Chaining For Flexible And High-Performance Key-Value Systems

Amar Phanishayee

CMU-CS-12-139

September 2012

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen, Chair
Garth A. Gibson
Timothy Roscoe, ETH Zurich
Srinivasan Seshan

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers CNS0546551, CNS0716287, and CCF0964474; by the Defense Advanced Research Projects Agency under grant number HR00110710025; by an IBM Ph.D. Fellowship award; and by gifts from Network Appliance, Intel, and Google.

Keywords: Key-Value Systems, Replication Protocols, Data Consistency, Flash

For K. Shantha (Amma) and S.D. Phanishayee (Appa):
my most influential teachers,
inspirational role models,
and my best friends.

And for Laura:
my constant source of enthusiasm, creativity, and love.

ॐ असतो मा सदगमय।
तमसो मा ज्योतिर्गमय।
मृत्योर्मा अमृतं गमय।
ॐ शान्ति शान्ति शान्ति॥

From ignorance, lead me to truth; From darkness, to light; From death, to immortality.
Let there be peace. AUM
(Brhadāranyaka Upanishad 1.3.28)

Abstract

Distributed key-value (KV) systems are a critical part of the infrastructure at many large sites such as Amazon, Facebook, Google, and Twitter. The first research question this dissertation addresses is: *How should we design a cluster-based key-value store that is fault tolerant, achieves high performance and availability, and offers strong data consistency?* We present a new replication protocol, Ouroboros, which extends chain-based replication to allow fast, non-blocking node additions to any part of the replica chain, and guarantees provably strong data consistency. We use Ouroboros to implement a distributed key-value system, FAWN-KV, designed with the goal of supporting the three key properties of fault tolerance, high performance, and generality. We present a formal proof of correctness of Ouroboros, and evaluate FAWN-KV on clusters with Flash storage.

FAWN-KV is, still, only a specific KV solution that offers strong data consistency and is optimized for clusters that have storage devices with slow random writes. The current diversity in hardware and application requirements have resulted in a plethora of KV systems today, with no one system meeting the needs of all applications. The second, and final, research question this dissertation addresses is therefore: *Is it possible for a KV architecture to be easily configured to support many points along the KV system design continuum?* We present a generalization of chain-based replication, Ouroboros+, which extends Ouroboros to effectively support a wide range of application requirements by (a) selecting from different update protocols between replicas, and, (b) selecting a query node in a replica chain. We describe Flex-KV, which uses Ouroboros+ with different datastores that expose a common storage interface to form heterogeneous replica chains. Flex-KV can support DRAM, Flash, and disk-based storage; can act as an unreliable cache or a durable store; and can offer strong or weak data consistency. The value of such a system goes beyond ease-of-use: We enable new choices for system designs that offer some applications a better balance of performance and cost than was previously available.

Acknowledgments

I am forever indebted to my advisor, David Andersen, who was the perfect mentor through my six years of graduate school at CMU. Dave's creativity, emphasis on intuition & visual aids, and his contagious excitement to tackle interesting research problems drew me to him even before accepting to join the Ph.D. program at Carnegie Mellon. Dave has been an inspirational teacher, always leading by example and never sermonizing. There are many things I learnt from Dave: the art of systems building; the rigorous science of evaluating systems; and the craft of precise, yet simple and concise, technical communication. I also learnt from Dave the fearlessness and persistence of questioning to tease out the core of a research idea. Dave was supportive of everything I did, both academically and otherwise, at CMU. He also got me all the resources I might have needed without batting an eyelid. Dave has been central in improving the technical quality of every part of this thesis. For these, and much more, I express my utmost gratitude to him.

Garth Gibson, Srinivasan Seshan, and Timothy Roscoe were kind enough to serve on my dissertation committee and to guide me along the way all these years. Garth's propensity to identify important research problems has left me in a state bordering reverential respect for his intellect; for me, Incast will always serve as a reminder of Garth's vision. Similarly, I have always been in awe of Srini's unique ability to be a couple of reasoning steps ahead of everyone else in the room, of asking clear questions that get straight to the point, and of his mastery of popular networking protocols like TCP (and all its variants). Finally, Mothy was kind enough to host me at ETH Zurich for six months. I learnt, and reasoned, more about research operating systems from Mothy than I ever had before; I don't think I will ever forget our discussion on *naming*. Garth, Srini, and Mothy have always been kind and friendly to me, and I thank them.

I also owe a great deal of gratitude to Ken Birman, my mentor at Cornell; I would not have been in graduate school without Ken's guidance. Ken encouraged and funded me–a masters student with no research experience–to help design and build the next generation of reliable multicast protocols. It was always a thrill to hear Ken's vision of clustered systems. His ability to constantly generate new ideas still amazes me. I was also equally struck by how easily approachable Ken was, and by how well he treated all his students (as an extended family). Paul Francis, P. Venkat Rangan, and Sin-Mei Tsai also encouraged me to apply to grad school; I thank them for their kindness.

Mahesh Balakrishnan initiated me into graduate school life, but more importantly, he taught me the intricacies of systems research and paper writing. It was refreshing to be around Mahesh: always creative, always funny. Pranam Pitamah. Incidentally, Mahesh was also my host when I first landed at Cornell and was looking for a place to live; hopefully, I repaid him later by spotting a "burglar" entering his house at midnight.

My work as a Ph.D. student was made possible, and constantly enjoyable, due to the collaboration with one of my closest friends, Vijay Vasudevan; we spent a lot of time together thinking about research problems, arguing about impact, spending sleepless nights writing class reports and papers on Incast & FAWN, hacking large code bases, running experiments, and, needless to say, goofing off. Without Vijay, grad school would have been nowhere close to this much fun.

I also had loads of fun working on Ditto with Fahad Dogar and Olatunji Ruwase. I clearly remember our euphoria when Ditto reconstructed chunks of data using overheard TCP streams. I also remember gawking at Zen master TJ's *crazy* gdb skills.

Building the FAWN cluster was no mean feat; apart from hacking on FAWN-KV, Jason Franklin was my partner in crime for many long days that included splitting and soldering wimpy power cables, configuring PXE boot for a non-standard hardware configuration, and visiting the CMU School of Art's Woodshop to get help in building a FAWN rack.

Kanat Tangwongsan helped me get into my office after I had locked myself out on my first weekend at CMU. But his kindness did not stop there; Kanat's help was critical in helping prove the correctness of Ouroboros.

I consider myself very fortunate to have worked with some incredibly bright student co-authors and collaborators at CMU: Elie Krevat, Himabindu Pucha, Hiral Shah, Lawrence Tan, Jack Ferris, Ankur Goyal, David Sontag, and Yang Zhang. You guys rock. So do, for the lack of a better group name, fellow *fawn-dev* hackers: Bin Fan, Iulian Moraru, Hyeontaek Lim.

I also shared office space with some super smart students, working in areas other than systems at CMU. Juri Leskovec, Anna Goldenberg, Runting Shi, Adam Wierman, David McWherter, Deepak Garg, Michael De Rosa, and Henry DeYoung: each one of you taught me something new and useful.

A shout-out to my *in-school-by-default* fellow students at CMU. Vyas Sekar, Varun Gupta, Avijit Kumar, Gaurav Veda, Hetunandan Kamisetty, and Swapnil Patil: you guys inspire me by working and playing hard. Milo Polte, Jiri Simsa, Stephen Oney, Jim McCann, Ronit Slyper: our social interactions have enriched my CMU experience.

Being a doctoral student at CMU has been a superlative experience. The Reasonable Person Principle, this department's Prime Directive, is incredibly effective. At times, it is difficult to tell the difference between students and faculty; students face no bureaucracy, and are offered the freedom and the resources to pursue their creative dreams. The people who make the magic happen are part of an elite group of superheroes; Deborah Cavlovich, Sharon Burks, Angela Miller, Karen Lindenfelser, Joan Digney, Jennifer Landefeld, and Kathy McNiff: I can't thanks you enough for what you have done for me, not only when I yelled out for help, but especially in those occasions when I was oblivious to the fact that you were working hard to ensure that we were comfortable and could concentrate on our *work*.

The Parallel Data Lab (PDL) taught me the importance of practicing & polishing technical talks. Greg Ganger, PDL's fearless leader, taught me two of the most important lessons in technical communication: (a) Early graph; and (b) 'Higher is better & Red is us'. PDL, also, gave me the opportunity to interact with industry experts at least once a year. It was in one such meeting that John Wilkes introduced me to Kent Beck's, simple yet compelling, *How to Get a Paper Accepted at OOPSLA*. I have benefitted greatly by my many interactions with CMU's faculty, including those outside the PDL. Alyosha Efros, taught me, by example, to stay simple and work hard. He also challenged me to make presentations on systems research fun by using only graphics and animations. Do you really need to guess his field

of research? Mor Harchol-Balter and Kayvon Fatahalian gave me invaluable feedback on my job application material and job talk.

My research was supported by the National Science Foundation, CMU, and an IBM Ph.D. Fellowship award. Many thanks to all for making my research possible.

My parents, Shantha and S. D. Phanishayee, are my most influential teachers, and the primary reason I have made it this far. They are inspirational role models, always prodding me to "do something earth shattering" and to "change the world for the better". Their support and unconditional love has helped me weather many a storm. Laura is my constant source of enthusiasm, creativity, and love; all this while plotting to take over the world with compelling human-like animated minions.

Thank you everyone, named and unnamed, who helped me get here.

Contents

List of Figures

List of Tables

List of Theorems

List of listings

Chapter 1

Introduction

Systems that deal with large amounts of data have traditionally used relational database management systems (RDBMS) [30] to store structured data and to easily access and manipulate this data using high-level languages like SQL [26]. Recently, however, both large and small web services have moved towards simpler solutions. To meet their needs for cost-effective high performance data access and analytics, many sites use simpler data model "NoSQL" systems. These systems store and retrieve data only by a primary key, do not provide the combined ACID guarantees (Atomicity, Consistency, Isolation, Durability) [45, 96], and do not require the complex querying and management functionality offered by an RDBMS [39].

High-performance distributed key-value (KV) storage systems—one popular type of NoSQL solution—are growing in both size and importance; they now are critical parts of major Internet services such as Amazon (Dynamo [39]), LinkedIn (Voldemort [85]), and Twitter & Facebook (`memcached`[72]). The workloads these systems support share several characteristics [8, 10, 18]: they are I/O, not computation, intensive, requiring random access over tens of terabytes to petabytes of datasets; they are massively parallel, with thousands of concurrent, mostly-independent operations; their high load requires large clusters consisting of tens of thousands of individual servers to support them; and the size of objects stored is typically small, e.g., 1 KB values for thumbnail images, 100s of bytes for wall posts, Twitter messages, etc. Unfortunately, large clusters have more frequent failures [44], and a challenge faced by many large scale distributed KV systems is maintaining high *performance-availability* [42, 86]: they must continue to offer high performance even in the presence of failures.

The first research question this dissertation addresses is: *How should we design a cluster-based key-value store that is fault tolerant, achieves high performance and availability, and offers strong data consistency?* Strong consistency asserts that (a) operations to query and update individual key-value pairs are executed in some sequential order, and, (b) the effects of update operations are necessarily reflected in results returned by subsequent query operations [4, 99, 102, 104]. We present *Ouroboros* [1], a new generalization of chain-based replication for fault tolerance that offers provably strong consistency (per-key linearizability [53]). Ouroboros is designed to allow node additions to any part of the replica chain, and to minimize blocking during node additions and deletions for high performance. Ouroboros uses log-based replication for high performance on storage devices with slow random write performance, such as Flash. Chapter 3 presents a formal proof of correctness of Ouroboros. We describe Ouroboros in the context of the FAWN-KV, a new KV system that uses Ouroboros on a consistent hashing ring. We use the FAWN ("Fast Array of Wimpy Nodes") hardware

[1] Ouroboros is an ancient symbol depicting a serpent or dragon eating its own tail. [3]

Configuration	Hardware			Application Needs				
	Mem	***Disk***	***Flash***	Durability		Fault Tolerance	Consistency	
				Cache	***Store***		***Strong***	***Eventual or Weak***
memcached	●			●				
repcached	●			●		●		●
memcachedb		●			●			
Dynamo		●			●	●		●
Hibari		●			●	●	●	

Table 1.1: Today's KV space consists of many point solutions. This table shows some examples of existing KV systems.

architecture as a motivating hardware configuration to evaluate FAWN-KV. We find that Ouroboros and FAWN-KV together achieve our goals of high performance, availability, and consistency.

FAWN-KV is, still, only a specific KV solution designed to offer strong data consistency and optimized for clusters that have storage devices with slow random write performance. The current diversity in storage technology, cluster hardware, and application requirements have resulted in a plethora of key-value systems to choose from today, with no one system meeting the needs of all applications. Consider first the diversity in storage technology: Amazon uses disk-based KV stores [39] and has recently started offering a Flash-based alternative [16]; Google maintains its entire index in tens to hundreds of terabytes of DRAM [37]; and Facebook caches over 90% of their data in massive farms of memory caches with mean cache hit rates of around 90% [18, 22]. These storage choices offer dramatically different tradeoffs between sequential and random read & write performance, durability, and power consumption. Next, there is diversity in cluster hardware; many applications use conventional, "brawny" platforms to provide computational headroom to contain software development costs [54]; others may use racks of "wimpy node" hardware such as that becoming available from startups like SeaMicro and Calxeda to minimize their energy consumption [94]. Finally, there are also trade-offs to consider when it comes to application requirements: some applications or operations demand synchronous, durable replication for high availability; for others, the cost of such safety is orders of magnitude too expensive, making it impossible to meet latency or throughput requirements.

