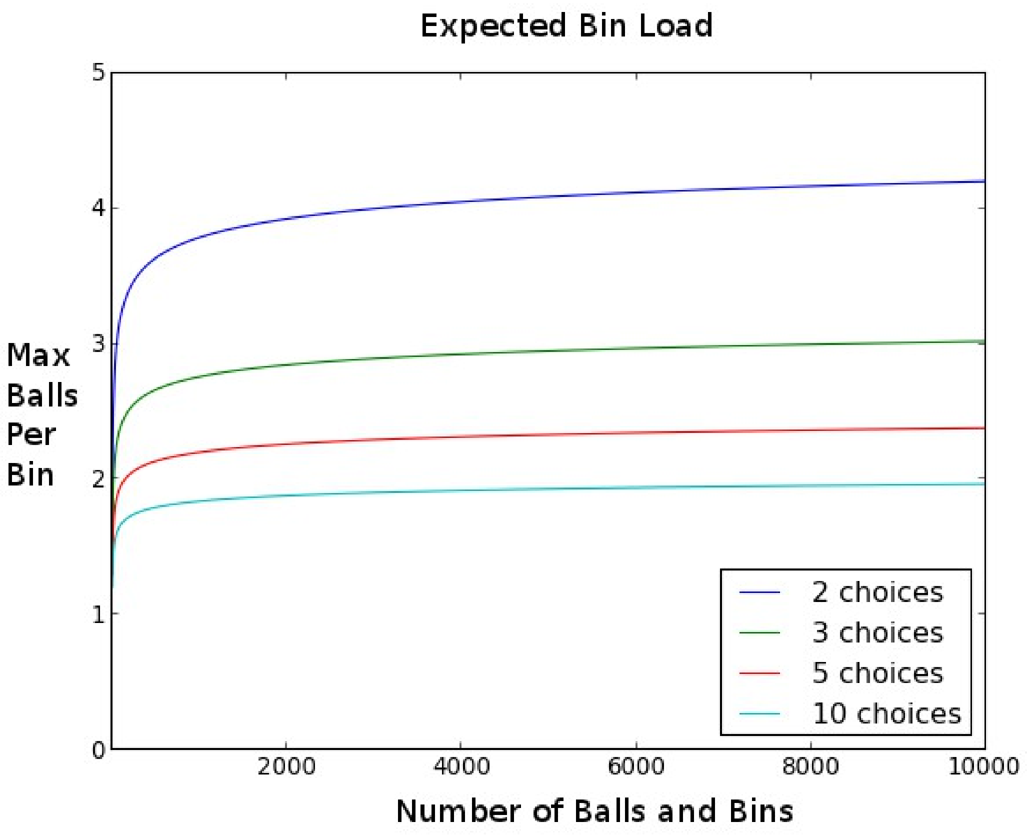

Figure 2. The upper bound on the number of balls per bin decreases with the number of bins that can be considered for each placement.

It turns out that the most crucial aspect of the bin selection process is that there are at least 2 bins to choose from. For Example, the “Always-Go-Left” [76] strategy enumerates the bins, divides them into roughly equal groups of size:

$$\theta\left(\frac{n}{d}\right) \tag{5}$$

and selects a bin from each [76]. The final selection is decided first by choosing the least loaded, as with [3], and breaks ties by selecting the left-most bin, which has the lower index [76]. This strategy further improves the maximum expected load to (where d is the number of choices):

$$\frac{\ln(\ln(n))}{d\ln(O(2))}+O(1), w.h.p. \text{ [76]} \tag{6}$$

The results from the “balls and bins” research is important for the areas of data arrangement. In particular, these results define the maximum expected load on a hash table 'bin'. This information is useful when tuning the distribution of a hash function and for providing adequate backing storage for a hash bucket (whether in memory or non-volatile storage).

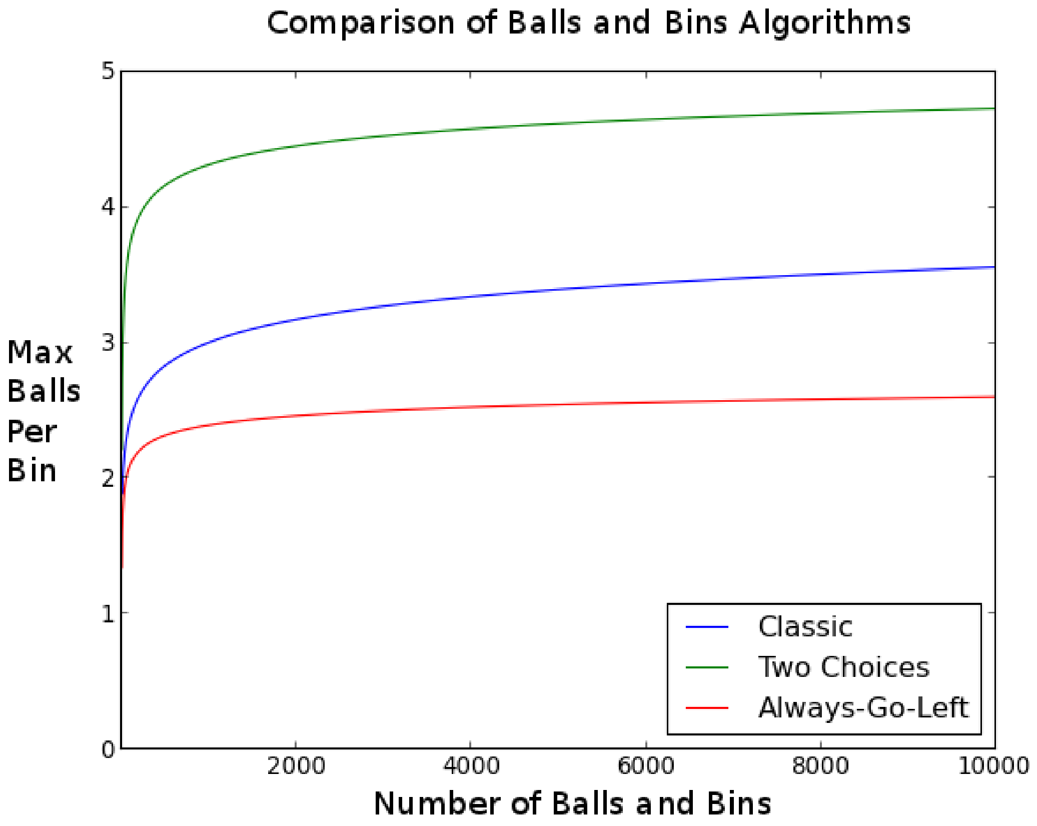

Figure 3. Comparison of the maximum expected loads of balls and bins algorithms.

In addition to the preceding static cases, a thorough overview of the “Two Choices” approach, analysis of the dynamic case and practical concerns can be found in Mitzenmacher [53].

It is important to note that each ball or entry may have multiple properties contributing to load. Extending the ball metaphor, balls may have different size, weight, colour and material. 'Balls' could even come in the form of different geometric objects like cubes, pyramids, tori or even an irregular object. These variables can be considered analogous to those that incur computational overhead such as processor time or memory. For bins, volume capacity, weight capacity and dimensions could be considered. For computing resources, CPU load, storage load and locality of reference (i.e. how 'close' data is to the processor). Considering these variables takes us into the domain of more difficult problems like the knapsack or bin packing problem. There are no known polynomial time solutions for determining optimal object arrangement, so this should not be a goal for this type of load balancing problem.

Chapter 3: Methodology

Distributed hash tables (DHT's) are a useful structure for maintaining storage performance in high traffic, high availability client server systems. Not only are these systems useful for providing cache or other storage layers but also as for distributing other types of remote service calls over many computing resources. One of the most popular methods of managing the distribution of cache entries between servers is consistent hashing [35][36][43].

The consistent hashing algorithm [35][36][43] provides a technique for maintaining monotonicity in a distributed cache when cache servers need to be added or removed. This technique was devised to provide a caching mechanism that could effectively span wide area networks (WANs), where it can be uncertain which servers and routes will be available. This approach can easily be adapted for more stable local area networks but also provide fault tolerance. Consistent hashing works by dividing the overall key domain for the cache entries into small chunks that can be spread amongst the servers. This is done by selecting sets of random, non-sequential integers from 0 to 2^{32}, and distributing each set among the cache servers. When a new entry is to be inserted into the cache, the key will be hashed to an integer within the range and will be placed on the server to which the next lowest integer has been assigned. This technique helps cope with server removal and addition for several reasons. First, since the distribution of the key domain is no longer reliant on the number of servers, far fewer cache entries will need to be moved between servers [33][35][36][43]. In comparison, it is likely that every entry would need to change servers in the alternative scenario [4]. Second, if there are problems with the weighting of the key domain distribution, the division of consecutive areas of key domain will help spread entries evenly between servers.

While consistent hashing is theoretically well-founded and in use, the goals of this work are to determine if constant width intervals are practical for providing fast node resolution, roughly uniform key distribution and a restructuring system that maintains the fault-tolerant behaviour of other implementations.

3.1 System Overview

F^{2}DR can be reduced down to few basic components, a set of intervals on a continuum, a set of nodes, a mapping between the previous, and a set of clients. The continuum in this work is a circular number line of discrete integers; a number greater than the range can be placed on the number line by wrapping back around to the beginning (see 5). The intervals are defined by a set of evenly spaced points on the interval and the nodes are proper sub-sets of all intervals on the continuum. Recall that intervals differ from the usual definition of hash table 'buckets', in that intervals will tend to contain multiple values over the range. For convenience, intervals and nodes are referred to by their start point on the continuum, beginning at zero.