These system and application requirements sit on a multi-dimensional continuum, with the breadth of NoSQL systems testifying to the value of finding a design and implementation well matched to one's requirements. Table 1.1 shows some examples of existing KV systems and the hardware & application requirements they target. Unfortunately, this demand places system designers in a bind: Do they run multiple stores, each operating at maximum efficiency, or do they optimize instead for system complexity by avoiding the need for multiple codebases, vendors, and so on? We argue that placing systems designers in this bind is unreasonable and unnecessary.

The second, and final, research question this dissertation addresses is therefore: *Is it possible for a key-value architecture to be easily configured to support many points along the KV system design continuum, from weakly-consistent, non-replicated caches [72] to strongly-consistent, durable disk-backed key-value stores [48]?* We present a generalization of chain-based replication, Ouroboros+, which extends Ouroboros to effectively support a wide

Contents	***Location***
FAWN-KV – the design and implementation of a high-performance strongly-consistent cluster key-value store	Chapter 2
Proof of correctness of the Ouroboros protocols described in § 2	Chapter 3
Empirical evaluation of FAWN-KV	Chapter 4
Flex-KV – the design and evaluation of a flexible key-value system	Chapter 5 Chapter 6
Related Work	Chapter 7

Table 1.2: A broad outline for this thesis document.

range of application requirements by (a) selecting from different update mechanisms between replicas, and, (b) selecting a query node in a replica chain. We describe Flex-KV, a flexible key-value storage system, which uses Ouroboros+ with different datastores that expose a common storage interface to form homogeneous or heterogeneous replica chains. Flex-KV can support DRAM, Flash, and disk-based storage, can act as an unreliable cache or a durable store, and operate consistently or inconsistently. The value of such a system goes beyond ease-of-use: While exploring these dimensions of durability, consistency, and availability, we find new choices for system designs, such as a cache-consistent memcached, that offer some applications a better balance of performance and cost than was previously available.

Chapter 2

FAWN-KV: The FAWN Key-Value System

We begin our search for a key-value storage system that works well on a variety of cluster hardware choices and sizes by designing the FAWN-KV distributed key-value storage system[1]. FAWN-KV is designed to work on cluster hardware ranging from small clusters with fast processors and large disk-based storage per node, to large clusters with slow processors and smaller Flash based storage per node. FAWN-KV is designed with the following four principles, with the goal of supporting the three key properties of fault tolerance, high performance, and generality:

1. *Sequential writes for high performance and generality.* Appending sequentially to storage can support the highest throughput available on a variety of storage devices, from

[1]In contrast, the focus of the thesis work of Vijay Vasudevan, who also worked on FAWN, is on the energy efficiency and on batching work to effectively use newer SSD storage devices.

memory, to Flash, to disk drives, and even older technologies such as tape drives or emerging technologies such as phase-change memory. Small random writes perform poorly on many of these devices and hence should be avoided. All update operations in FAWN-KV are append-only.

2. *Replication for fault tolerance*: FAWN-KV tolerates node failures by using replication [20, 93]. But replication brings with it the issue of data consistency. For simplicity, FAWN-KV supports only the "highest common denominator" of strong consistency [2]; Section 5 addresses support for flexible consistency models.

3. *Protocols with minimal blocking for high performance under churn*: As large clusters have more frequent failures, a cluster-based key-value system must handle churn in the system. FAWN-KV uses a new variant of chain replication on a consistent hashing ring, called **Ouroboros**. Ouroboros is designed to minimize blocking during node additions and deletions while maintaining provably strong data consistency. Furthermore, node additions and removals involve splitting/merging and transferring datastores; these operations avoid time consuming random writes to the datastore and instead use only single-pass sequential scans and sequential writes.

4. *Load balancing for high performance*: FAWN-KV partitions data storage and request handling responsibilities among backend nodes. FAWN-KV also limits the global work done in the system on a node addition and removal, while avoiding disproportionately increasing the load on a single node.

[2]We formally define and prove the guarantees offered by FAWN-KV in Section 3

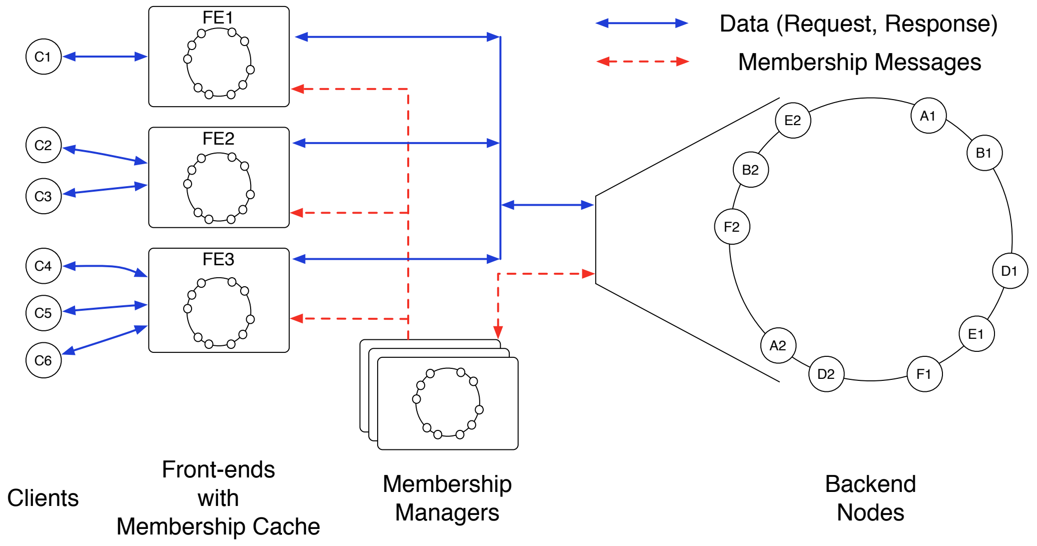

Figure 2.1: FAWN-KV Architecture.

In the following sections we describe the design, implementation, and evaluation of FAWN-KV using FAWN [17] as a motivating hardware configuration. FAWN-KV not only targets Flash-based clusters, but is generic enough to be applied to other cluster designs, such as those with hard disk drives; FAWN is only one design point, optimized for energy efficiency, in that space. We provide a brief overview of the system, followed by a detailed description of the FAWN-KV distributed system, including partitoning, replication, and consistency in the face of node arrivals and failures.

2.1 Design Overview Of The FAWN-KV System

Figure 2.1 depicts an overview of the entire FAWN system. Client requests enter the system at one of several *front-ends*. FAWN-KV balances load by partitioning data storage and request handling responsibilities among a large number of back-end storage nodes. The front-end nodes forward the request to the *back-end* FAWN-KV node responsible for serving that particular key. The back-end node serves the request from its FAWN-DS datastore and returns the result to the front-end (which in turn replies to the client). Writes proceed similarly.

FAWN back-end nodes use Flash storage. Flash provides fast random reads ($\ll 1$ ms) [75, 84], but small random writes on Flash are very expensive [77]. This performance problem motivates the need for log-structured techniques to write data to flash [58, 60, 76, 77, 89]. FAWN-DS is a simple log-structured data store for key-value pairs. The data associated with each virtual ID is stored on flash using FAWN-DS.

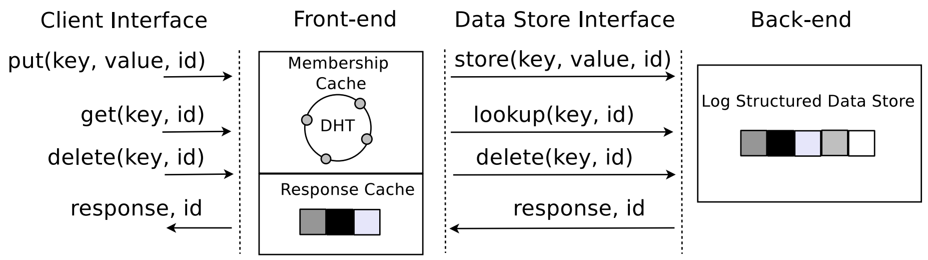

Figure 2.2: FAWN-KV Interfaces—Front-ends route requests and cache responses. Back-ends use FAWN-DS to store key-value pairs.

Figure 2.2 depicts FAWN-KV request processing. Client applications send requests to front-ends using a standard put/get interface. Clients link against a front-end library and send requests using Thrift RPC [15]. Front-ends send the request to the back-end node that owns the key space for the request. The back-end node, with associated queues and threads to make parallel use of Flash using a staged execution model similar to SEDA [105, 106], satisfies the request using its FAWN-DS and replies to the front-ends.

FAWN-KV caches data using a two-level cache hierarchy. Back-end nodes implicitly cache recently accessed data in their filesystem buffer cache. The FAWN front-end maintains a small, high-speed query cache that helps reduce latency and ensures that if the load becomes skewed to only one or a few keys, those keys are served by a fast cache instead of all hitting a single back-end node.

2.2 Consistent Hashing: Mapping key ranges to nodes

A typical FAWN cluster will have several front-ends and many back-ends. How should storage and request processing responsibilities be partitioned among the backend nodes?

If we have a cluster of n back-end nodes (0 to $[n-1]$), a naive approach might store the key-value pair `(k, v)` on back-end node $[h(k) \mod n]$, where `h()` is a hash function. A good hash function implies $[h(k) \mod n]$ is uniform across $0, \ldots, (n-1)$ for a reasonable distribution of keys, thus spreading the load evenly across the back-end nodes. The problem with this scheme is key redistribution due to a node addition to the cluster: $n/(n+1)$, nearly all the keys, end-up being remapped affecting all the nodes in the cluster.

Consistent hashing [59], popularized by the Chord Distributed Hash Table [95], solves this problem. Like the previous scheme, consistent hashing spreads data evenly across the cluster. But in contrast to the naive scheme described earlier, on a node addition, consistent hashing requires only the data that resides on that node, a relatively small amount of data, to be moved there. By using consistent hashing, only K/n keys need to be remapped on average, where K is the number of keys, and n is the number of back-end nodes.

In FAWN-KV the key space is represented as a ring. The large number of back-end storage nodes are organized into this ring using consistent hashing. As in systems such as Chord, keys are mapped to the node that follows the key in the ring (its *successor*). To balance load and reduce failover times, each *physical node* joins the ring as a small number (V) of *virtual nodes*, each virtual node representing a *virtual ID* ("*VID* ") in the ring space. Each physical node is thus responsible for V different key ranges. FAWN-KV does not use DHT routing—instead, front-ends maintain the entire node membership list and directly forward queries to the back-end node that contains a particular data item.

Each front-end node uses the backend *VID* membership list and handles queries for a large contiguous chunk of the key space; in other words, the circular key space is divided into pie-wedges, each owned by a single front-end. A front-end receiving queries for keys outside of its range forwards the queries to the appropriate front-end node. This design either requires clients to be roughly aware of the front-end mapping, or doubles the traffic that front-ends must handle, but it permits front ends to cache values without a cache consistency protocol.

A single manager node is responsible for maintaining the membership and key space allocations for both the front-ends & back-ends; this node can be replicated using a small Paxos cluster [23, 55, 63, 112] to mask single management node failures.

When a back-end node joins, it sends its request to the manager node. On receiving the *VIDs* it represents from the manager, each of its virtual nodes joins the ring, one *VID* at a time. The back-end virtual node identifier (and thus, what keys it is responsible for) is a deterministic function of the back-end node ID. The manager node notifies the front-end nodes of a change in ring membership.

The FAWN-KV ring uses a 160-bit circular ID space for *VIDs* and keys. Virtual IDs are hashed identifiers derived from the node's address. Each *VID owns* the items for which it is the item's successor in the ring space (the node immediately clockwise in the ring). As an example, consider the cluster depicted in Figure 2.3 with five physical nodes, each of which has two *VIDs*. The physical node A appears as *VIDs* $A1$ and $A2$, each with its own 160-bit identifiers. *VID* $A1$ owns key range $R1$, *VID* $B1$ owns range $R2$, and so on.

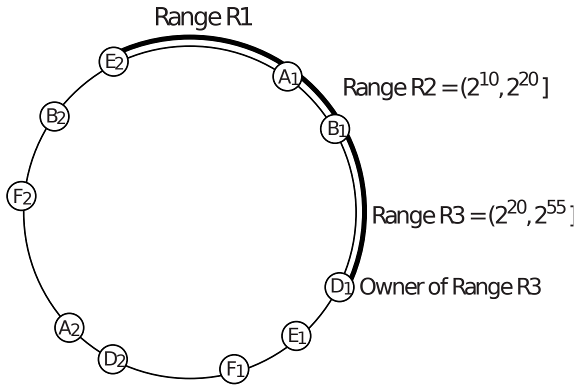

Figure 2.3: Consistent Hashing with 5 physical nodes and 2 virtual IDs each.

Consistent hashing provides incremental scalability without global data movement: adding a new *VID* moves keys only at the successor of the *VID* being added. For example, in Figure 2.4 adding a virtual node with identifier $C1$ causes the old $R3$ range (from 2^{20} to 2^{55}) to be split, making $C1$ the owner of the new $R3$ range (from 2^{20} to 2^{35}) and $D1$ the owner of $R4$ (from 2^{35} to 2^{55}). $C1$ gets the new $R3$ range from node $D1$ that initially owned it. FAWN-KV uses single-pass sequential scans and sequential writes to datastores to handle such changes efficiently.

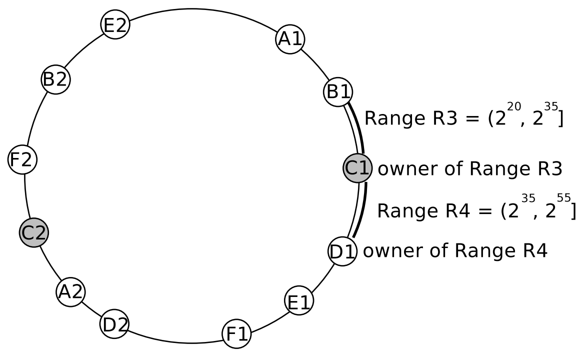

Figure 2.4: A new node C added to the cluster is represented by two virtual nodes $C1$ and $C2$. The new virtual node $C1$ when added to the ring results in movement of data for the new $R3$ range from its successor $D1$.

2.3 Replication and Consistency

FAWN-KV offers a configurable replication factor for fault tolerance. Items are stored at their successor in the ring space and at the $R-1$ following virtual IDs.

FAWN-KV uses a new generalization of chain replication [102], called Ouroboros, to provide strong consistency on a per-key basis [4, 99, 104]. Updates are sent to the head of the chain, passed along to each member of the chain via a TCP connection between the nodes, and queries are sent to the tail of the chain. Traditional chain replication only allows new nodes to join as tail nodes in a replica chain. Our variant of chain replication allows node additions, with minimal locking, to *any* position in a replica chain while ensuring strong data consistency (per-key linearizability [53]).

One way to implement chain replication on a consistent hashing ring is shown in Figure 2.5. In this approach each node is part of one chain. The first important issue to address in this approach is: how does one support the addition of one physical node? Here, one

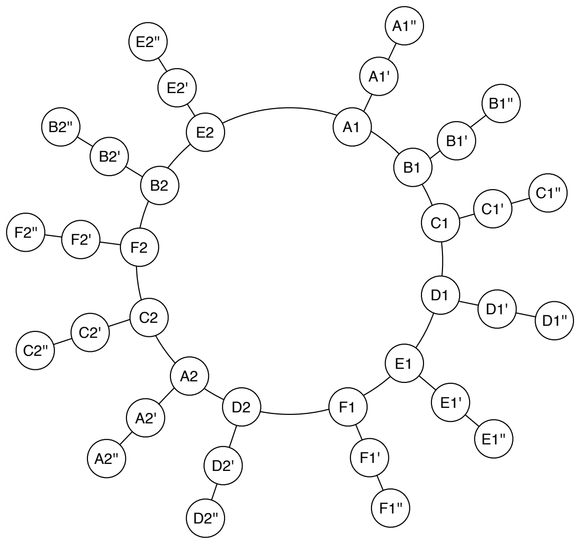

Figure 2.5: Non-Overlapping Chains on the Ring.

physical node should have a multiple of R virtual nodes. This avoids the naive approach of having $R - 1$ other physical nodes join for every new physical node. The next issue to address is: where do these virtual nodes join in the chain? The naive approach would be to determine the location of the new virtual node on the ring, and have it join the tail of the chain at that location. This approach risks increasing the load imbalance in the system; if N is the total number of physical nodes, only N/R physical nodes are tails and serve queries.

FAWN-KV uses a simple, yet effective technique, to solve this problem. By mapping chains to the consistent hashing ring, each virtual ID in FAWN-KV is part of R different chains: it is the "tail" for one chain, a "mid" node in $R - 2$ chains, and the "head" for one. This replica selection strategy is similar to that used in systems like CFS [32, 33], DHash++ [34], Dynamo [39], and PAST & Pastry [41, 90, 91]. However, none of these systems use chain based replication and they provide weaker consistency guarantees. Figure 2.6 depicts a ring with six physical nodes, where each has two virtual IDs ($V = 2$), using

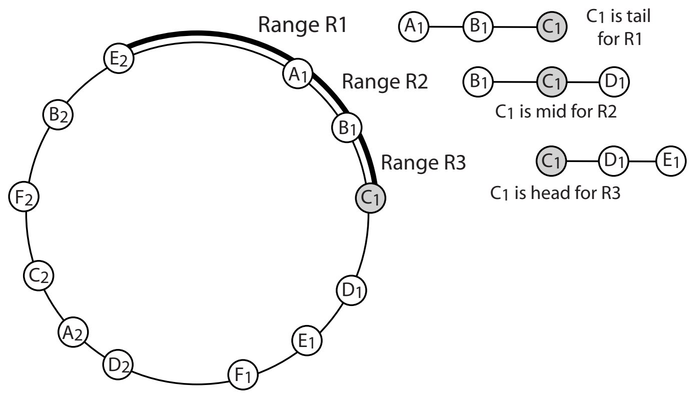

Figure 2.6: Overlapping Chains in the Ring – Each node in the consistent hashing ring is part of $R = 3$ chains.

a replication factor of three ($R = 3$). In this figure, node $C1$ is thus the tail for range $R1$, mid for range $R2$, and tail for range $R3$.

Figure 2.7 shows a put request for an item in range $R1$. The front-end routes the put to the key's successor, *VID* $A1$, which is the head of the replica chain for this range. After storing the value in its datastore, $A1$ forwards this request to $B1$, which similarly stores the value and forwards the request to the tail, $C1$. After storing the value, $C1$ sends the put response back to the front-end, and sends an acknowledgment back up the chain indicating that the response was handled properly.

For reliability, nodes buffer put requests until they receive the acknowledgment. Because puts are written to an append-only log in FAWN-DS and are sent in-order along the chain, this operation is simple: nodes maintain a pointer to the last unacknowledged put in their datastore, and increment it when they receive an acknowledgment. By using a purely log structured datastore, chain replication with FAWN-KV becomes simply a process of streaming the growing datastore from node to node.

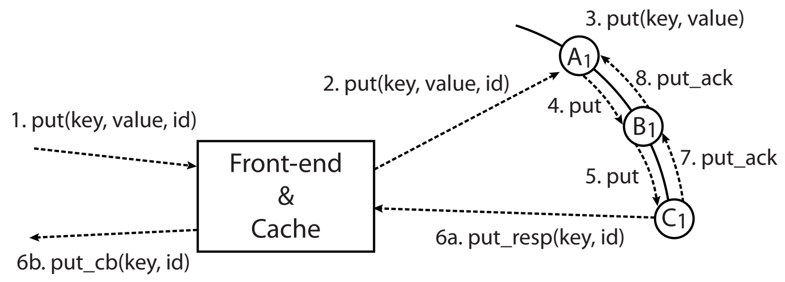

Figure 2.7: Lifecycle of a put with chain replication—puts go to the head and are propagated through the chain. Gets go directly to the tail.

Gets proceed as in chain replication. Figure 2.8 shows a get request for an item in Range $R1$—the front-end directly routes the get to the tail of the chain for range $R1$, node $C1$, which responds to the request. Chain replication ensures that any update seen by the tail has also been applied by other replicas in the chain.

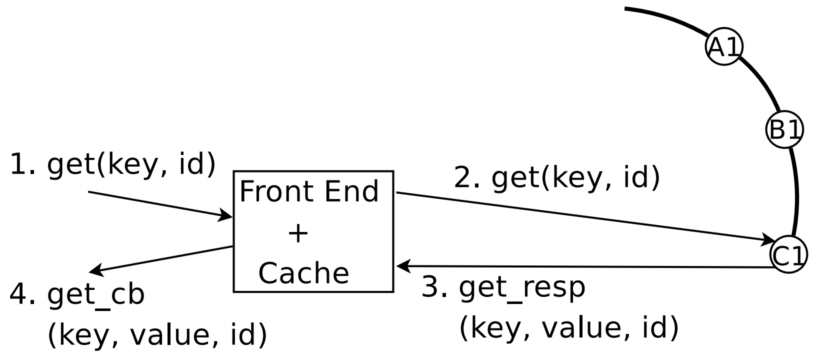

Figure 2.8: Gets go directly to the tail of the chain as in Chain Replication.

2.4 Support for multiple front ends

As mentioned earlier, each front-end node uses the backend *VID* membership list and handles queries for a large contiguous chunk of the key space; in other words, the circular key space is divided into pie-wedges, each owned by a single front-end. This permits front ends to cache values without a cache consistency protocol.

Each back-end replica chain is associated with a monotonically increasing *chain-view-id*. Any change to the back-end membership is handled by the manger node appropriately, and the new membership view, along with the chain-view-id, is communicated to the appropriate back-ends and to all the front-ends. The chain-view-id is incremented only when the head or a tail of a replica chain changes.

Similarly, any change to the front-end membership is handled by the manger node appropriately, and the new front-end membership view, along with the *fe-view-id*, is communicated to all the front-end and back-end nodes. The combined membership ID, *fe-view-id.chain-view-id*, is stamped in all requests sent from the front-ends and all responses sent by the back-ends. Requests or responses that do not match the local membership ID are dropped. Also, updates directed at back-end nodes that are not the head, and queries directed at backend nodes that are not the tail, are dropped.

With multiple front ends, tail nodes wait for an acknowledgment to put responses sent to the front-end, before propagating their acknowledgment for the put back up the chain. In the case of a front-end failover, the tail resends all the put responses that were not acknowledged by the previous front-end. This ensures that clients always receive a response for their put request.

2.5 Node Joins In Ouroboros

When a node joins a FAWN-KV ring:

1. The new virtual node causes one key range to split into two.
2. The new virtual node must receive a copy of the R ranges of data it should now hold, one as a primary and $R - 1$ as a replica.
3. The front-end must begin treating the new virtual node as a head or tail for requests in the appropriate key ranges. The replication factor goes up by one (to $R + 1$).
4. Virtual nodes down the chain may free space used by key ranges they are no longer responsible for. The replication factor goes down by one and is restored to R.

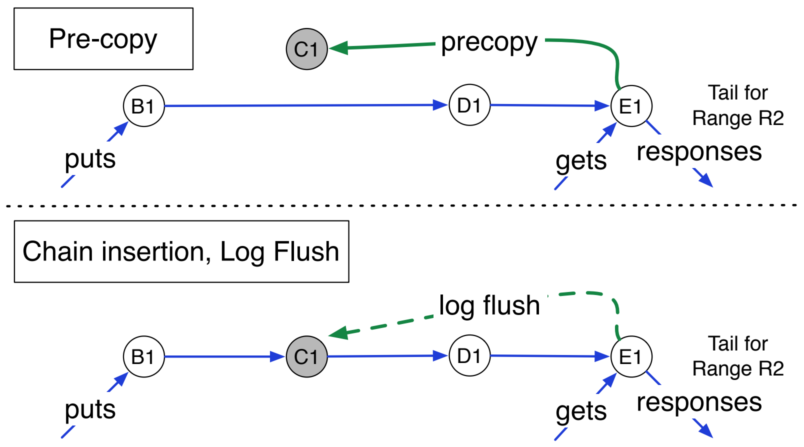

Figure 2.9: Phases of the join protocol on node arrival.

The first step, key range splitting, can occur concurrently with the rest (split and data transmission can overlap). For clarity, we describe the rest of this process as if they were separate.

Each virtual node must become a working member of R chains. For each of these chains, the node must receive a consistent copy of the datastore file corresponding to the key range. One way to achieve this is by blocking all updates and queries to all the chains where the new replica is being added, and to resume processing requests once the new replica has received a consistent copy of the datastore file. This is similar to the stop and transfer techniques used in Internet Suspend & Resume [62] and μ-Denali [107].

The first important issue to address in the above approach is availability: This method can make the replica chains unavailable for long durations of time; time proportional to the size of the datastore. The duration of unavailability can be minimized by repeatedly copying new data over, not blocking requests to the chain, and blocking all requests at the "end" to ensure that the new replica has a consistent view of the datastore. This is a technique similar to that used in distributed databases and live migration of virtual machines [29, 67, 80], and still results in a period of unavailability due to request blocking. The second important issue to address is: how does one decide that the new replica has a consistent copy of the datastore? A naive approach approach could involve a two phase commit with all replicas agreeing that the copy at the new replica is consistent with their copy of the datastore. This technique, once again, results in a period of unavailability for updates and queries.

Our protocol, described below, is a non-blocking protocol for node additions and ensures that if the node fails during the data copy operation, the existing replicas are unaffected. We illustrate this process in Figure 2.9 where node $C1$ joins as a new middle replica for range $R2$.