F^{2}DR implements a DHT system with a few basic components, a set of intervals on a continuum, a set of nodes, a mapping between intervals and nodes and a set of clients. The system clients use this mapping to resolve a key (falling on the continuum) to a node. Each node provides a set of services, such as *CRUD* (Create, Read, Update & Delete) operations, for each client in the system to use. Key to node resolution operates through a hash function based on the identical base-two multiple spacing of intervals. This function evaluates to the index in to the list of the interval-node mapping (which is in ascending order from the interval at zero). In this way, the DHT operation of storing a key-value pair can be implemented by resolving the key to a node and invoking the 'C' or create operation.

At the simplest level, the system clients use the mapping to resolve a value or 'key' on the continuum to one of the nodes. This mapping uses a hash function that takes advantage of the spacing of intervals to produce a function that is fast to evaluate, and is impacted minimally as nodes join and depart. Together, these behaviours provide a system that is both scalable and fast.

To set up F^{2}DR, each interval is assigned to a node with each node receiving a roughly equal number of intervals. The intervals are kept reasonably small, and assigned in a sequence such that each node is represented by a broad range of the entire continuum. This is to help prevent too many requests from being directed to a single node, in cases where keys fall in clusters on the continuum, rather than a more balanced distribution. This by no means prevents imbalance, but users will always have a clear picture of how data will be distributed in an F^{2}DR system. If intervals were distributed at random, then for a given key domain, the balance could vary wildly from one set of assignments to another.

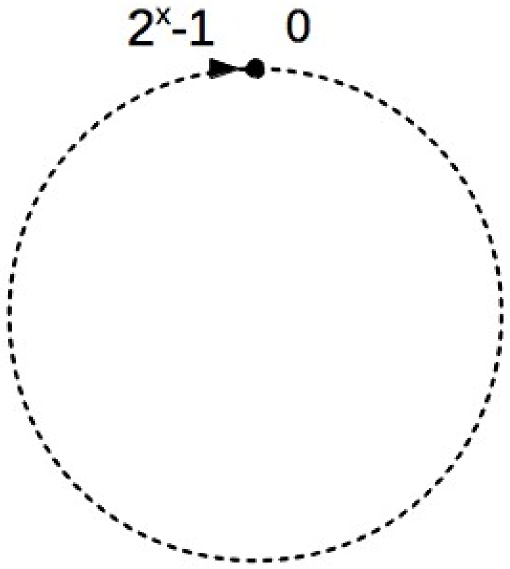

Figure 4. The continuum, from 0 to 2x - 1.

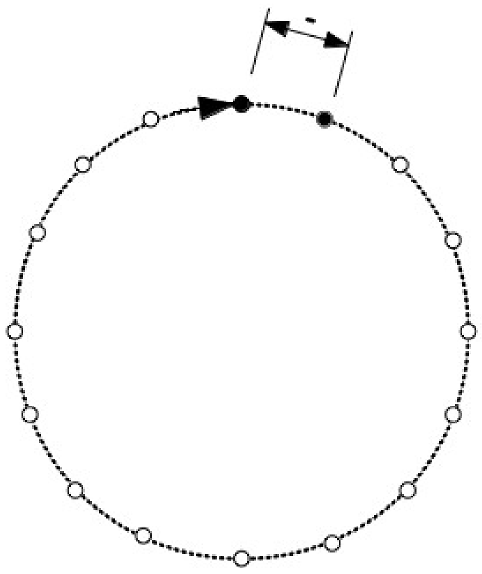
Figure 5. An interval on the continuum.

As previously stated, keys are resolved to a node using a hash function that relies on specific properties of the interval set. Each interval in F^{2}DR is defined by an arithmetic sequence, beginning at zero and each falling on a multiple of a power of two. This divides the continuum into pieces of equal magnitude, starting at points that are derived with a simple calculation. Using a list of assignments, sorted in ascending order by interval, the node can be determined by locating the index in the assignment mapping using the key and gap. The preceding two calculations along with the speed of base-2 arithmetic make it possible for F^{2}DR to meet our performance goals.

The other facet of the system is dealing with change, when nodes join and depart. Our algorithm differs from the standard consistent hashing algorithm [35][36] in order to account for the fixed distribution of intervals. To maintain this distribution while creating new intervals, each existing interval must be split in half, doubling the total number of intervals and invalidating roughly half of the entries. To avoid this disruptive operation, existing intervals are reallocated to the new node or from a departing node. When nodes join, a roughly equal number of intervals are reassigned from each original node to the new node. For example, in Figure 4. there is a possible Interval-Node mapping before and after a join. Intervals 1, 5, 10 & 16 are reassigned to the new Node, 4.

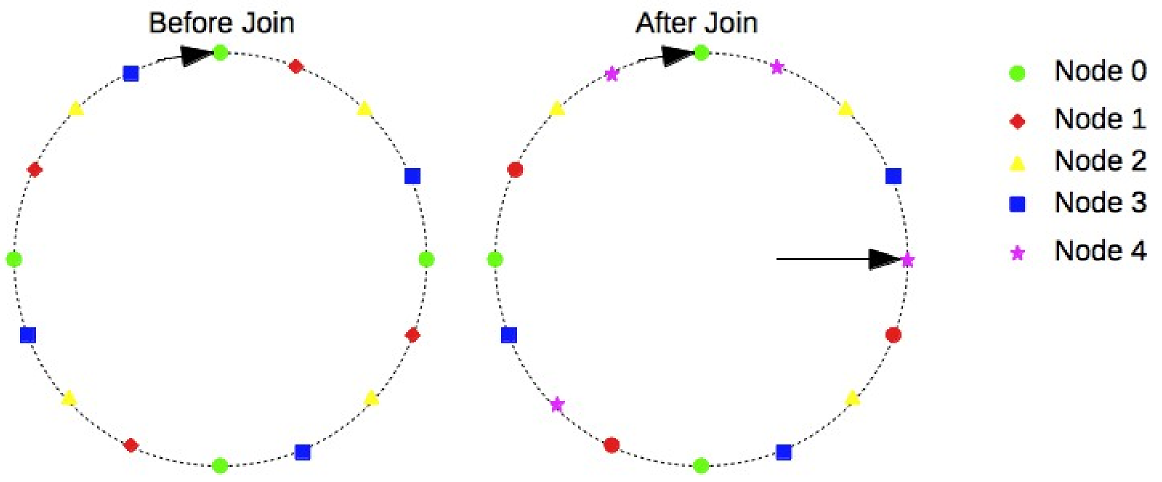

Figure 6. The Interval-Node mapping before and after a node join.

A scenario where F^{2}DR will perform sub-optimally is when many clients have inconsistent views of the available nodes. This can result in identical entries being stored on separate nodes, and look-up failures when the client is unaware of the node that contains that entry. In order to reduce the occurrence of these situations, it must be ensured that join and departure operations occur in order on each client and with minimal delay. To ensure system integrity, some operations will need to be blocked, but resolution can still take place if a conflict resolution system is in place [16]. This can be accomplished with an approach like TCP's (A Protocol for Packet Network Interconnection) for enforcing packet delivery order. In combination with a prompt means of communicating changes to all clients, the duration that the system spends in an inefficient state can be reduced.

Only when the system has grown to contain enough nodes that relatively few intervals can be assigned to each node, should new intervals be allocated on the continuum. When this is necessary, the new intervals can be introduced over time, reducing the proportion of keys that are invalidated.

3.2 Operating Environment

There are several factors in the operating environment that impact the performance of systems like F^{2}DR. For that reason, assumptions are made about these factors in order to clarify the design decisions in the work. First, communication between clients and node will perform best on a low latency-high bandwidth network. Latency is especially important because the look-up and synchronization algorithms in the work are easily affected by response time. In a busy network or one with too much distance between components, operations taking tens of milliseconds or less are easily overtaken by latency. This also becomes a problem as the number of nodes increases and network capacity may need to be shared between client requests and system communication. However, F^{2}DR does not include mechanisms to reduce the cost of network routing. In the interest of reliability, it is helpful for components to be interchangeable. Attempting to optimize algorithms in the system for communication performance would add considerable complexity and these optimizations may not continue in effectiveness as the system changes. For example routing information may need to be recalculated and synchronized if a node joins or departs.

3.3 Components

At the highest level, F^{2}DR is a system providing *clients* the ability to route *entries*, identified by *keys* from a *continuum*, to *nodes*. The main use case for the system is a distributed hash table where the entries comprise data that must be accessed quickly [17][24][35][43]. A means to achieve this speed is by distributing load among many nodes, allowing each node to respond faster. This strategy is also commonly used for load balancing service requests, such as for web servers, databases, among others [16] [17][18][23].

3.3.1 The Continuum

The continuum can be visualized as a circle, numbered with integers, with range `[0, 2^x - 1]`, inclusive:

```
def continuum(continuum_power):
    return range(0, 2 ** continuum_power - 1)
```

This differs from the original definition given by Karger et al. [35][36], which uses the unit circle. However the ranges in [16][23][64] are more appropriate for dealing with hash values, which tend to fall within the preceding range. There is also the benefit of working with faster integer arithmetic, rather than floating point.

3.3.2 Intervals

The unit for dividing responsibility among nodes and the interval, which divide the continuum into distinct segments. The intervals have the properties that they are equal in length, contiguous, non-overlapping and represent a proper subset of the entire continuum. That is to say that all possible keys fall within one of the intervals.

In the work, intervals are referred to in terms of their start value (inclusive), end value (non-inclusive) and range, which maintains the equality of *`(start + range) == end`*. The interval magnitude is defined by an integer power of two, *gap_power* where `0 =< gap_power =< continuum_power`.

```
def intervals(gap_power):
    gap = 2 ** gap_power
    # for each int in the continuum, stepping by gap
    return [(x, x + gap - 1) for x in continuum(10)[::gap]]
```

So given `continuum(10)`, `intervals(6)` gives us interval tuples of:

```
[
     (0, 63), (64, 127), (128, 191), (192, 255), (256, 319), (320, 383),
     (384, 447), (448, 511), (512, 575), (576, 639), (640, 703), (704, 767),
     (768, 831), (832, 895), (896, 959), (960, 1023)
]
```

The result of this function is that continuum is divided into equal intervals of magnitude *gap*. It is also important that this list remain sorted by interval, as this property is relied on by the resolution function.