Phase 1: Datastore pre-copy. Before any ring membership changes occur, the current tail for the range (*VID* $E1$) begins sending the new node $C1$ a copy of the datastore log file. The pre-copy phase is the most time-consuming part of the join, potentially requiring hundreds of seconds. At the end of this phase, $C1$ has a copy of the log that contains all records committed to the tail. If the replication factor is R, pre-copy takes place in R replica chains, as depicted in Figure 2.10 for $R = 3$. Pre-copy does not change the structure of the replica chain; a failure of a non-tail node does not affect pre-copy, and the failure of any replica (non-tail, tail, new node) does not affect the membership state maintained at the manager.

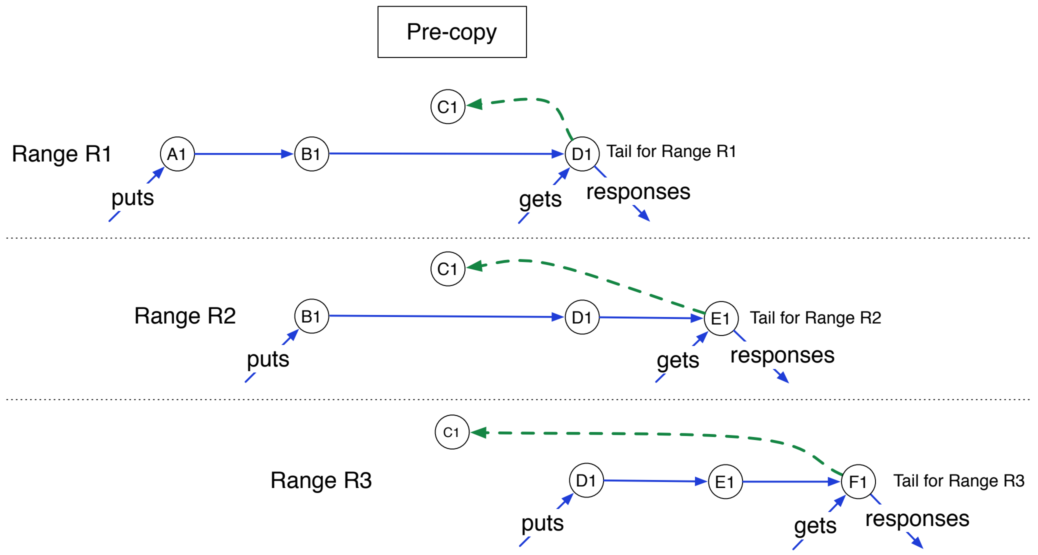

Figure 2.10: Phase 1 of Join: Datastore pre-copy. For every chain it is a part of, a joining node, $C1$ in this case, has the datastore streamed to it from the tails of those ranges.

Phase 2: Chain insertion, log flush and play-forward. After $C1$'s pre-copy phase has completed, the front-end sends a *Flush message* through the chain. This message plays two roles: first, it updates each node's neighbor state to add $C1$ to the chain; second, it ensures that any in-flight updates (sent after the pre-copy phase completed) are flushed to $C1$.

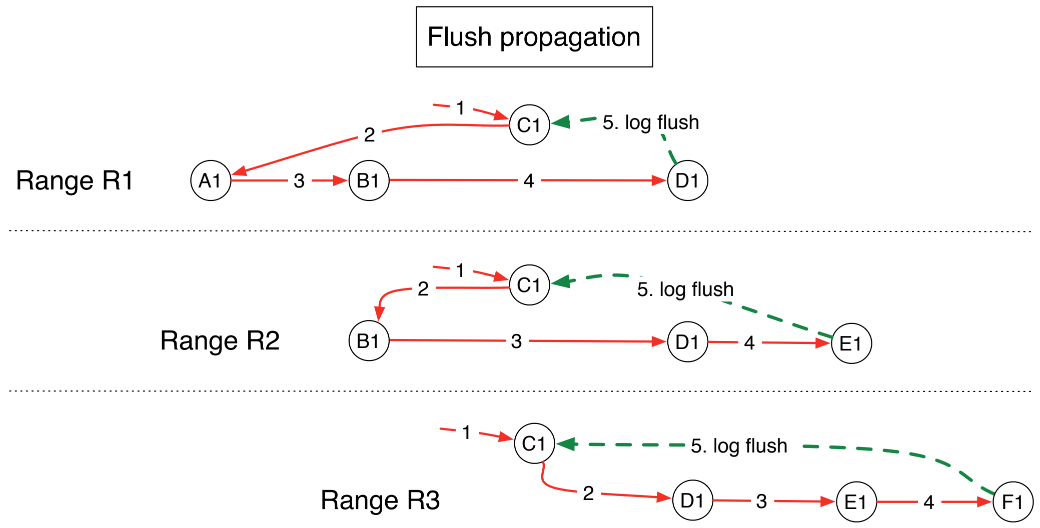

Figure 2.11: Phase 2 of Join: Flush. The Flush protocol ensures that a new node can join R chains at appropriate locations while maintaining strong consistency, and does so with minimal blocking. The Flush message for the case where the new node joins as the head, Range $R3$ in the example above, also has a flag indicating that $D1$, $E1$, and $F1$ can *split* the range from $B1$–$D1$ into $B1$–$C1$ and $C1$–$D1$.

More specifically, in Figure 2.9, this message propagates to $C1$ and then in-order through $B1$, $D1$, and $E1$, and finally back to $C1$ after a log flush from $E1$. Nodes $B1$, $C1$, $D1$, and $E1$ update their neighbor list, and nodes in the current chain forward the message to their successor in the chain. Updates arriving at $B1$ after the reception of the Flush message at the head of the chain now begin streaming to $C1$, and $C1$ relays them properly to $D1$. At this point, $B1$, $D1$, and $E1$ have correct, consistent views of the datastore, but $C1$ may not: A small amount of time passed between the time that the pre-copy finished and when $C1$ was inserted into the chain. To cope with this, $C1$ logs updates from $B1$ in a temporary datastore, *not* the actual datastore file for range $R2$, and does not update its in-memory hash table. During this phase, $C1$ is not yet a valid replica.

All put requests sent to $B1$ after it received the Flush message are replicated at $B1$, $C1$, $D1$, and $E1$. On receiving the Flush message, $E1$ pushes all entries that might have arrived in the time after $C1$ received the log copy and before $C1$ was inserted in the chain. $C1$ adds these entries to the $R2$ datastore. These messages have a special flag indicating that they are part of the log flush operation. At the end of this process, $E1$ sends the Flush message back to $C1$, confirming that all in-flight entries have been flushed. $C1$ then merges (appends) the temporary log to the end of the $R2$ datastore, updating its in-memory hash table as it does so. The node briefly locks the temporary log at the end of the merge to flush these in-flight writes.

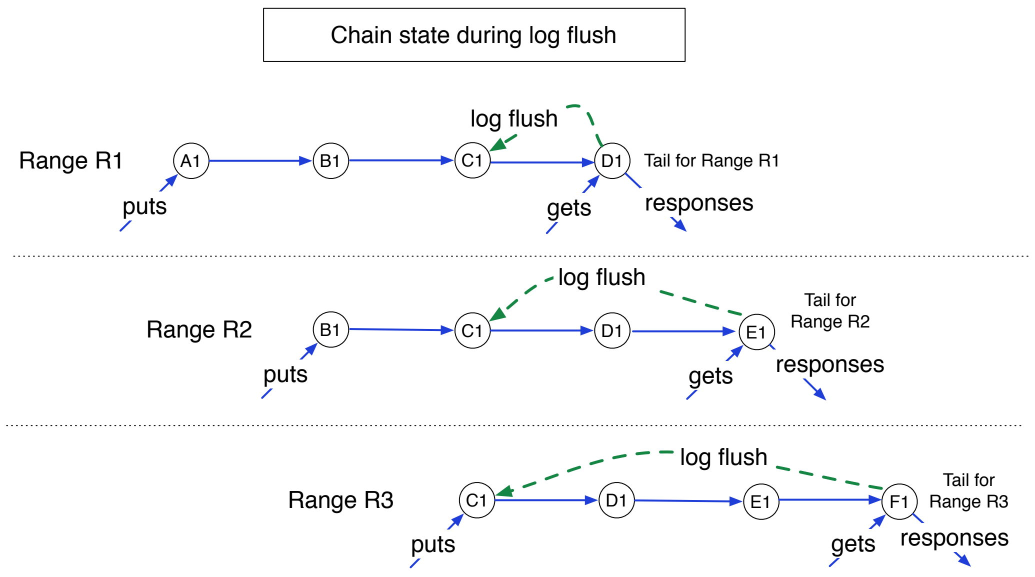

Figure 2.12: When Flush is in progress, puts generated after the flush message was sent to the head of the chain, also go through the new node $C1$. These puts are stored at a temporary datastore for each range at $C1$, and can be applied (appended) to the datastore once the log flush for these ranges complete.

After phase 2, C1 is a functioning member of the chain with a fully consistent copy of the datastore. This process occurs R times for the new virtual node—e.g., if $R = 3$, it must join as a new head, a new mid, and a new tail for one chain. Figures 2.10, 2.11, and 2.12 show the process for $C1$ joining three chains: tail for range $R1$, mid for range $R2$, head for range $R3$. Figure 2.11 shows the propagation of the Flush message in the three different replica chains that node $C1$ is joining. For the case where the new node joins as the head, Range $R3$, the range from $B1$–$D1$ is split into two ranges at all the replicas, $B1$–$C1$ and $C1$–$D1$. Figure 2.12 shows the propagation of puts and gets *during* Flush. Pre-copy, log flush, split, and merge all involve sequential scans of the data store and append only update operations.

Joining as a head or tail: The process for joining as a head or tail node is identical to that of a new mid. When a node joins as a new head node, new updates arrive at it before propagating through the replica chain. But a new node never joins as a tail node directly during node additions. To join as a tail, a node joins before the current tail. It does not serve get requests until the replication factor for the chain is restored (end of Phase 3).

Although the manager always has the correct view of the head and the tail of the chain being flushed, it can never be certain of the precise structure of the chain *during* Flush. The manager is always certain of the structure of the chain before a node addition, and after flush completes; during Flush, however, the manager does not know the "current" location of Flush in the replica chain and hence does not know the state of successor and predecessor links at individual nodes in the replica chain. A node failure during Flush results in the master locking the chain state and repairing the chain (node removal). The new replica's addition to the chain is redone once the replica chain is fully repaired. Phase 1, Pre-copy, minimizes the amount of time that the master is uncertain of the precise structure of the chain.

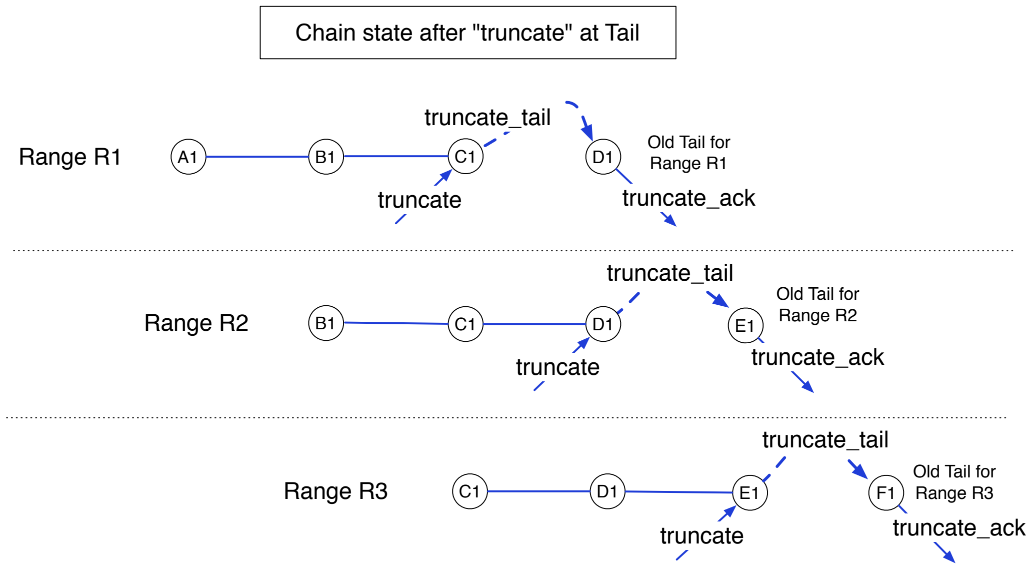

Figure 2.13: Phase 3 of Join: Truncate restores the replication factor back to R by discarding the tail replica as a node in the replica chain.

Phase 3: Truncate. At the end of Phase 2 (Flush), the replication factor of each of the replica chains goes up by one ($R \rightarrow (R + 1)$). Figure 2.13 shows the final phase of the protocol that restores the replication factor by truncating the chain and discarding the tail replica.

Putting it all together. Figure 2.14 shows the detailed timing digram of the join protocol, with the contents of messages exchanged between the Manager and the joining node where appropriate. The new node (Q) sends a join request to the manager node, and for each of the virtual nodes it represents goes through the three phases described so far.

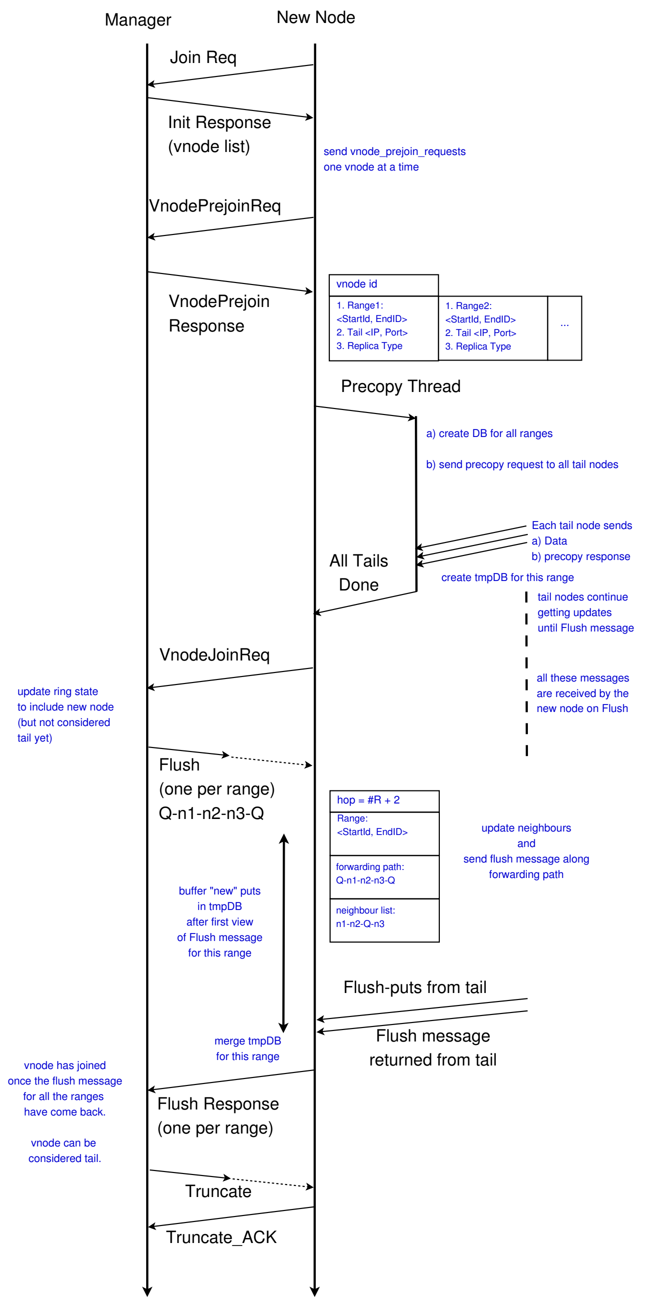

Figure 2.14: The detailed join protocol showing the order of messages exchanged between the Manager and the joining node.

2.6 Node Removals In Ouroboros

The effects of a voluntary or involuntary (failure-triggered) leave are similar to those of a join, except that the replicas must *merge* the key range that the node owned. As above, the nodes must add a new replica into each of the R chains that the departing node was a member of.

This process is shown in Figure 2.15 where node $C1$ leaves a chain of three replicas where it is the mid replica for range $R2$. Node $D1$ takes over as the tail for this range for gets, until $E1$ catches up with $D1$.

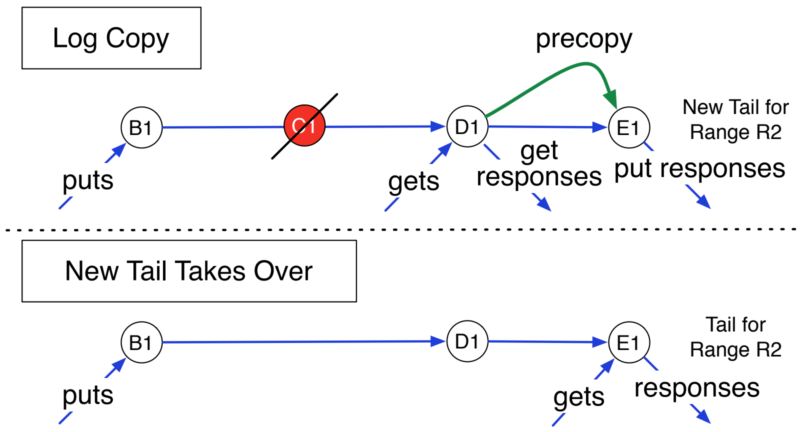

Figure 2.15: A simplified view of the process of restoring the replication factor of a chain when a node fails.

Following are the steps to handle a node departure:

1. **Detect that a node has failed.** Nodes are assumed to be fail-stop [92]. A recent study indicated a MTTF for nodes to be between 4 to 5 months [44]; a 100 node cluster would experience 1 node failure every day, even if it is a transient failure. The manager node exchanges heartbeat messages with its back-end nodes every t_{hb} sec-

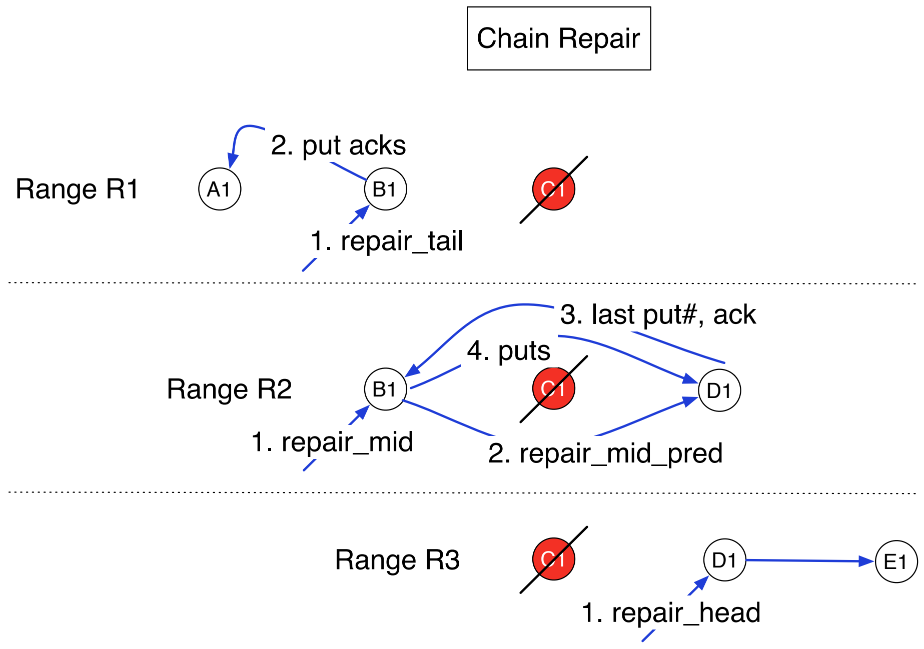

Figure 2.16: When a node leaves, all chains that it was a part of are repaired to route around the failure. This process also ensures that pending updates (due to the node failure) are propagated down the chain, and pending acks are propagated up the chain aiding in garbage collection.

onds. If a node misses $fd_{threshold}$ heartbeats, the manager considers it to have failed and initiates the Leave protocol. Because the Join protocol does not insert a node into the chain until the majority of log data has been transferred to it, a failure during join results only in an additional period of slow-down, not a loss of redundancy. In addition to assuming fail-stop, we assume that the dominant failure mode is a *node* failure or the failure of a link or switch, but our current design does not cope with a communication failure that prevents one node in a chain from communicating with the next while leaving each able to communicate with the front-ends and managers. We also do not consider problems associated with data corruption in the storage stack [19], and leave it up to interested applications to verifying data integrity during client reads.

2. **Repair the affected replica chains.** Figure 2.16 shows this chain repair process for node $C1$ leaving a system configured with $R = 3$. Apart from routing around the failed node, chain repair ensures that pending updates are propagated down the chain, and garbage collection can be performed by propagating put-acks up the chain. In the case where a tail is lost, e.g. Range $R1$ in Figure 2.16, the new tail ($B1$) sends responses for all pending puts to the front-end.

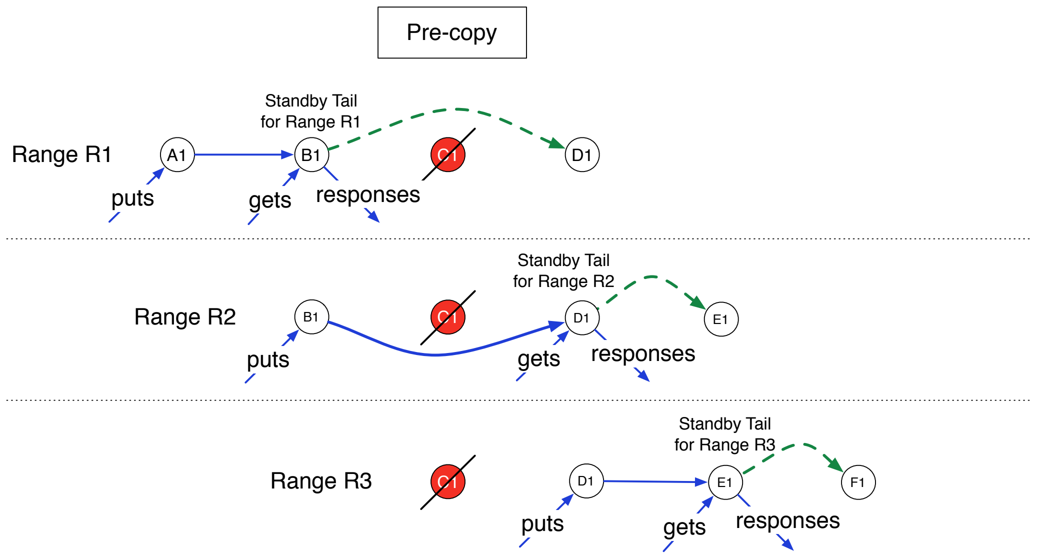

Figure 2.17: To restore the replication factor, a node departure results in new tail nodes for all chains it was part of. The datastore for these ranges is streamed to these new tails.

3. **Prepare new replicas to join the affected chains as tails.** This is done by copying data to the successor of the current tail of each of the R chains. Figure 2.17 shows this pre-copy process for $D1$, $E1$, and $F1$ for ranges $R1$, $R2$, and $R3$ respectively.

4. **Chain insertion, log flush and play-forward.** This is similar to Phase 2 in join, except that the flush message propagates in order through the chain. Figure 2.18 shows this step for $C1$ leaving the system. Until the log flush process completes, an interim tail serves gets. For example, in Figure 2.19, $B1$ is the interim tail for range $R1$ until the log flush to $D1$ completes.

5. **Integrate new replica into ring.** This is when log flush completes and the new tail can start processing get requests.

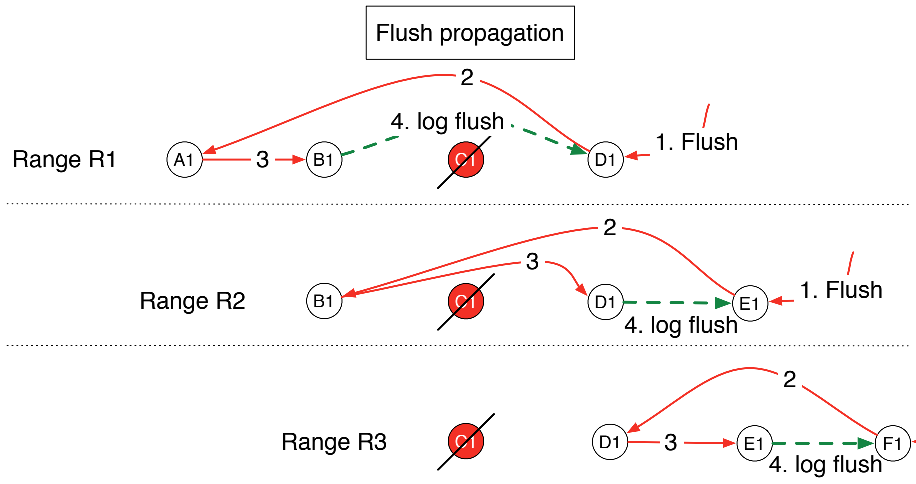

Figure 2.18: The flush protocol ensures all updates sent before the flush message reached the head of the chains are propagated to new tail nodes.