3.3.3 Keys and Entries

Keys are integral values on the continuum that can be used to identify *Entries*. Entries may be of any type that can be uniquely identified, and can be serialized to a format suitable for storage. Immutable entries are preferable, but logic for handling expiration and consistency for mutable data can be handled by the client. However this may incur a non-trivial increase in complexity and overhead.

3.3.4 Nodes

Each *node* in F^{2}DR is a container, with a unique identifier, that services requests, by entry key. These requests may refer to CRUD operations backed by an appropriate hash table implementation, or some other form of remote procedure call (RPC). All responsibility for interval mapping and system restructuring falls outside of the nodes, and the nodes need not have any knowledge of how these operations are performed.

An important property of the nodes as a group, is the load placed on each. Load is a complex concept, with many situation dependent definitions; but in F^{2}DR this is considered to be the extent that the most scarce resources are utilized. It is left to the reader to provide an evaluation of resource scarcity.

Over top of the nodes, there must exist a management interface that provides the ability to perform restructuring operations, such as joins and departures. The mechanism used for determining how to make interval-node assignments is the *node assignment queue* (or NAQ), which is implemented a min-max priority queue [32]. This type of priority queue is implemented with the 'heap' data structure, a type of tree where nodes have consistent ordering throughout. Heaps are useful for this purpose because they allow for fast assess to the 'top' item (the 'pop' operation). Inserting items is somewhat slower because the heap must be restructured to maintain the ordering (the 'heapify' operation).

Nodes to be assigned additional intervals are popped from one end, and nodes to have intervals reassigned, from the other. This allows us to manage load through the application of a priority function to calculate a value based on the node's perceived load. Finally, there are two low level operations associated with nodes, used to add and remove interval assignments.

3.3.5 Clients

The *clients* provide means for outside access to an F^{2}DR system for other software or users. They will have access to all of the information required to route keys to nodes, and can also perform restructuring operations when nodes join or depart, or if intervals must be reassigned in order to balance load.

Clients are entities that have possession of the interval-node mapping, which can be modified in order to restructure the system. If developed using a shared-nothing architecture [71], the clients can be synchronized when the system is initialized, and the clients need not communicate with each other to maintain a consistent state. These decisions could be performed in a separate component of the system, but the communication latency needed to manage this elsewhere is expensive.

3.4 System Operations

3.4.1 Keys and Resolution

Given the proposed uses for F^{2}DR, requests must be serviced at a high rate, perhaps thousands per second. Requests may be to perform a standard CRUD operation or invoke a remote procedure. In this situation, a small difference in run-time complexity can quickly add up over time. In other reviewed consistent hashing systems, resolution is performed using a binary search for the key over the list of intervals [24][70]. However, systems that use partitions of equal magnitude can better handle this operation using an interpolation algorithm [43]. In systems where the intervals are distributed on the continuum at random, there is the possibility of pathological cases. In this work, pathological cases are considered to be instances of a hashing system with an unusually high number of collisions. For example, if all defined keys happened to fall within intervals assigned to a single node. This would result in one node doing all of the work; the system would be unbalanced. However, F^2DR is agnostic to the hash algorithm used, as long as the algorithm produces values that can be mapped onto the continuum. This allows implementors to select a specific hash function that produces uniform values in the range, potentially avoiding problematic cases.

It is critical to the effectiveness of F^2DR that keys are resolved to a node quickly. In the considered use cases, such as a distributed data retrieval system, the system will be required to service thousands or tens of thousands of requests per second. A small increase or decrease in performance will result in a measurable change over a short period of time.

In the other systems reviewed in Chapter 2, resolution is performed using a modular arithmetic hash function [23], a binary search for a suitable node [33][35][43] or a constant time hash [33][43]. The modular technique meets the speed requirement, however nodes cannot join or depart the system without a large disruption to the interval assignments. The standard consistent hashing approach for handling this problem uses binary search to resolve nodes, which is an order slower. The algorithm proposed in this work strikes a balance between the previous techniques in that nodes can be resolved quickly, while preserving the properties of consistent hashing and providing a mechanism for evening load distribution between nodes.

This algorithm uses a perfect minimal hash function [30][49] which takes the key as input and returns an index to the interval-node assignment list (sorted in ascending order by interval). As described in the background chapter, a perfect minimal hash function is one that produces no collisions and requires no more buckets than keys. Our function works by performing truncation division on the key, by the gap, which gives the interval index in the interval-node assignment mapping:

```
def resolve(key, mapping, gap_power):
    return mapping[key / 2^gap_power]
```

Also, since the intervals fall on power of two multiples, resolution can be performed by shifting left by `log`$_2$`(gap)` bits, using the result as the index to the interval-node mapping array.

For example, with a gap of *128* (2^7), the interval set would be: *{0, 128, 256, 384, 512, 640, 768, 896, 1024, ..}*. Using this set, the following nodes would be returned by the *resolve()* function given the keys *(13000, 128, 127 5087)*:

```
Key     Interval    Index
13000   12928       101
128     128         1
127     0           0
5087    4992        39
```

In General: Given a gap of 2^x and any key `k` within the range, the interval and index are defined by (7) and (8) respectively:

$$\begin{gathered} mask = 2^{continuum\_power} - 2^{gap\_power} \\ interval = k \wedge mask \end{gathered} \tag{7}$$

$$index = k \gg gap \tag{8}$$

As there are practical limits on the minimum magnitude of the continuum, collisions will become increasingly likely due to the pigeonhole/Dirichlet's box principle [78] and the *birthday problem* [65]. Stated briefly, the pigeonhole principle tells us that if n containers and *n + 1* objects must be stored, then at least one container must contain multiple objects. Similarly, the birthday problem states that in roughly 25 randomly selected people, there is better than 50% probability that two people were born on the same date in the year. A simple way to deal with collisions is to use a UUID [42] or other large range hash function [42] to provide unique identification for the entry. This identifier can be stored with the entry in order to break ties where entries have the same key. The identifiers can be hashed onto our continuum and when a collision occurs, the identifier used to generate the key for the look-up can be compared with the identifiers stored with the entries. As long as collisions are rare, this method can deal with collisions reasonably well using chaining. More effective means of collision resolution, such as probing would be problematic due to the cost of querying multiple nodes.

3.4.2 System Initialization

To construct an instance of the system, a continuum must be chosen, then a set of intervals and finally an interval-node mapping. Care must be taken with these selections because the mapping and interval magnitude have a strong influence on the future performance of the system. Also, there is a non-trivial overhead for performing later restructuring operations to change these parameters.

To define an interval-node mapping, let there be a set of nodes $\{node_0, node_1, .., node_{n-1} \mid n \in Z+, n > 0\}$. Assume that each interval in the system is initially expected to contain roughly the same number of entries and subjected to the same volume of requests. In this case it is reasonable to assign a roughly equal number intervals to each node. As load statistics are collected during the operation of the system, this scheme can be adjusted so that intervals are distributed in a more desirable manor.

The intervals and initial mapping can be created simultaneously, adding only a constant factor to the time complexity. To do this, first set a count for each node, that represents the number of intervals that can be assigned. Then for each interval, select a node from the list and decrement that node's count. Once a node's count reaches the maximum, it is no longer considered for selection. The selection process continues until no more intervals remain to be assigned.

A roughly equal number of intervals should be assigned to each node. However, there are cases where the number of intervals will not be divisible by the number of nodes, so some nodes would be responsible for a greater proportion of the continuum. Some strategies to overcome this problem are:

- increase the ratio of intervals to nodes

- maintain a number of nodes that divides the intervals evenly

Both of these strategies can be used when initializing the system. If the set of intervals is much larger than the set of nodes, each interval will account for a small proportion of the continuum. As a result, a small difference between the number of intervals assigned to each nodes will account for a small proportion of the continuum. During initialization, the second point can be implemented by setting the magnitudes of the interval set to be divisible by that of the node set.

3.4.3 Node Assignment Queue

After system initialization, the interval-node mapping of the system is clean, and nodes should be responsible for a diverse proportion of the continuum. However, as intervals are reassigned during joins or departures the tidy distribution of intervals is unlikely to be maintained. The mechanism for adapting to these system changes is the node assignment queue (NAQ), which will determines how intervals will be distributed during a node join or depart.

In some configurations, the number intervals cannot be divided evenly among the set of nodes. Some nodes will become responsible more intervals and thus a larger proportion of the continuum while other nodes will be responsible for too little. A general purpose algorithm is proposed here to handle this issue. However, the expected distribution and characteristics of the keys distribution will be best known by the implementer so this algorithm is not expected to be comprehensive.

The NAQ will possess the following interface:

- `insert(NAQ, node)`
- `popFromBottom(NAQ)`
- `popFromTop(NAQ)`
- `remove(NAQ, node)`
- `priority(node)`

The insert, pop and remove functions are self explanatory, and can use a standard implementation [5][32]. Things get more interesting when defining a heuristic for the implementation of the priority function. In this case, a naive implementation would be assign priority based on the number of intervals assigned to a node, where nodes with the most intervals would reside at the top, and the least at the bottom. This way when intervals are to be assigned to a new node, nodes with the most intervals are popped first, and vice versa. However, this strategy is only effective when keys are distributed uniformly and stored entries incur uniform load. Priority algorithms for diverse data will be more effective by taking several load variables into account (such as interval popularity, node occupancy, node response time, *etc.*).

As an example of this approach, consider a function that evaluates nodes to an element on the set of integers `{.., -2, -1, 0, +1, +2, ..}` where each element represents a load value. Nodes with a greater value would reside at the top of the queue and would be considered to have the most load, and those with lesser value at the bottom. In effect, nodes at the top of the queue have 'intervals to spare', which can be reassigned to nodes with less load. This type of behaviour can be implemented with a min-max priority queue, which incurs only a small penalty in run-time complexity over a standard heap-based queue [5][32].