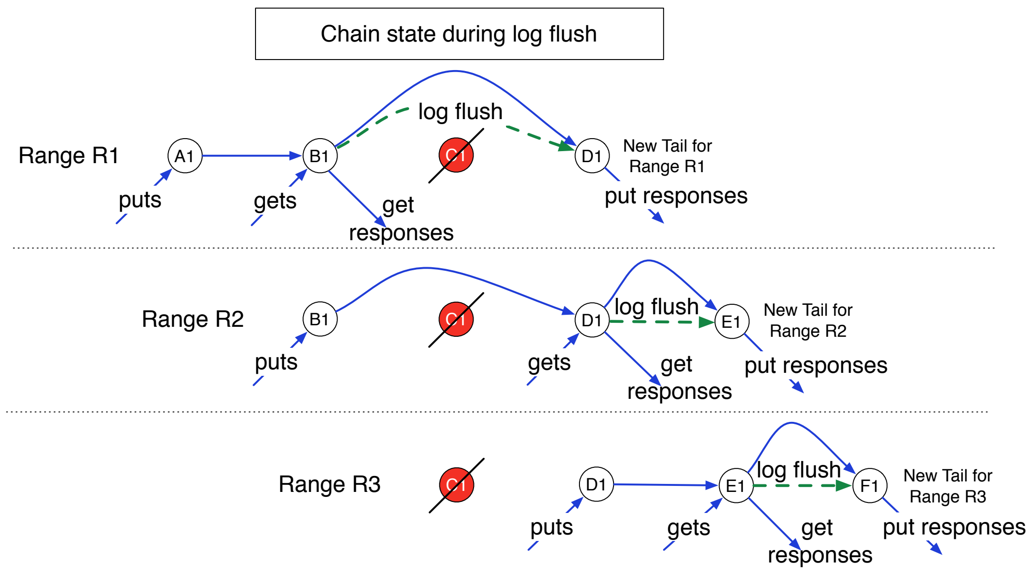

Figure 2.19: During flush, interim tails serve get requests. Puts issued after the flush message reached the head of the chains are propagated to the new tail, but these messages are stored in a temporary store corresponding to these ranges. Once log flush is complete, the temporary store can be merged into the flushed store at the new tails.

Chapter 3

Ouroboros Correctness Proof

We evaluate FAWN-KV in two parts. In this section we formally state the guarantees provided by Ouroboros and we prove that the protocol described in Section 2.5 correctly provides these guarantees. In Section 4, we empirically evaluate FAWN-KV on a 2008-era FAWN cluster built from commodity PCEngine Alix 3c2 devices [81] with CF cards and on a newer FAWN cluster built from Intel Atom D510 processors [56] with Intel SSDs.

3.1 Ouroboros Guarantees

Ouroboros provides the following three guarantees:

Theorem 3.1 (Query Guarantee) *:*

`Query(K)` *either gets the value corresponding to the last successful update for K, or fails.*

Theorem 3.2 (Update Guarantee) *:*

If `Put(K, V)` *succeeds, then this update is registered by the system, unless the replication factor is zero.*

Theorem 3.3 (Replication Guarantee) *:*

Ouroboros maintains a replication factor R for all keys provided that in the face of failures there is at least one working replica and $N >= R$.

3.2 Assumptions

We make the following assumptions for the proof of correctness:

- Fail-stop node failures [92].
- There is a single manager node which does not fail. A Paxos replicated set of manager nodes can be used to emulate a single manager node to relax this assumption. The master event loop in Listing 2 consisting of the manager's role and the data routing role of the front-end.
- There are R replicas in steady state.
- Reliable and in-order message channels: All network channels offer FIFO ordering, maintain message integrity, do not deliver duplicate messages, and have bounded delays. Neighboring nodes of a replica chain are linked by TCP channels. TCP channels also link individual backend nodes to the master and the frontend nodes.

- We consider one chain at a time for the proof. A node is part of multiple chains, but we consider only one chain at at time for the correctness proof. Chains (and key ranges) are independent of one another. The proofs provided here are for key ranges; as keys are part of a keyrange the properties we prove hold true for individual keys too.

3.3 Proof Of Correctness

Definition 3.4 (Successful Update) *An update,* update(k, v), *is successful if the corresponding* `UPDATE_ACK` *message has been received at the master node.*

Definition 3.5 (Segment (π)) *For a given key, the segment, π(x), at a given node 'x' is defined as:*

- $\pi(null) = \emptyset$
- $\pi(x) = x \oplus \pi(x.Successor)$, *where $\oplus$ is sequence concatenation.*

Definition 3.6 (Replica Chain) *A replica chain for a given key is the segment that contains all of the active replica nodes for that key. This definition excludes nodes that have failed, are inactive, or are in the process of joining the system. The first node of the chain is called the head, and the last node the tail. The replica chain is $\pi(head)$. The length of the replica chain is called the replication factor. A chain becomes* irreparable *iff its replication factor drops to* 0.

Definition 3.7 ($<_t$) *For a replica chain containing nodes x and y, if x appears before y in $\pi(x)$ at time t then $x <_t y$*

Invariant 3.8 (**$Tail_k$**) *Every key 'k' has a corresponding query node $Tail_k$, the tail node of the replica chain, unless the replication factor is* 0.

Invariant 3.9 (Tail's successor) *At tail T,* `T.Successor` $= null$

Definition 3.10 (Backend Node State) *A backend node can be in one of four states:* `INACTIVE`, `JOINING`, `ACTIVE`, *and* `FAILED`. *A new node goes from* `INACTIVE` *to* `JOINING` *to* `ACTIVE`.

A node that is pre-copying data while attempting to join a replica chain is in the `INACTIVE` *state. When Flush is in progress, this node is in the* `JOINING` *state. Upon completion of Flush the node transitions to the* `ACTIVE` *state (see* `FLUSH` *case in Lisitng 1).*

Definition 3.11 (History of updates (**H**)**)** *The History of updates at a backend node for a key 'k' consists of all the updates for that key. This includes all the updates that have been acknowledged as well as all the updates with pending acknowledgments not yet processed by the tail; i.e.* $H = H\_acked \cup H\_unacked$

Invariant 3.12 (**$LA <= LU$**) *At a backend node, the Last Acknowledged Update (LA) is always <= Last Update (LU)*

$LA = LU$ to begin with. LA is modified either on receiving an `UPDATE_ACK`, or, in the case of a tail, before sending the `UPDATE_ACK`. LU is only modified at a backend node when an update is received (`UPDATE` case in Lisitng 1). As there can never be an `UPDATE_ACK` for an update not received at a node, $LU >= LA$.

Invariant 3.13 (Tail Has No Unacknowledged Updates) *If the master node considers node T as a tail, then* `T.H_unacked` $= \emptyset$.

The TRUNCATE and CR_TAIL cases in Lisitng 1 are the only cases when a node becomes a tail. In both cases before the node sends TRUNCATE_ACK or CR_TAIL_DONE to the master, it acknowledges all pending updates present in H_unacked.

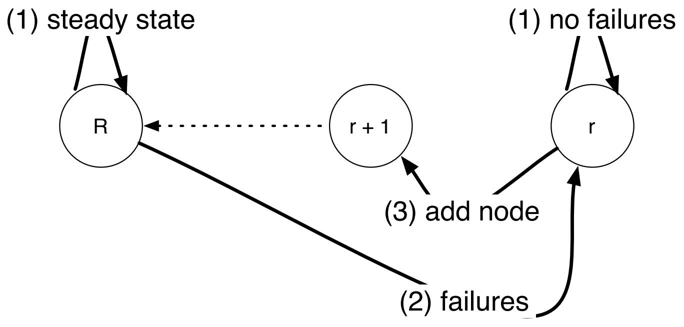

Figure 3.1: Replication factor state machine.

Invariant 3.14 (Chain Property) *For a replica chain containing nodes x and y, if $x <_t y$, then $H_t^y \subseteq H_t^x$*

We prove that the above invariant is maintained by induction on the structure of the replica chain. Figure 3.1 shows the events that modify the replica chain and the effect of these events on the chain replication factor. Let us systematically consider the events that change the structure of the replica chain.

- Case 1: Steady state (no failures or node additions).

  Proof by contradiction. Consider a replica chain containing nodes x and y, where $x <_t y$ and there is no other node between x and y as shown in Figure 3.2 (case 1). Let u be an update that was seen at node y and not at node x. y could have received

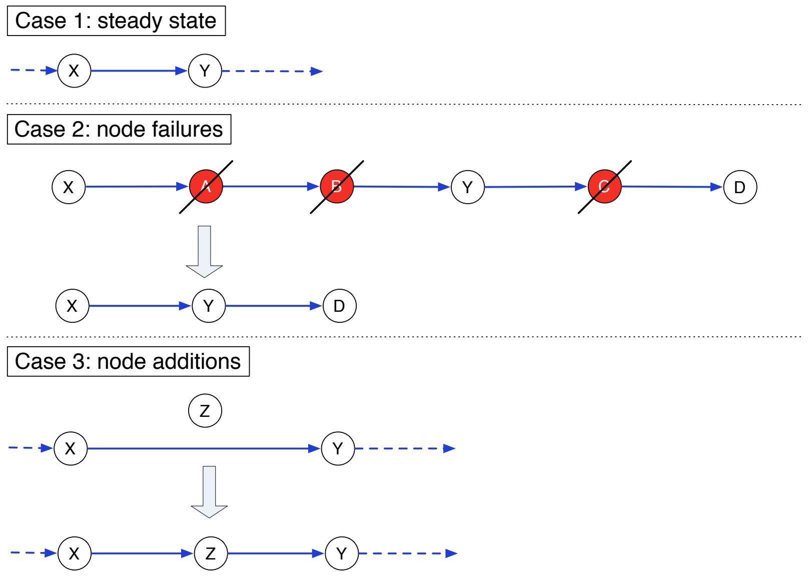

Figure 3.2: Chain invariant cases.

this update only from its predecessor (x), and so this update must also be present at x. Furthermore, as x and y are connected by a reliable FIFO channel, the updates at y are applied in the same order as the updates at x. Hence, there can be no update u at y that is not present at x. Node x, though, might have some updates that it has not yet sent to y. Hence $H_t^y \subseteq H_t^x$. By transitivity, this property holds for any two pairs of nodes in the replica chain irrespective of whether they are adjacent or not.

- Case 2: Node Failures (single or multiple failures).

Consider the case of node failures irrespective of their position in the chain (head, mid, or tail) as shown in Figure 3.2 (case 2). Failures could be single node failures or multiple failures with groups of nodes dispersed across the replica chain failing.

The `CR_HEAD, CR_MID, CR_TAIL` messages change the successor and predecessor links so as to skip over the failed nodes. The replica chain, during and after these operations, is a subsequence, albeit not a contiguous subsequence, of the replica chain before the failures. Because the relative ordering of the nodes in the replica chain has not changed, if $x <_t y$ then $H_t^y \subseteq H_t^x$ (from Case 1).

We can also make the following (stronger) claim based on this observation: For a replica chain containing two nodes x and y, in which x is still active and y failed at time t_f, if $x <_t y$, $t \leq t_f$, and $t' > t_f$, then $H_t^y \subseteq H_{t'}^x$

- Case 3: Node Addition.

The final case to consider is node additions. Figure 3.2 (case 3) shows a node z joining a replica chain between nodes x and y. We must prove that the following two relations hold true on completion of Flush at time t:

1. $x <_t z \implies H_t^z \subseteq H_t^x$

   Consider all updates at node z. H_t^z is depicted in Figure 3.3. The updates can be classified as: (a) all updates **before** *Flush* reaches the head of the chain ($H_t^{z^{old}}$); and (b) all updates **after** *Flush* reaches the head of the chain ($H_t^{z^{new}}$).

   First, consider $H_t^{z^{old}}$. z received these updates from the tail either as part of precopy or during the subsequent *Flush*. Therefore, $H_t^{z^{old}} \subseteq H_t^{Tail}$.
   Also, $x <_t Tail \implies H_t^{Tail} \subseteq H_t^x$.
   Hence, $H_t^{z^{old}} \subseteq H_t^x$.

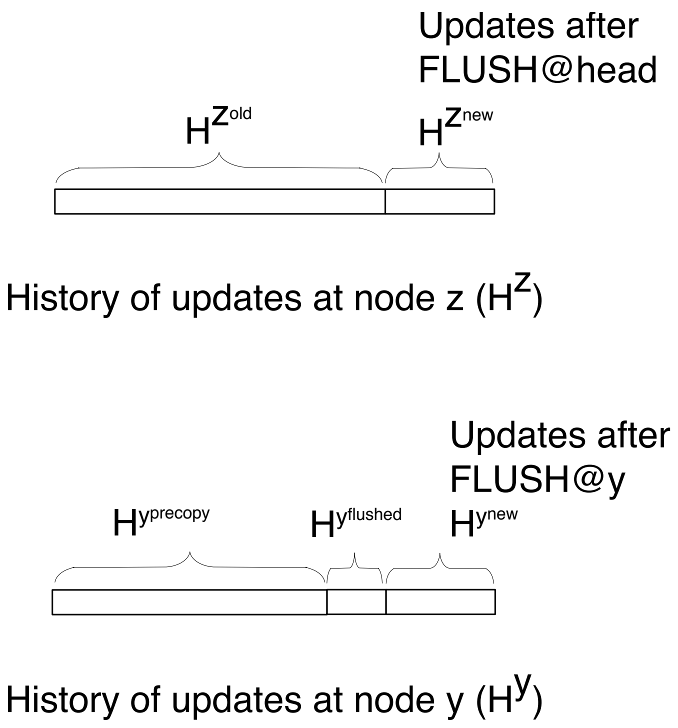

Figure 3.3: History at nodes y and z at time t.