A secondary use for the NAQ, is for gathering information on the health of the system by analyzing the load value for each node. With this information in hand, informed decisions can be made about whether restructuring is required, by reassigning intervals or changing the number of available nodes. In an ideal scenario where entries and access are uniform and each node is assigned an equal number of intervals, a flat distribution of load values about zero is expected. However, in real world cases, the best that can be expected to be achieve is a set of loads resembling a normal distribution, centred tightly near zero.

This information can be used to determine the overall system load by checking for a positive or negative skew to the values. The existence of a strong negative skew over an extended duration likely indicates that there are insufficient nodes to handle the load. In this case, additional nodes should be joined to the system. Conversely, a positive skew indicates that there are too many nodes, so the system could operate as effectively with fewer nodes. If nodes are found to have too little activity, then a controlled departure of node(s) can be performed. Load will then shift from the departed nodes to others in the system.

Another indicator of system is how well the interval-node mapping accounts for variance in the load over the continuum. This variance may be caused by differences in the number of keys falling in each interval, or computational overhead from a specific keys. For example a key to an entry that requires a lot of space to store, or a time-consuming calculation to be performed. By monitoring the requests against each node, detection of outlier load values could be accomplished. This information about load could then be used to redistribute load more evenly across the system nodes.

There a number of considerations to take into account when choosing an algorithm for the 'selectFromNode' function, including:

- Will the assignment result in consecutive intervals assigned to the same node?
- Is each node assigned roughly the same number of entries?
- Are a roughly equal number of look-ups resolved to each node?
- Distribution of interval-node mappings
- How much of each node's storage capacity is used?

The use scenario dictates which considerations are most important. In non-trivial cases, it is unlikely that a select function can be found that can be proved to be globally optimal though. However, there are several situations to be on guard for. Is there a possibility that the key distribution over the continuum will match that of one or a few of the nodes, which would skew the load? If the node ordering is not sufficiently random, the departure of a node could cause many of the keys assigned to that node to fall back to the same node (if two nodes are often paired on the continuum in sequence). This may double the load on one node and leave the load of other nodes unaffected. Also, over time, clusters of closely packed intervals may come to be assigned to the same node. If the interval-node mapping is not diverse then a non-uniform distribution of keys is likely to undermine the fault tolerance of the system. A complete framework for handling these situations is not presented in this work, but it should be possible to detect this cases by analyzing the load values in the NAQ.

With functions in hand to manage the NAQ, a higher level interface is defined for handling interval assignments:

```
def assign(NAQ, interval):
    n_min = popFromBottom(NAQ)
    assignToNode(n_min, interval)
    insert(NAQ, n_min)

def reassign(NAQ, node):
    n_max = popFromTop(NAQ)
    interval = selectFromNode(n_max)
    insert(NAQ, n_max)
    //Assignment operation from the node's interface
    reassignToNode(node, interval)
    insert(NAQ, node)
```

There are several other node assignment considerations that have not been outlined. There is a need to balance the cost of updating load values, heapify and system performance. To maintain absolute consistency for load statistics, restructuring operations will need to be blocked while statistics are gathered. However this could be foregone if the desire for restructuring performance is deemed to be more important than stronger assurance on statistic accuracy. This compromise could also be applied to heapifying the NAQ, since this process could be halted in order to perform further restructuring, with an increased possibility that load variance between entries becomes greater.

3.4.4 Node Join & Departure

The preceding operations and the NAQ come together to provide convenient means of updating the interval-node mapping. When a node joins, intervals are *`assign()`*'ed until the new node has a priority no greater than that of the bottom node, then the new node is *`insert()`*'ed. For departures, the departing node is removed from the queue, and each assigned interval is *`reassign()`*'ed to nodes from the front of the queue. Recall that the NAQ places the nodes able to receive additional load at the front of the queue. Once an additional interval has been assigned to the node, the priority value will change and should cause the node to move toward the end of the queue (via the heapify operation). In this way, the system uses joins and departures as an opportunities to improve the uniformity of system load.

```
def join(NAQ, node):
    while(bottomPriority(NAQ) > priority(node)):
        reassign(NAQ, node)

    insert(NAQ, node)

def depart(NAQ, node):
    remove(NAQ, node)

    for i in node.interval:
        assign(NAQ, select(i))
```

For each node, a few bookkeeping functions are required to maintain a record of which intervals are assigned. As the system is restructured, intervals will need to be added and removed from a node's accounting. It is also helpful to keep load statistics for each individual interval to better inform assignment decisions. For example, to allow the NAQ to avoid assigning too many 'busy' intervals to a single node. Finally, a function is required to select an interval from the node, to be reassigned to another node. As with the NAQ, this decision should be made based on the interval loads, which should be maintained at a level that is neither too high or low. It is not necessarily best to select the interval with the highest load. For instance if this load is incurred by a number of popular entries then it is likely that shifting this load will have a measurable performance impact from the clients until the entries have migrated to different nodes.

3.4.5 Fanning Out for Scale

As the system grows in size by the addition of nodes, the number of intervals assigned to each node will decrease. With two few intervals assigned, flexibility to balance load by reassigning intervals between nodes is lost. The fan operations are triggered when there are too few intervals to provide a balanced distribution of keys between the nodes. While this point is arbitrary, existing systems use between 100-200 [24][33]. Due to the performance of our resolution algorithm, it is feasible to use many more intervals, which should reduce the cases where the key distribution is skewed.

This comes at a relatively small memory cost to each client.

The FanOut operation creates more intervals, in addition to those that are already assigned. Since all of our intervals are equidistant with their neighbours and fall on multiples of powers of two, our new intervals will fall in the middle of each existing interval. The new intervals are then assigned to nodes using the standard assignment scheme.

Performing all the assignments at once could cause significant disruption since entries on half of the continuum can become vacated. Fortunately, if a set of intervals is chosen that is large enough to account for the expected system growth, then this operation will be used rarely, if at all. If it is necessary to FanOut while the system is under load, this disruption could be reduced by staging interval assignments over a longer duration.

Since the interval allocations are predictable, the difficulties with array insertion and growth could be side stepped by appending blocks to the end of the list. During a fan out, the number of intervals is increased by a factor of two, so it should be possible to add a separate contiguous block of memory to contain the new interval-node mappings. This means that the intervals will not be contiguously ordered in memory but a jump table could account for this. Set an initial value `g_prime` to that of the gap, all intervals `i % g_prime == 0` will fall in the first memory block.

To implement *`FanOut()`*, first each client must be notified to begin the procedure. Next, each client must create the new set of intervals but not integrate these with the existing intervals. Finally, each client will loop through the set of new intervals and reassign each according to the assignment scheme. The new interval assignments can be applied all at once or in batches. Applying changes in batches delays the eviction of entries to reduce key misses and large numbers of inserts.

This is a blocking operation, so joins and departures cannot occur until the fan completes. If non-blocking, it is necessary to halt the progress of the operation at the same point on each node in order for the interval-node mappings to remain consistent between nodes. If implemented in parallel with appropriate locking, this operation will not prevent the clients from responding from requests.

Chapter 4: Results

4.1 Overview

To quantify the effectiveness of our algorithms, a time and space complexity analysis of each F^{2}DR operation is presented. This approach is chosen over statistical analysis because there are extensive results published on the characteristics of consistent hashing [24][35][36][33][69][70] and load balancing, such as balls & bins [53] and others [76]. Many of the algorithms in this work are comparable to those presented in Chapter 2: Background. These results are drawn upon to illustrate the performance of the proposed system.

4.2 List of Symbols

For convenience, the following variables used in our algorithm are defined below.

Symbol	Definition
`c`	The magnitude of the continuum. This value can be any 2^x, $x >= 0$, although $x \in \{32, 64\}$ are most appropriate due to their corresponding processor architectures.
`nodes`	The set of member nodes in the system.
`||nodes|| or N`	The magnitude the nodes set
`intervals & gap`	`intervals` is the set of all intervals on the continuum, where the `gap` is the magnitude. `intervals = {i	0 <= i < c, gap	i}`
`||intervals|| or I`	The magnitude of the set of intervals is `c = ||intervals|| * gap` where `(||nodes|| * n) <= c` and `(||nodes|| * n)	c`
`n`	The number of intervals per node in the system is `||intervals|| / ||nodes||`

4.3 Data Structure Performance

Several data structures are needed by each client, the list of nodes and intervals, the interval-node mapping, as well as the node assignment queue (or NAQ). This section will describe the time and space complexity of the structures used in F^{2}DR.

4.3.1 Lists and Maps

Nodes themselves have a small number of properties, so their space requirement as defined as constant. The node list must contain a reference to each node in the system, and provide the ability to look up each by an identifier (preferably in constant time). This requires `O(||nodes||)` space.

The interval list is defined as a set containing an arithmetic sequence. This list can be generated as necessary and only requires `O(1)` space for storage since it can be defined by a simple arithmetic sequence. The interval-node mapping can be reduced to an enumeration of each interval, with an associated node, therefore `O(||interval||)` space is needed.

The proceeding space complexities can be simplified to `O(||nodes|| + ||intervals||)` by removing the constant space needed for the interval list. In well-formed examples of our system `||nodes|| <= ||intervals||`, since each node should be responsible for part of the continuum. In order for there to be a high probability that entries are distributed evenly between nodes, `||nodes|| << ||intervals||`. Using results from The Birthday Problem [65], it is possible to select values for the preceding variables that will result in acceptable entry distributions. In this case, the storage space complexity will be proportional to `||intervals||`, giving a combined space complexity of `O(||intervals||)` for the lists and mapping.