Next, consider $H_t^{z^{new}}$. These updates at z are all the updates following the FLUSH message at the head of the chain; the FLUSH message is sent first to the joining node and then propagates through the current chain before returning back to the joining node. During Flush, these updates are stored at z in a temporary log corresponding to the key range. These updates must have been received from node x, as FLUSH at x would make it set z as its successor. Thus, for updates in $H_t^{z^{new}}$, $x <_t z$, and using the argument from Case 1 we can conclude that $H_t^{z^{new}} \subseteq H_t^x$.

On receiving FLUSH from the tail, z merges its two update sets;
i.e. $H_t^z = H_t^{z^{old}} \cup H_t^{z^{new}}$.
Using our earlier observations that $H_t^{z^{old}} \subseteq H_t^x$ and $H_t^{z^{new}} \subseteq H_t^x$, we can conclude that $H_t^z \subseteq H_t^x$

2. $z <_t y \implies H_t^y \subseteq H_t^z$

   Consider all the updates at node y. H_t^y is depicted in Figure 3.3. The updates can be classified as: (a) all updates that reached the tail **before** *precopy* begins at the Tail ($H_t^{y^{precopy}}$); (b) all updates **between** *Flush* at y and the previous precopy event ($H_t^{y^{flushed}}$); and (c) all updates after FLUSH is received at y ($H_t^{y^{new}}$).

   First, consider $H_t^{y^{precopy}}$. These are updates that z received from the tail as part of precopy. By definition, these are all the updates (and no more) at y that the $Tail$ transfers to z. If the $Tail$ had these updates then y definitely had these updates.
   Therefore, $y <_{precopy} Tail \implies H_{precopy}^{Tail} = H_t^{y^{precopy}} = H_t^{z^{precopy}}$.

   Next, consider $H_t^{y^{flushed}}$. From the arguments in Cases 1 and 2, we know that due to the FIFO nature of channels between nodes and by transitivity, all of these updates arrive at y in the same order , and before FLUSH, as they arrive at $Tail$.
   Therefore, $H_{flushed}^{Tail} = H_t^{y^{flushed}} = H_t^{z^{flushed}}$.

   Last, consider all the updates at y after it receives FLUSH: $H_t^{y^{new}}$. Looking at the FLUSH case in Listing 1 we conclude that y must have got the FLUSH message

from x. On receiving FLUSH at x, it would have updated its Successor to z. All updates at x **after** the FLUSH message therefore would be relayed to z. z, the first node to get the FLUSH message would have already set its Successor to y, and relays these new updates to y. Hence updates in $H_t^{y^{new}}$ must have arrived at y from node z.
Therefore, $H_t^{y^{new}} \subseteq H_t^{z^{new}}$.

It is worth pointing out that y does not process, and instead queues, updates it might receive from z in the short time interval that z is not its Predecessor (the time between when FLUSH was processed at x and then y). This can be seen in the UPDATE case in Listing 1. Once FLUSH is processed at y and z is its Predecessor, y can then process the queued messages from z (see the FLUSH case in Listing 1). This scheme ensures that messages relayed by z (after FLUSH at x) are not processed out of order (before FLUSH) at y.

The history of updates at node $z(H_t^z) = H_t^{z^{precopy}} \cup H_t^{z^{flushed}} \cup H_t^{z^{new}}$.

Using our earlier observations:
$H_t^{y^{precopy}} = H_t^{z^{precopy}}$,
$H_t^{y^{flushed}} = H_t^{z^{flushed}}$,
and $H_t^{y^{new}} \subseteq H_t^{z^{new}}$.

Therefore, we can conclude that $H_t^y \subseteq H_t^z$.

A node failure during node addition (*Flush*) results in the Master ensuring that the predecessor and successor of the joining node are directly linked as was before *Flush*, and notifying them (and the tail) to ignore the FLUSH message. Each FLUSH message therefore is identified by a unique sequence number. The Master locks the chain state during this repair process. On completion of repair, the master unlocks the chain state and triggers the removal of the failed nodes from the replica chain. The new node's addition to the replica chain is redone once the replica chain is fully repaired.

Definition 3.15 (Last Successful Update (LSU)) *:*
master.Recv(UPDATE_ACK(k,v)) $\implies LSU[k] = v$.

The Last Successful Update for a key k ($LSU[k]$) is the value corresponding to the last update(k, v) for which the corresponding UPDATE_ACK message was received at the master node. We denote the node that sent this UPDATE_ACK as the tail corresponding to the LSU.

- $LSU[k]$ at time $t0$ (start): $LSU[k]_{t0} = \emptyset$.
- $LSU[k]$ at time t: $LSU[k]_t = LSU[k]_{t-1}$ if no UPDATE_ACKs for key k were received at the master node between $(t-1)$ and t.

Lemma 3.16 (Update ACK) *At tail node T, if* T.send(master, UPDATE_ACK(k,v)) *at time t, then $T.value[k] = v$ at time t.*

This follows from the fact that in the UPDATE case in Listing 1, the update is logged to the datastore before the UPDATE_ACK is sent.

Lemma 3.17 (Query Response) *At time t, if*

`master.Recv(T, QUERY_RESPONSE(k,v))` *and $T = Tail_k^t$, then $v = LSU[k]$.*

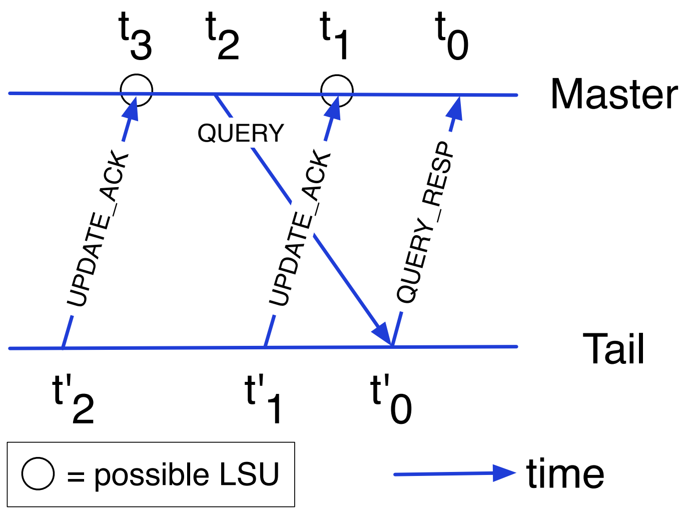

Figure 3.4: If the tail node has not changed since the last UPDATE_ACK, the QUERY_RESPONSE returns the value corresponding to the LSU.

We prove the above lemma by considering two possible cases for the tail node that sent the `QUERY_RESPONSE`:

- The current tail that sent the `QUERY_RESPONSE` is the same as the tail node corresponding to the LSU.

  Assume that for a `Query(k)` it sent out at t_2, the Master node receives a `QUERY_RESPONSE` from the tail node at time t_0. This scenario is depicted in Figure 3.4. The `QUERY` must have been received, and a `QUERY_RESPONSE` sent, by the tail node at some time $t'_0 < t_0$. There are two possibilities for the LSU at the Master: (a) the LSU was updated on receiving an `UPDATE_ACK` at time t_1 such that $t_2 < t_1 < t_0$; or (b) the LSU was updated on receiving an `UPDATE_ACK` at time t_3 such that $t_3 < t_2$. In both of these cases, the corresponding `UPDATE_ACK` must have been sent by the tail node at some time t' (either t'_1 or t'_2) such that $t' < t'_0$. From Lemma 3.16, $Tail_k^{t'_1}.value[k] = LSU[k]_{t_1}$.

By the definition of LSU, $LSU[k]_{t_1} = LSU[k]_{t_0}$.
Also, $Tail_k^{t'_1}.value[k] = Tail_k^{t'_0}.value[k]$. This is true because there is no update at the tail between t'_1 and t'_0; if there was an update in this time range, the LSU should have been present between t_1 and t_0 at the Master which follows from the FIFO nature of the channel between the tail and the Master (a contradiction).
By transitivity, $Tail_k^{t'_0}.value[k] = LSU[k]_{t_0}$.

- The current tail that sent the `QUERY_RESPONSE` has changed since the Last Successful Update; i.e. the current tail is different from the tail node corresponding to the LSU.

  Consider the events at the Master. Specifically, in the time interval between receiving the `QUERY_RESPONSE` and receiving the LSU (the last `UPDATE_ACK` was from node T), there are three possible events pertaining to changing the chain tail: `CR_TAIL_DONE`, `TRUNCATE_ACK`, `FLUSH_ACK`. The tail node could have changed due to one of two reasons:

  1. Tail node failure or truncation (Figure 3.5 Case 1).
     Assume that T_0 is the new tail that sent a `QUERY_RESPONSE`. This implies that for T_0 to be considered a tail by the Master, the Master must have received either a `CR_TAIL_DONE` or `TRUNCATE_ACK` message from T_0 before the `QUERY_RESPONSE`.

Therefore, $T_0 < T \implies H^T \subseteq H^{T_0}$. Furthermore, as T_0 is the new tail, it should have sent UPDATE_ACKs before sending CR_TAIL_DONE or TRUNCATE_ACK (see cases for these messages in Listing 1). As there are no UPDATE_ACK messages from T_0 to the Master (we are considering the LSU after all), $H^T = H^{T_0}$. Hence, $T_{0k}.value[k] = LSU[k]$.

2. Addition of a new tail node to restore the replication factor (Figure 3.5 Case 2). Let us assume T_1 is the new tail that sent a QUERY_RESPONSE. From the above case, we know that $H^T = H^{T_0}$. CR_TAIL_DONE must have been received at the Master from T_0, followed by restoration of the replication factor with precopy and *Flush* for $T1$ resulting in a FLUSH_ACK from T_1. Similar to the earlier case, the lack of an UPDATE_ACK from either T_0 or T_1 implies that $T_{1k}.value[k] = LSU[k]$.

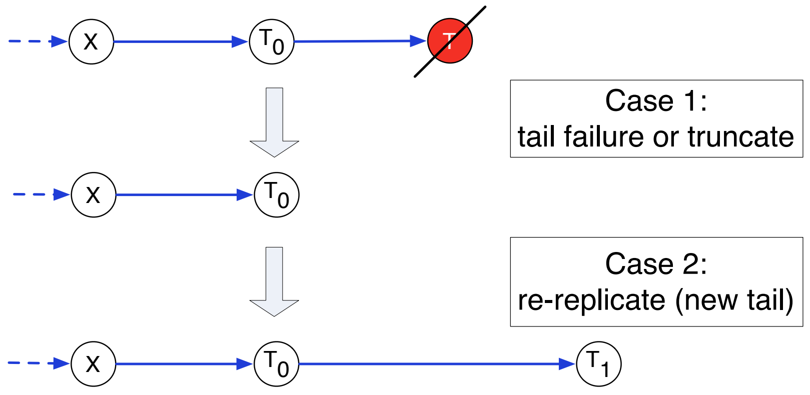

Figure 3.5: Cases when tail node changes.

Proof for Theorem 3.1 Query(K) either gets the value corresponding to the last successful update for K or fails. This follows from Lemma 3.17 and the check at the Master to ensure that the QUERY_RESPONSE is always from the *current* tail node (see QUERY_RESPONSE in Listing 2). A query could also fail if the replica chain is irreparable ($R = 0$).

Proof for Theorem 3.2 This follows from Lemma 3.16 and Invariant 3.14. If Put(K, V) succeeds, then this update is maintained by the system, unless $R = 0$. The update could be overridden by another successful update for the same key K.