4.3.2 Node Assignment Queue (NAQ)

The Min-Max priority queue described in [5] is used as a performance baseline for F^{2}DR. This study includes the following time complexities, that will be used to illustrate the performance of the system's NAQ:

Operation	Time Complexity
Create Queue	$\Theta((7 * \|\|intervals\|\|) / 3)$
Insert Element	$\Theta(0.5 \log(\|\|intervals\|\| + 1)$
Delete Min	$\Theta(2.5 * \log(\|\|intervals\|\|))$
Delete Max	$\Theta(2.5 * \log(\|\|intervals\|\|))$

Table 4. Time complexities of the Min-Max priority queue

These time complexities reflect a scalar increase in run time over those of a standard tree based priority queue [5][32]. The scalars in the time-complexity functions have been included To be faithful to the source literature, as similar systems differ in time-complexity by small factors [32]. For simplicity, these scalars will not be included in the complexity analysis for the proposed system.

When a node departs the system, this operation is needed in order to remove the node from the NAQ. The implementations described in [5][32] do not offer a direct way to find a particular node in the queue, but there are several possibilities. As a worst case, the entire tree can be traversed in $O(||nodes||)$ time. Since the queue maintains a sorted list of the weight of each node, it may be possible improve this performance through searching for the node using an interpolation heuristic. Since each node stores a copy of the assigned nodes and each node is responsible for a roughly logarithmic proportion of nodes, a node can be removed from the NAQ in $\Theta(\log(||nodes||))$ on average, using a binary search.

Operation	Time Complexity
Remove (Avg. Case)	$\Theta(\log(\|\|nodes\|\|))$
Remove (Worst Case)	$O(\|\|nodes\|\|)$

Table 5. Additional Min-Max priority queue time complexities

4.4 System Operations

4.4.1 System Construction

In construction, the set of nodes must be iterated over, mapped to a node, and the NAQ must be updated. It is assumed that the assignment can be performed in constant time, such as by popping a node from a list, however a more complex algorithm and data structure may be used. If the creation of the NAQ is deferred until all intervals have been assigned the time complexity of `O(||intervals||) + Θ(log(||nodes||))`. In general, F^2DR will assign hundreds of intervals to each node to reduce the chance of imbalance, causing the linear term to be more likely to dominate.

4.4.2 Adding a Node

The intervals for new nodes must be reassigned from existing nodes in the system. However, the number of intervals to be assigned to the new node depends on the priority algorithm. This requires cycling through the existing nodes, reassigning an interval from each to the new node, until the priority of the new node is equal to that of the bottom node in the NAQ. The time complexity of needed to iterate the list and rebuild the NAQ is that of popping and inserting elements from the mix-max tree algorithm used.

	Time Complexity
Average Case	`Θ(||intervals||/||nodes||) * O(log(||nodes||))`
Worst Case	`O(||intervals||) * O(log(||nodes||))`
Best Case	`Ω(log(||nodes||))`

Table 6. Time complexity of adding a node

For example, for each interval, pop a node in `O(1)`, then re-insert the node in `O(log(nodes))`. The Min-Max queue maintains a sorted list of nodes so it isn't necessary to re-sort the queue after each operation. In the worst case there is a negligible number of nodes, so nearly all intervals must be iterated and re-inserted. In the best case, there is only a single interval per node so there are few intervals to iterate. The configurations leading to these upper and lower bounds are likely to undermine the fault tolerance of the system so the average case performance is most descriptive of the overall performance.

When adding new nodes to the system, the ratio of intervals to nodes must remain high enough to maintain the probability that the distribution of keys remains relatively uniform across the nodes. This process adds an additional linear term to the time complexity of adding a node. This is because it is necessary to iterate for `||intervals||` steps, since the fan out algorithm bisects existing intervals so after the operation there are `2 * ||intervals||` in total.

4.4.3 Removing a Node

The process of removing a node from the system is the reverse of adding one, so the complexities are similar and the same concerns apply.

	Time Complexity
Average Case	`Θ(||intervals||/||nodes - 1||) * O(log(||nodes - 1||))`
Worst Case	`o(||intervals||) * O(log(||nodes - 1||))`
Best Case	`Ω(log(||nodes - 1||))`

Table 7. Time complexity of removing a node

4.4.4 Resolving a Key to a Node

As with Chord [69][70], the primary operation of the client is to locate the responsible node by evaluating the interval-node assignment. Since F^2DR's intervals are uniformly distributed over the continuum, that hash table can be constructed to store the interval-node mapping using a perfect minimal hash function and providing `O(1)` look-ups [43]. This provides a tighter bound on the overall performance of the system as latency and overhead from handling collisions are reduced. This comes at the expense of ensuring that the list of intervals remains sorted.

The evidence for a constant time `resolve()` function is as follows. The resolution process can be composed of two operations, an integer division and an array look-up. While the integer division can be expensive in terms of CPU cycles, this operation runs in constant time. Since gap's in F^{2}DR are positive powers of two, division can be replaced with a right arithmetic bit-shift, which requires fewer cycles. Array look-ups can also be implemented in O(1) time if memory operations are disregarded. Since the `resolve()` function can be decomposed into two constant time components, the overall time complexity is `O(1)`.

Chapter 5: Discussion and Conclusion

5.1 Overview

The objective in creating F^{2}DR was to show that a consistent hashing system that could resolve requests in consistent time and provide support for with hundreds of intervals assigned to many clients can be constructed such that keys resolved to nodes in constant time and load among nodes can be dynamically managed. This will be accomplished though a consistent hashing look-up function that runs in constant time. Also proposed is a means of maintaining a uniform distribution of keys among nodes where there is flexibility in how uniformity of load is defined. The achievement of these goals provides node look-up latency can be described with stricter bounds due to the time complexity of the look-up algorithm and the tighter bounds on load achieved by the restructuring system.

Distributed storage is an essential requirement for computer systems that must be both scalable and performant. A number of studies and commentators have linked website performance with usability and customer satisfaction [45][52][54]. In particular, it has been found that users consider abandoning the use of websites when load times are just a few seconds [45][52]. Using A/B testing Google and Amazon correlated slower page loads with a signification decrease in their conversion rates [52]. A/B testing is an experiment where participants evaluate one of two choices, with only a minor difference between choices. This type of test is used to identify the choice likely to increase the chances of a desired outcome.

Another important concern is predictability of performance across the range of request and user types. For example, pages in the Amazon e-commerce application rely on many separate service requests to gather data [16], and poor performance by any of these could result in lost sales.

Many large distributed systems exist where subsets of a large quantity of data are retrieved by clients. In some of these systems, the number of clients is large and each client may make requests for data at a high rate. Some common examples of such systems are the websites Facebook [62], Live Journal [25], Amazon [16], Yahoo [68] and others.

Implemented in the system is a distributed look-up protocol based on DHT systems like Consistent Hashing [35][36] and Chord [69][70] and others. Also introduced is a novel interval division and constant time node look-up technique that draws inspiration from Bonwick's slab allocation algorithm [8]. Like Chord [69][70], data entries are looked up via keys. Nodes may join and depart the system at any time with a minimal disruption to the key to node mapping. This is due to the monotonic behaviour of consistent hashing functions [35][43].

In this chapter, the time complexity of various F^2DR operations will be contrasted against the original Consistent Hashing algorithm (constant and log time) [35], Chord [69][70] and Memcached [23]. The Dynamo [16] system cannot be included in this comparison because the authors have not published sufficient implementation details. The time complexity of the considered systems are summarized in Table 8. Discussion on potential security and performance issue is also included. For simplicity, the complexity analysis focuses on system construction, resolving a key to a node and the join or departure of nodes.

This remainder of this chapter will include further comparative analysis on the systems using following characteristics:

- time complexity
- interval & key distribution
- fault tolerance

Finally, will be some concluding remarks on future work and on the success of F^2DR in terms of the research goals set forth in Chapter 1.

5.2 Time Complexity Analysis

The comparative time complexity analysis will cover the following operations for the Consistent Hashing, Chord, Memcached and F^2DR systems:

- construction
- node resolution
- node join
- node departure

5.2.1 Construction

The Construction of each system is the process of initializing each data structure needed for the system to begin operation. The cost of this processes is comparatively small over a long period of operation, but the time-complexity of this provides insight into the performance of data structures used.

The fastest of the analyzed systems for construction is the Modulo variant of Memcached [24] as only a modulo value must be assigned to each node to define the intervals. F^2DR, the Consistent Hashing version of Memcached [33] and the other systems each perform in roughly quasilinear time (i.e. `nlog(n)`) based on the number of nodes per interval times the base-2 logarithm of the number of nodes. In either set of cases, the number of steps required to construct one of the systems is likely to be small.

5.2.2 Resolving a Key to a Node

For resolution, F^2DR requires constant time to resolve a key to a node, while several systems require logarithmic time. For the systems that provide constant time resolution, the authors of Consistent Hashing did not describe their implementation in detail [35] or provide an explanation for why this behaviour is not the default [43]. The modulo variant of Memcached is certainly fast, but can suffer from the absence of the fault tolerance properties of Consistent Hashing systems.

F^2DR has a few advantages over the descriptions of the other constant time systems. Unlike the original Consist Hashing [35][43] a discrete continuum of integers is used. Since integer math on current CPU's is faster than floating point. Also, many common hash algorithms such as MD5 [61] and SHA1 [20] already produce integers, so keys can be mapped directly onto the continuum domain. The use of integers also avoids the precision issues of floating point arithmetic so ensuring uniform precision around the entire continuum.

F^{2}DR matches the speed of the naive modular hashing scheme in Memcached [24] and the constant time Consistent Hashing variant [43], while also providing the stability of consistent hashing. Additional benefits are fast node resolution, a flexible restructuring algorithm and could provide tighter bounds on response times. In situations where routing is performed many times a second and systems comprise dozens or more nodes, the system is expected to have a modest speed advantage. However, this advantage can only be realized when communication has very low latency and high bandwith. If communication delays are introduced, there may not be an advantage over the logarithmic time complexity of other solutions. The resolution speed is very important for applications that must meet strict service level agreements (SLA's), such as Amazon [16].

5.2.3 Join and Depart

The join and depart operations are analyzed together because they have similar average case time complexities across the systems. As with construction, the constant time modulo variant of Memcached [24] has the fastest time complexity because of the simplicity of the node-interval mapping operation. Consistent Hashing [35][36][43], Chord [69][70] and F^{2}DR have similar time complexities, but differ in several key ways. In the original Consistent Hashing system, additional load balancing was achieved by assigning keys using one of k randomly selected hash functions [35][43]. In Chord [69][70], it is necessary to communicate with multiple nodes to update their routing tables so round trip times could cause a substantial delay to the operation. F^{2}DR has the advantage of not requiring multiple hash functions and a shared-nothing architecture so no synchronization between nodes is needed.

Operations	Systems					
	Consistent Hashing [35][36][43]		Chord [69][70]	Memcached [24][33]		F²DR
	Original	Constant		Modulo	Consistent	
Construction	$O(knlog_2kN)$	$O(knlog_2kN)$	Not Available	$O(N)$	$O(Inlog_2(N))$	$O(Ilog_2(n))$
Resolve	$O(log_2kN)$	$O(1)$	$O(Llog_2(n) + log_2(I/n))$	$O(1)$	$O(log_2(I))$	$O(1)$
Join	$O(log_2kN)$	$O(log_2kN)$	$O(log_2(I) * RTT)$	$O(N)$	$O(nlog_2N)$	$O(nlog_2(N))$
Depart	$O(log_2kN)$	$O(log_2kN)$	$O(log_2(I) * RTT)$	$O(N)$	$O(nlog_2N)$	$O(nlog_2(N))$

Table 8. Summary of comparative time complexity of F2DR vs. Consistent Hashing, Chord and Memcached. This complexity functions are summarized from section 2.4 through 2.6.

I – The total number of intervals.

N – The total number of nodes

n – The number of intervals per node

k – The number of hash functions used

L – The communication latency between two nodes

RTT – The communication latency to communicate with a node and receive a response (or ~$2*L$)

5.2.4 Interval and Key Distribution

For intervals and key distribution, a specific advantage or disadvantage over the other systems is not claimed, but F^{2}DR's design has few drawbacks and allows for a range of distribution characteristics. To be specific, the online balls and bins problem (where bins are uniform) has been well studied, which includes theoretic bounds on bin load. There also exist several useful sources on how load can be further reduced, for example, by placing balls in the least loaded of a set of bins.

Unlike the other systems in this analysis, F^{2}DR can take advantage of the NAQ to re-balance the system over the course of operation. When a node joins or departs, or even at an arbitrarily chosen time, restructuring can be performed to spread heavily populated or requested intervals (as well as under populated ones) between nodes. Short of rebuilding the entire node-interval mapping (at great expense), the other systems lack an adaptive way of restructuring. Restructuring operations are either fixed where fall back to a predecessor node occurs [70] or at random when intervals are created for a new node [24]. This allows a chance for imbalance to grow through subsequent restructuring operations. While Chord's [69][70] restricted views of the network for each node make this system excellent for wide area networks, the communication latency needed to communicate system changes to nodes is high.

While there is additional cost though in F^{2}DR to support the NAQ and uncertainty on which concept of load is used to order the queue. Restructuring requires logarithmic time relative to the number of intervals used, which is comparable to other systems. The holds true as long as there are sufficient intervals to be assigned to each node. A downside is when the ratio of intervals to nodes is small, then restructuring creates n new intervals at the midpoint of each existing interval. If these new intervals are added to the system, it is very expensive because $O(n/2)$ entries will be evicted. In comparison, the standard approach creates a smaller proportion of intervals, resulting in only $O(1/N)$ entries evicted. However, in both cases, if intervals are added over time, few or no entries need be evicted. This is likely an advantage when a node is being added in a situation where nodes (or the resource they proxy for) are already overloaded and increasing the rate of misses will worsen this situation.

5.3 Fault Tolerance Analysis

A major design goal of Consistent Hashing and other predecessor systems was fault tolerance. This section outlines some of the challenges that can be expected in the operation of F^2DR or a comparable system.

5.3.1 Preventing Denial of Service (DOS) Attacks

In the case where the adversary has little or no knowledge of the internal state of the system, the analyzed systems are resistant to this type of attack. Since the load will have a roughly uniform distribution between nodes, there is not a clear weak point that can be overwhelmed. Also, knocking a small number of nodes out of the system need not be catastrophic since load can be re-routed to the remaining nodes. An attacker with knowledge of the state of the system could specifically craft requests directed to a single node, perhaps overburdening that node. The state information required for this is the random seed used to synchronize the system, along with the number of nodes, and the number of intervals per node. The attacker would also need access to make requests against the system. However, DHT systems tend to be resistant by design to individual nodes departing unexpectedly (or being knocked offline). For this reason, this resistance is somewhat effective against DOS attacks.

There are several approaches to mitigate this type of problem. First, direct access to the system must be restricted to trusted parties, which will limit exploits based on the inner workings of the system. Indirect accesses, such as with information submitted to a web application using the system, must also be secured. For example, a dictionary attack launched through the keys and based on the distributions of known hash functions can be limited using a randomized hashing scheme (i.e. n-random hash functions or a hash salt) [35][53].

For the remaining information on system state, there is little that can be done. If an attacker is able to gain enough access obtain system configuration values, such as the node or interval count, there is a strong possibility that the machine hosting the client could be compromised.

5.3.2 Cryptographic vs. Non-cryptographic Hash Functions

The contrast between cryptographic vs. non-cryptographic hash functions is a trade-off between security and retrieval convenience. F^2DR and the other evaluated systems are able to operate on keys generated by either type of function. Cryptographic functions have the benefits of being unpredictable and reasonably fast to compute while a non-cryptographic hash is useful when insufficient information is available to reproduce an encrypted key. The benefit of cryptographic functions is the defence from denial of service attacks (intentional or not). This is because similar inputs will usually be mapped to very different areas of the continuum due to the “avalanche effect”. Attempting to overload one or more nodes by pre-selecting keys is equivalent to decrypting specific cryptographic hash values, a task known to be computationally difficult. A weakness exists in this approach in the form of key distribution. Cryptographic hash functions do not provide guarantees on the distribution for a subset of the possible keys. Only given a large enough set of keys will the function tend toward filling the range which may result in a non-uniform distribution. Many systems will not contain enough records to achieve this for the 128 bit MD5 [61] or 160 bit SHA1 [20] functions.

Alternatively, a function that performs horizontal partitioning (or sharing) can be used on the key. In this case keys fall consecutively across the continuum in a predictable fashion. Although this is only suitable for systems where clients are trusted, there several benefits. The predictability of this approach makes it much easier to design a key selection algorithm. It is possible to project the expected key distribution across the entire continuum if the keys will be roughly continuous and if number of keys can be approximated. It is also much easier to retrieve consecutive records since there are fewer nodes to contact. This is very useful for applications that need to operate on large sets of data, although intervals must be large enough to contain multiple records of the same type.

Unfortunately, the stark division between these two approaches that makes it difficult to strike a compromise. If the benefit is large enough though, it should be possible to design a function that has some cryptographic behaviour. For example, if the conditions for the avalanche effect are reduced such that keys will be grouped together but the groups are not necessarily consecutive on the continuum.

5.3.3 Deterministic Behaviour for Node Joins and Departures

F^{2}DR and the proposed systems use a 'Shared Nothing' architecture in which neither CPU, memory or storage need be shared among nodes [71]. For the system to perform well, all system clients must have a consistent view of the available nodes, for the majority of the time. This requires interval states to be synchronized between nodes which could cause delay in restructuring operations. The system need not support distributed transactions, but failure to synchronize interval states will cause the entries for some keys to become duplicated between nodes. Synchronization must occur within a reasonably narrow window to prevent excessive inserts to nodes that will no longer be assigned those keys. This behaviour is problematic for applications that expect entries to be available immediately after they are inserted.

A means of performing this task is to use an algorithm for allocating intervals that has deterministic behaviour as in [74]. If it is possible for clients to agree in advance on a scheme for further interval assignments so that eventual consistency [77] is achieved. This does place practical limitations on what criteria can be considered in the selection function though. In order to include fine grained load information, the problem of synchronization between clients must be considers to insure that clients can agree on system structure. The 'gossip' approach in Chord [69][70] for communicating changes to node membership may provide a better means of accomplishing this. Even if consistency between nodes cannot be archived via gossip, the faster nodes can be informed of system-wide changes, the sooner these changes can be initiated.

5.3.4 Catastrophic Node Loss

Losing a large number of nodes at once would have an adverse affect to all of the evaluated systems. This scenario would be encountered if there was a network connectivity disruption occurs between groups of nodes. On top of the overhead to reallocate the intervals assigned to the departing nodes, the remaining nodes may need to write many new cache entries after cache misses occur. A similar problem could occur for computational services if requests require additional data to be loaded into memory. Once this process of shifting load starts, little can be done but to weather the storm until the nodes can reach a steady performance state. Since there is no shared state between nodes, the separated groups of nodes can operate independently. The penealty of this is a reduced capacity for entries and reduced efficiency due to duplicated entries between node groups.

What can be done prior to node loss is to ensure that a sufficient number of nodes are in the system to begin with. Also, if a roughly uniform amount of load can maintained between each node then there is a less chance of an already strained node being over-burdened. Each system in the analysis attempts to achieve uniform load by spreading many intervals across the nodes or as in [35] providing a hierarchical mirror system. However, both Memcached [24][33] implementations use intervals of random size, with no control on how intervals are distributed on the continuum. This means that intervals that happen to be larger could be overloaded while there are smaller intervals with too little load. Aside from using equal sized intervals, the load assessment ability of F^{2}DR's NAQ could lessen the impact of this problem by evenly distributing overloaded and underloaded intervals between nodes.

5.3.5 Frequent Node Joins or Departures

Frequent system-wide change to F^{2}DR or any of the other evaluated systems causes entries stored against reallocated intervals to move between nodes. This results in performance loss because clients will need to will decrease if clients to the system are frequently reallocating intervals. This increased latency because the clients are busy updating interval assignments. This is undesirable as a greater number of misses will occur until entries become resident on their new nodes. For unplanned departure, striving to maintain a balanced quantity of entries between nodes is most helpful. This way the bound on the number of entries that are vacated is tighter to `1 / ||nodes||`. When a join or departure is planned then F^2DR could throttle the rate at which intervals are assigned to the new node. This has the advantage of reducing the miss rate.

Another situation that may be faced when joins or departures occur frequently is that the system enters a state where it is continuously restructuring. This will hinder system performance, so a cool-down period for changes could be applied the prevent this. During the cool-down, system changes could be queued and applied in batches. It may be possible to re-order system changes to further reduce the performance impact, such as first reassigning intervals from a departing nodes to ones that are joining.

5.3.6 Overloaded Nodes

While the number of intervals assigned to each node will be balanced, the load associated with some intervals will be higher than others. Some intervals may contain a larger number of entries, which incurs more frequent requests. Also, some individual entries may be accessed at a higher rate than others. Finally, some entries may be more expensive to request than others. This may be due to differences in costs to retrieve or transmit the entry. The method proposed in the system to deal with this problem is to trade 'busier' intervals from a node with high load to one with lower load. The effectiveness of this approach is subject to the selection heuristic. In this scenario, both nodes should be write-locked to avoid 'ghost writes'. For this to be effective, the interval selected from the loaded node must fit within the load capacity of the new load. The process for minimizing the resulting load on each node after the interval swap would be useful, but is beyond the scope of this work.

5.3.7 Too Many Sequential Intervals Assigned to A Node

For traditional Consistent Hashing functions, intervals assigned to a departing node would fall back to the assignee of an adjacent interval in the continuum [23][35]. The interval-node assignment mappings will change as nodes join or depart. Over time it is possible for one node to be assigned intervals that are sequential or are unrepresentative of the entire range of the continuum. There are several downsides to this. If one such node drops from the system, many sequential entries will suddenly become unavailable at once. Afterwards, these entries will all need to be migrated to a single adjacent node, which worsens the situation [23][35]. F^2DR avoids this by reassigning intervals at random among the remaining nodes. This allows us to consult the selection heuristic and provides a measure of control against unbalance. This does incur some performance penalty but this should not be a problem unless there are many thousands of intervals or joins or departures occur too frequently. Greater aggressiveness can be also be applied to avoid bad assignments, by selecting multiple possible assignees and testing their load. A simple and fast approach like 'Always go left' [53] approach could resolve these issues.

5.3.8 Synchronization Issues due to number of clients

For systems on a LAN with a few thousand or less clients, it is feasible to push updates to all clients. This reduces the window in which clients have inconsistent views of the system. However, on larger systems or those on a high latency or WAN network, this may not be possible. Of the analyzed systems, the approach of Chord [69][70] handles this well, where all nodes do not need to be aware of each other. The overhead for this approach could be improved using our resolution algorithm to search Chord's finger table [69][70]. This will be helpful as long as the communication latency between nodes is not much greater than the time to search for a particular mapping.

5.4 Future Work and Conclusion

The work is concerned with the partitioning of an online stream of data among a fluid set of computing resources. The resulting function for assigning intervals to nodes is monotonic (strictly increasing, or decreasing), requires minimal space (all 'buckets' are used), has no collisions, can be computed in constant time and allows for graceful restructuring if the number of computing resources change. The resulting algorithms used in F^2DR are applicable in the areas of distributed computing, caching, network routing, message queues and others.

There are several aspects of this work that would benefit from future investigation. This work has not provided a thorough theoretical analysis on the benefits of F^2DR's Node Assignment Queue (NAQ) so the claimed benefits cannot are not substantiated. It would be useful to run simulations with sample key-sets in order to evaluate the load balancing performance. This could be done by selecting several system configurations (node count, interval gap, *etc.*) and populating each with keys sampled from several numerical distributions such as random, power law, poisson, *etc*. Nodes should also join and depart during this process to evaluate load balance after restructuring. An appropriate statistical test would need to be selected for comparing the actual load distribution against the uniform distribution. Once assembled, these tests could be assembled into an analysis toolkit for evaluating the behaviour of a running system, or to inform decisions on how many nodes to provision and how many intervals to use.

While it has been suggested that selection of NAQ's priority function can make it possible to detect load imbalance or to signal that restructuring should be performed, this work has not provided direct guidance on specific ways this can be accomplished. So the algorithms presented for the NAQ presented can only be treated as a starting point.

Although not discussed in this work, the handling of multidimensional data by consistent hashing could be of some interest for data categorization problems. For example, consider a set containing many subsets of fluid elements such as a set of business expense accounts. Consider that these accounts are logical groupings of similar expenses, but based on business needs this system could be reorganized so that expenses may move to any other account or that new account types could be added and others removed. Consistent Hashing [35][36][43] has only a one dimensional domain and range and its behaviour when nodes depart entries contained in that that node's intervals are subsumed by a node with a neighbouring interval. Back to our accounting example that my have $n \times n$, logical dimensions, adjacent accounts may not have any relation to the account being removed, but expenses that were contained must be moved to an appropriate account.

The goal of future study in this area would be to create a consistent hashing function that is capable of representing more than two adjacent intervals and reassigning entries to the most 'similar' interval. Results of this study could have implications for large classification systems with many inter-dependencies between classes and a high overhead for restructuring.

This product of this work was DHT system, F^2DR that achieves the goal of performing node-interval resolution in constant time and linear space, while also retaining the fault tolerance and stability of consistent hashing. These included comparative analysis show that the theoretical performance of F^2DR improves on that of several other systems for the resolution operation and there is potential to provide better load balancing characteristics.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, “Farsite: federated, available, and reliable storage for an incompletely trusted environment,” SIGOPS Oper. Syst. Rev., vol. 36, pp. 1–14, Dec. 2002.

[2] “Amazon Web Services,”. amazon.com, 2006. [Online] Available: http://aws.amazon.com/.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, “Balanced allocations (extended abstract),” in Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, pp. 593–602, 1994.

[4] R. Alonso, D. Barbara, and H. Garcia-Molina, “Data caching issues in an information retrieval system,” ACM Trans. Database Syst., vol. 15, 1990, pp. 359-384.

[5] M. D. Atkinson, J.-R. Sack, B. Santoro, and T. Strothotte, “Min-max heaps and generalized priority queues,” Commun. ACM, vol. 29, no. 10, pp. 996–1000, Oct. 1986.

[6] J. Atwood, “Performance is a Feature,” Coding Horror. June 20th, 2011. [Online] Available: http://www.codinghorror.com/blog/2011/06/performance-is-a-feature.html.

[7] J. Atwood and J. Spolski, “StackExchange.” 2008. . [Online] Available: http://stackexchange.com.

[8] J. Bonwick, “An Object-Caching Kernel Memory Allocator,” USENIX Summer 1994 Technical Conference.

[9] J. Byers, J. Considine, and M. Mitzenmacher, “Simple Load Balancing for Distributed Hash Tables,” in Peer-to-Peer Systems II, 2003, pp. 80-87.

[10] M.J. Carey et al., “Data caching tradeoffs in client-server DBMS architectures,” Proceedings of the 1991 ACM SIGMOD international conference on Management of data, Denver, Colorado, United States: ACM, 1991, pp. 357-366.

[11] M. Castro and B. Liskov, “Practical byzantine fault tolerance and proactive recovery,” ACM Trans. Comput. Syst., vol. 20, no. 4, pp. 398–461, Nov. 2002.

[12] A. Chankhunthod et al., “A Hierarchical Internet Object Cache,” Proceedings of the USENIX 1996 Annual Technical Conference, USENIX Association, 1996.

[13] B. Chidlovskii and U.M. Borghoff, “Semantic caching of Web queries,” The VLDB Journal, vol. 9, 2000, pp. 2-17.

[14] T.H. Cormen et al., Introduction to Algorithms, MIT Press and McGraw-Hill, New York, USA, 2001.

[15] D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON). RFC 4621, July 2006.

[16]G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” SIGOPS Oper. Syst. Rev., vol. 41, no. 6, pp. 205–220, 2007.

[17]L. Degenaro et al., “A middleware system which intelligently caches query results,” IFIP/ACM International Conference on Distributed systems platforms, New York, New York, United States: Springer-Verlag New York, Inc., 2000, pp. 24-44.

[18]P.M. Deshpande et al., “Caching Multidimensional Queries Using Chunks,” In Proceedings of the ACM SIGMOD Conference on Management of Data, 1998, pp. 259-270.

[19]M. Dietzfelbinger, A. Karlin, K. Mehlhorn, H. Rohnert, and R. E. Tarjan, “Dynamic perfect hashing: Upper and lower bounds,” in Foundations of Computer Science, 1988., 29th Annual Symposium on, pp. 524–531, 2002.

[20]D. Eastlake and P. Jones, US secure hash algorithm 1 (SHA1). RFC 3174, September, 2001.

[21]R.J. Enbody and H.C. Du, “Dynamic hashing schemes,” ACM Comput. Surv., vol. 20, 1988, pp. 850-113.

[22]“Facebook,” Facebook, 2004. [Online]. Available: http://www.facebook.com/.

[23]B. Fitzpatrick, “Distributed caching with memcached,” Linux J., vol. 2004, 2004, p. 5.

[24]B. Fitzpatrick, “memcached: a distributed memory object caching system,” 2004. [Online] Available: http://danga.com/memcached/.

[25]B. Fitzpatrick and L. Phillips, “LiveJournal’s Backend and memcached: Past, Present, and Future,” Atlanta, Georgia, United States, 17-Nov-2004.

[26]A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, “Cluster-based scalable network services,” SIGOPS Oper. Syst. Rev., vol. 31, pp. 78–91, Oct. 1997.

[27]M. L. Fredman, J. Komlos, and E. Szemeredi, “Storing a sparse table with O(1) worst case access time,” in Foundations of Computer Science, 1982. SFCS '08. 23rd Annual Symposium on, pp. 165-169, 1982.

[28]C. Galindo-Legaria et al., “Database change notifications: primitives for efficient database query result caching,” Proceedings of the 31st international conference on Very large data bases, Trondheim, Norway: VLDB Endowment, 2005, pp. 1275-1278.

[29]T. Haerder and A. Reuter, “Principles of transaction-oriented database recovery,” ACM Comput. Surv., vol. 15, no. 4, pp. 287–317, Dec. 1983.

[30]T. Hagerup and T. Tholey, “Efficient minimal perfect hashing in nearly minimal space,” STACS 2001, LNCS 2001, pp. 317–326, 2001.

[31]J. Hamer, “Hashing revisited,” SIGCSE Bull., vol. 34, 2002, pp. 80-83.

[32]A. Hasham and J.-R. Sack, “Bounds for min-max heaps,” BIT, vol. 27, no. 3, pp. 315–323, Sep. 1987.

[33]R. Jones, “libketama - a consistent hashing algo for memcache clients – Last.fm,” RJ's Journal.

[34]R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, “H-store: a high-performance, distributed main memory transaction processing system,” Proc. VLDB Endow., vol. 1, no. 2, pp. 1496–1499, Aug. 2008.

[35]D. Karger et al., “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web,” Proceedings of the 29th Annual ACM Symposium on Theory of Computing, 1997, pp. 654–663.

[36]D. Karger et al., “Web caching with consistent hashing,” Proceedings of the eighth international conference on World Wide Web, Toronto, Canada: Elsevier North-Holland, Inc., 1999, pp. 1203-1213.

[37]R. M..Karp, M. Luby, and F. M. A. D. Heide, “Efficient PRAM simulation on a distributed memory machine,” in Proceedings of the twenty-fourth annual ACM symposium on Theory of computing, pp. 318-326, 1992.

[38]K. Kato and T. Masuda, “Persistent caching: an implementation technique for complex objects with object identity,” Software Engineering, IEEE Transactions on, vol. 18, 1992, pp. 631-645.

[39]S.N. Khoshafian and G.P. Copeland, “Object identity,” SIGPLAN Not., vol. 21, 1986, pp. 406-416.

[40]L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” Commun. ACM, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[41]L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem,” ACM Trans. Program. Lang. Syst., vol. 4, pp. 382–401, Jul. 1982.

[42]P. J. Leach, M. Mealling, and R. Salz, RFC4122, A Universally Unique IDentifier (UUID) URN Namespace. IETF, 2005.

[43]D. M. Lewin, “Consistent hashing and random trees: algorithms for caching in distributed networks,” Thesis, Massachusetts Institute of Technology, 1998.

[44]T.G. Lewis, B.J. Smith, and M.Z. Smith, “Dynamic memory allocation systems for minimizing internal fragmentation,” Proceedings of the 1974 annual conference on XX - Volume 2, ACM, 1974, pp. 725-728.

[45]G. Linden, “Marissa Mayer at Web 2.0,” Geeking with Greg. 09-Nov-2006.

[46]W. Litwin, M. Neimat, and D.A. Schneider, “LH* - a scalable, distributed data structure,” ACM Trans. Database Syst., vol. 21, 1996, pp. 480-525.

[47]B. Liu, W. Lee, and D.L. Lee, “Distributed caching of multi-dimensional data in mobile environments,” Proceedings of the 6th international conference on Mobile data management, Ayia Napa, Cyprus: ACM, 2005, pp. 229-233.

[48]“LiveJournal,” 1999. [Online]. Available: http://www.livejournal.com/.

[49]Y. Lu, B. Prabhakar, and F. Bonomi, “Perfect Hashing for Network Applications,” in Information Theory, 2006 IEEE International Symposium on, pp. 2774-2778, 2006.

[50]Q. Luo et al., “Middle-tier database caching for e-business,” Proceedings of the 2002 ACM SIGMOD international conference on Management of data, Madison, Wisconsin: ACM, 2002, pp. 600-611.

[51]R. Malda, “Slashdot’s Setup, Part 2- Software,” Slashdot. 26-Oct-2007. [Online] Available: http://news.slashdot.org/story/07/10/22/145209/slashdots-setup-part-2—software.

[52]M. Meyer, “What Google Knows,” 09-Nov-2006. [Online] Available: http://www.web2con.com/pub/w/49/schedule.html.

[53]M. Mitzenmacher, “The Power of Two Choices in Randomized Load Balancing,” IEEE Transactions on Parallel and Distributed Systems, vol. 12, no. 10, pp. 1094-1104, 2001.

[54]J. Nielsen, Usability Engineering. Morgan Kaufmann, 1994.

[55]M. G. Norman, T. Zurek, and P. Thanisch, “Much ado about shared-nothing,” SIGMOD Rec., vol. 25, no. 3, pp. 16–21, Sep. 1996.

[56]N. Tolia and M. Satyanarayanan, “Consistency-preserving caching of dynamic database content,” Proceedings of the 16th international conference on World Wide Web, Banff, Alberta, Canada: ACM, 2007, pp. 311-320.

[57]M. Pease, R. Shostak, and L. Lamport, “Reaching Agreement in the Presence of Faults,” J. ACM, vol. 27, no. 2, pp. 228–234, Apr. 1980.

[58]J. Petrovic, “Using Memcached for Data Distribution in Industrial Environment,” Systems, 2008. ICONS 08. Third International Conference on, 2008, pp. 368-372.

[59]C. Plaxton and R. Rajaraman, “Fast fault-tolerant concurrent access to shared objects,” Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on, 1996, pp. 570-579.

[60]V. Ramasubramanian and E. G. Sirer, “Beehive: Exploiting power law query distributions for O (1) lookup performance in peer to peer overlays,” in Symposium on Networked Systems Design and Implementation, San Francisco CA, San Francisco, California, 2004.

[61]R. Rivest, The MD5 Message-Digest Algorithm. RFC 1321, April, 1992.

[62]P. Saab, “Scaling memcached at Facebook,” Facebook Engineering. 12-Dec-2008.

[63]A.N. Saharia and Y.M. Babad, “Enhancing data warehouse performance through query caching,” SIGMIS Database, vol. 31, 2000, pp. 43-63.

[64]S. Sanfilippo and P. Noordhuis, “Redis.” [Online]. Available: http://redis.io/.

[65]Z. E. Schnabel, “The estimation of total fish population of a lake,” American Mathematical Monthly, vol. 45, no. 6, pp. 348–352, 1938.

[66]R. Schwartz, L. Laporte, and J. Berkus, “FLOSS Weekly 18: Josh Berkus on PostgreSQL”. November 2nd, 2008 [Online] Available: http://twit.tv/floss18.

[67]“Slashdot,” 1997. [Online] Available: http://slashdot.org/.

[68]M. Sordo, “Mixer – The Data Service that Powers Yahoo! News Activity,” Yahoo Developer Network. 22-September 22nd, 2011. [Online] Available: http://developer.yahoo.com/blogs/ydn/posts/2011/09/mixer-%E2%80%93-the-data-service-that-powers-yahoo-news-activity/?-the-data-service-that-powers-yahoo-news-activity/.

[69]I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, 2001, p. 160.

[70]I. Stoica et al., “Chord: a scalable peer-to-peer lookup protocol for internet applications,” IEEE/ACM Transactions on Networking (TON), vol. 11, no. 1, p. 32, 2003.

[71]M. Stonebraker, “The case for shared nothing,” Database Engineering Bulletin, vol. 9, no. 1, pp. 4–9, 1986.

[72]M. Stonebraker, “VoltDB The New SQL database for high velocity applications.” [Online]. Available: http://voltdb.com/.

[73]J. Teuhola, “Effective clustering of objects stored by linear hashing,” Proceedings of the fourth international conference on Information and knowledge management, Baltimore, Maryland, United States: ACM, 1995, pp. 274-280.

[74]A. Thomson and D. J. Abadi, “The case for determinism in database systems,” Proc. VLDB Endow., vol. 3, no. 1–2, pp. 70–80, Sep. 2010.

[75]J.S. Vitter, “External memory algorithms and data structures: dealing with massive data,” ACM Comput. Surv., vol. 33, 2001, pp. 209-271.

[76]B. Vöcking, “How asymmetry helps load balancing,” J. ACM, vol. 50, no. 4, pp. 568-589, 2003.

[77]W. Vogels, “Eventually consistent,” Commun. ACM, vol. 52, no. 1, pp. 40–44, 2009.

[78]E. W. Weisstien, "Dirichlet's Box Principle." [Online] Wolfram Math World Available: http://mathworld.wolfram.com/DirichletsBoxPrinciple.html.

[79]J.G. Williams, “Storage utilization in a memory hierarchy when storage assignment is performed by a hashing algorithm,” Commun. ACM, vol. 14, 1971, pp. 172-175.

[80]K. Yagoub et al., “Caching Strategies for Data-Intensive Web Sites,” Proceedings of the 26th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., 2000, pp. 188-199.

[81]“The Official YAML Web Site”. [Online] Available: http://www.yaml.org/.

[82]“13.1 pickle -- Python object serialization”. [Online] Available: http://www.python.org/doc/2.5.2/lib/module-pickle.html.

[83]“Discover the secrets of the Java Serialization API”. [Online] Available: http://java.sun.com/developer/technicalArticles/Programming/serialization/.

[84]“Extensible Markup Language (XML) 1.0 (Fifth Edition)”. [Online] Available:http://www.w3.org/TR/REC-xml/.

[85]“PHP: serialize - Manual”; http://ca3.php.net/manual/en/function.serialize.php.

[86]“YouTube Content ID,” YouTube. [Online] Available: http://www.youtube.com/t/contentid.