Proof for Theorem 3.3 Node failures trigger CR_HEAD, CR_MID, and CR_TAIL messages. Node additions trigger FLUSH. The structure of a replica chain changes at the Master when it receives the ACKs for these events being processed to completion: CR_HEAD_DONE, CR_MID_DONE, CR_TAIL_DONE, FLUSH_ACK. In all of these cases (see Listing 2) restore_replication_factor() ensures that if the current replication factor $(r) > R$, the chain in truncated, and if $r < R$ then a tail node is added (resulting in *Flush*). This ensures that the replication factor quiesces at $r = R$.

```
/*  Per node per range state at backend node N.
1. Predecessor, the predecessor of N
2. Successor, the successor of N
3. Set of the history of updates H  =  H_acked ∪ H_unacked
4. Queue of updates representing H_unacked and some parts of H_acked
5. LA: last update for which N got an ACK from Successor
6. LU: last update got from Predecessor
7. State: one of INACTIVE, JOINING, ACTIVE
*/
void Recv(mesg)
{
  switch (mesg) {

    // precond: N is a replica for mesg.key
    case UPDATE:
      if (State == JOINING) {
        // at a joining node data goes to ``tmp'' store
        ds.tmp.log(mesg.key, mesg.val)
      }
      else (State == ACTIVE) {
        if (mesg.sender != Predecessor) {
            pending_queue(mesg.sender).enqueue(mesg)
        }
        else {
            ds.log(mesg.key, mesg.val)
                H_unacked := Union(H_unacked, {<mesg.seq, mesg.key, mesg.val>})
            if (Successor == null) {
                // send asks to master and along pathway
                send(master, UPDATE_ACK(mesg))
                send(Predecessor, ACK(mesg.seq))
                LA = mesg.seq
            }
            else {
                // forward to next replica
                send(Successor, mesg)
                LU = mesg.seq
            }
        }
      }

    case FLUSHED-UPDATE:
      ds.log(mesg.key, mesg.val)

    // precond: Successor = null, N is a replica for mesg.key, H_unacked = {}
    // only ACTIVE nodes get query requests
    case QUERY:
      v = ds.get(mesg.key)
      send(master, QUERY_RESPONSE(mesg.key, v, mesg.continuation))

    // precond: mesg is in H
    // cumulative ACKs are used to acknowledge the receipt of updates among backend nodes
    // UPDATE_ACK is used acknowledge the receipt of an update to the Master
    case ACK:
      LA := max(mesg.seq, LA)
      // send ack back (towards head) along the replica chain
      if (Predecessor != null) {
        send(Predecessor, ACK(LA))
      }

    case CR_HEAD:
      Predecessor := null
      send(master, CR_HEAD_DONE(mesg.range))

    // sent to the node just before the failed node
    case CR_MID:
      Successor := mesg.NewSuccessor
      // blocking call
      <NewSuccessor.LA, NewSuccessor.LU> = getRepairInfo(N, LA, NewSuccessor)
      if ( <NewSuccessor.LA, NewSuccessor.LU> != null ) {
        // catch-up on the acks NewSuccessor might have sent
        if (NewSuccessor.LA > LA) {
          LA := NewSuccessor.LA
          if (Predecessor != null) {
              // cumulative ACK
              send(Predecessor, ACK(LA))
          }
        }

        // send NewSuccessor updates it might have missed
        while (update u: [u in H_unacked] and [u.seq > NewSuccessor.LU]) {
          send(NewSuccessor, u)
        }

        send(master, CR_MID_DONE(mesg.range))
      }

    case CR_MID_REPAIR:
      // new predecessor is the node that sent the mesg
      Predecessor := mesg.NewPredecessor
      if (LA == null) {
        LA := mesg.NewPredecessor.LA
      }
      send(Predecessor, REPAIR_INFO(LA, LU))

    // this is sent to the new tail
    case CR_TAIL:
      Successor := null

      // send pending acks to master and along pathway
      if (Predecessor != null) {
          while (update u: [u in H_unacked] and [u.seq > LA]) {
              // send asks to master and along pathway
              send(master, UPDATE_ACK(u))
              send(Predecessor, ACK(u.seq))
```

```
            }
        }

        send(master, CR_TAIL_DONE(mesg.range))

      case FLUSH:
        // contains:
        //  (a) range
        //  (b) forwarding path ( JoiningNode-n1-n2-...-nTail-JoiningNode ),
        //  (c) chain member list (n1-n2-n3-JoiningNode-...-nTail)

        // set Predecessor, Successor as per the chain membership view
        New_Predecessor := getPredecessor(mesg.chain_member_list)
        if (New_Predecessor != Predecessor) {
            Predecessor = New_Predecessor
            event_loop_queue.enqueue(pending_queue(mesg.sender))
        }
        Successor := getSuccessor(mesg.chain_member_list)

        if (N == nTail) {
          // at Tail node (second-last on the forwarding path)
          // asynch copy of log to new node, at the end of which
          // the flush mesg is forwarded
            log_flush(JoiningNode, mesg)
         }
         else if (N == JoiningNode) {
             if (State == INACTIVE) {
               // flush is seen the first time
               State = JOINING
               create_tmp_store() // for updates that follow flush
               // Queue, LA, LU are all associated with this tmp store
               send(getNextNode(forwarding_path), mesg)
             }
             else if (State == JOINING) {
               State = ACTIVE
            // flush is seen the second time
            merge(tmp, ds) //merge tmp store into copied store
            send(master, FLUSH_ACK(mesg.payload))
          }
      }
      else {
        // forward mesg at an intermediate node
        send(getNextNode(forwarding_path), mesg)
      }

    // when a node joins the mid/head of a chain, once flush is complete,
    // TRUNCATE is sent to the new tail
    case TRUNCATE:
      if (Successor != null) {
        send(Successor, mesg)
        Successor := null

        // send pending acks to master and along pathway
        if (Predecessor != null) {
            while (update u: [u in H_unacked] and [u.seq > LA]) {
                // send ACKS to master and along pathway
                send(master, UPDATE_ACK(u))
                send(Predecessor, ACK(u.seq))
            }
        }
      }
      else {
        send(master, TRUNCATE_ACK)
      }

  } // switch
}
```

Listing 1: Backend Event Loop.

```
/* State at Master/Frontend.
1. Ring
2. The replication group (G) for every key range.
   G is a tuple of the form:
   <chain_membership_list, state(JOINING/STEADY), joining_node>
3. last_successful_update_map
*/
void Recv(mesg)
{
  switch (mesg) {

    case QUERY:
      client_map[continuation++] = <mesg.client, mesg.client_continuation>
      send(getQueryNode(mesg.key), QUERY(mesg.key, continuation))

    case QUERY_RESPONSE:
      // accept only if it is from the current tail
      if (mesg.sender == get_tail(mesg.key)) {
          <client, client_continuation> = client_map.remove(mesg.continuation)
          send(client, QUERY_RESPONSE(mesg.key, mesg.val, client_continuation))
      }

    case UPDATE:
      client_map[continuation++] = <mesg.client, mesg.client_continuation>
      send(getUpdateNode(mesg.key), UPDATE(mesg.key, mesg.value, continuation))

    case UPDATE_ACK:
      // accept only if it is from the current tail
      if (mesg.sender == get_tail(mesg.key)) {
          <client, client_continuation> = client_map.remove(mesg.continuation)
          send(client, UPDATE_SUCCESS(client_continuation))
          last_successful_update_map[mesg.key] = mesg.value
      }

    case JOIN:
      Ring.add(mesg.new_node, joining, 0)
      replica_groups = getReplicaGroups(mesg.new_node)
      forall groups G in replica_groups {
        send(mesg.new_node,
             FLUSH(G.key_range, //<start_id, end_id>
                   G.forwarding_path, //<new_node, n1, n2, n3, new_node>
                   G.chain_member_list //<n1, n2, new_node, n3>
                  )
             )
      }

    case FLUSH_ACK:
      if (allFlushCollected(mesg.new_node)) {
          Ring.add(mesg.new_node, normal) // adds only if not already present
      } else {
          flush_count := Ring.getFlushCount(mesg.new_node) + 1
          // if added to Ring "normally" earlier, Ring.add ignores op
          Ring.add(mesg.new_node, flush_count)
      }

      // restore replication factor by
      // 1. adding tail when replication factor is under limit
      // or 2. removing tail (TRUNCATE sent to tail's pred) when over limit
      restore_replication_factor(mesg.range)

    case NODE_FAILURE:
      replica_groups = getReplicaGroups(mesg.failed_node)
      forall groups G in replica_groups {
        if(failed_node == G.chain_member_list.first) {
          // head failure
          new_head = getNextWorkingHead(G.chain_member_list)
          send(new_head, CR_HEAD(G.range))
        }
        else if(failed_node == G.chain_member_list.last) {
          // tail failure
          new_tail = getNextWorkingTail(G.chain_member_list)
          send(new_tail, CR_TAIL(G.range))
        }
        else {
          // mid failure
          // send CR_MID to predecessor of failed node
          pred = getPredecessor(G, mesg.failed_node)
          new_neighbor = getNextWorkingNode(G.chain_member_list, pred)
          send(pred, CR_MID(G.range, new_neighbor))
        }
      }
      Ring.remove(failed_node)

    case CR_HEAD_DONE:
    case CR_MID_DONE:
    case CR_TAIL_DONE:
      // restore replication factor by adding tail when required
      restore_replication_factor(mesg.range)

  } // switch
}
```

Listing 2: Event Loop at Master Node.

Chapter 4

Empirical Evaluation Of FAWN-KV

We study a prototype FAWN-KV system running on a 21-node cluster built from commodity PCEngine Alix 3c2 devices [81], commonly used for thin-clients, kiosks, network firewalls, wireless routers, and other embedded applications. These devices have a single-core 500 MHz AMD Geode LX processor, 256 MB DDR SDRAM operating at 400 MHz, and 100 Mbit/s Ethernet. Each node contains one 4 GB Sandisk Extreme IV CompactFlash device. A node consumes 3 W when idle and a maximum of 6 W when deliberately using 100% CPU, network and flash. The nodes are connected to each other and to a 27 W Intel Atom-based front-end node using two 16-port Netgear GS116 GigE Ethernet switches.

We also use a newer 85-node FAWN cluster for front-end cache experiments. This cluster has nodes with dual-core 1.66 GHz Intel Atom processors, 1 GB DRAM, and Intel SSDs.

Evaluation Workload: FAWN-KV targets read-intensive, small object workloads for which key-value systems are often used. The exact object sizes are, of course, application depen-

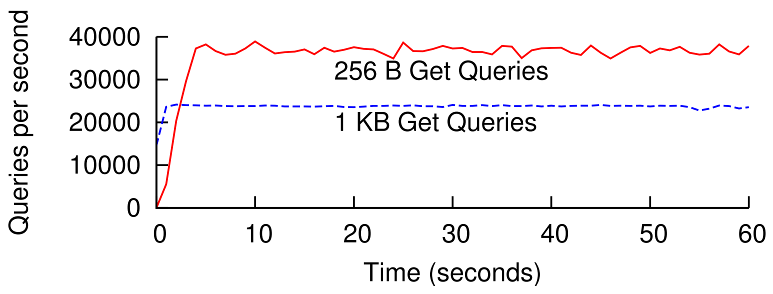

Figure 4.1: Query throughput on 21-node FAWN-KV system for 1 KB and 256 B entry sizes.

dent. In our evaluation, we show query performance for 256 byte and 1 KB values. We select these sizes as proxies for small text posts, user reviews or status messages, image thumbnails, and so on. They represent a quite challenging regime for conventional disk-bound systems, and stress the limited memory and CPU of our wimpy nodes.

4.1 FAWN-KV System Benchmarks

In this section, we evaluate the query rate of our 21-node FAWN-KV system, and the impact of ring membership changes on query throughput and latency. We present results from our 2009 SOSP paper [17].

System Throughput: To measure query throughput, we populated the KV cluster with 20 GB of values, and then measured the maximum rate at which the front-end received query responses for random keys. We disabled front-end caching for this experiment. Figure 4.1 shows that the cluster sustained roughly 36,000 256-byte gets per second (1,700 per second per node) and 24,000 1-KB gets per second (1,100 per second per node). A single node serving a 512 MB datastore over the network could sustain roughly 1,850 256-byte gets per second per node, while it could serve the queries locally at 2,450 256-byte queries per second per node. Thus, a single node serves roughly 70% of the sustained rate that a single FAWN-DS could handle with local queries. The primary reason for the difference is the addition of network overhead and request marshaling and unmarshaling. Another reason for difference is load balance: with random key distribution, some back-end nodes receive more queries than others, slightly reducing system performance.[1]

Impact of a front-end cache. We show the impact of having a front-end cache on query performance using the new FAWN-KV cluster consisting of one high-performance front-end node and 85 low-power Atom-based backend nodes. The front-end and backend node's specifications are shown in Table 6.3. All nodes are connected to a switch; the front-end node uses a 10 GbE link, while back-end nodes use 1 GbE links. The backend nodes each have Intel SSDs [57], but not every backend node is equipped with an SSD. A single node serves approximately 10,000 128-byte queries/second queries from its SSD. We then emulate the SSD I/O behavior by having the backends serve data at 10,000 requests/second from a rate-limited memory-based disk to scale our experiments to more nodes than we have SSDs.

Figure 4.2 shows the overall query throughput as the number of backend nodes increases for three different scenarios:

[1]This problem is fundamental to random load-balanced systems. Terrace and Freedman [98] devised a mechanism for allowing queries to go to any node using chain replication; direct queries to the least-loaded replica can improve load balance.

	Front-end node	Back-end node
CPU:	2× Intel Xeon L5640	Intel Atom D510 [56]
	2.27 GHz	1.66 GHz
# cores:	2×6	2
CPU cache:	2×12 MiB (L3)	512 KiB (L2)
DRAM:	2×24 GiB	1 GiB

Table 4.1: Specifications of front-end and backend nodes

- a uniform distribution across $n * 100,000$ key-value pairs, where n is the number of backend nodes in the system
- a Zipf distribution with parameter 1.01
- a Zipf distribution with parameter 1.01 with a front-end cache.

The experiment with a front-end cache was done by Bin Fan [43]. Bin's work addresses the problem of sizing the front-end cache. They prove an $O(n \log n)$ lower-bound on the necessary cache size and show that this size depends only on the total number of back-end nodes (n), not the number of items stored in the system. The cache size for this experiment was set to $8 * n * \log(n+1)$.

The throughput of the uniform workload scales linearly as the number of nodes grows. The throughput of the Zipf workload grows slowly with diminishing returns with each additional node. With Zipf, the workload is biased to a small set of keys, and the nodes serving these keys become a bottleneck, limiting the overall throughput of the cluster. Zipf's bias towards a small number of keys benefits from having a font-end cache; the system performance even exceeds the aggregate raw throughput that the back-end nodes can provide.

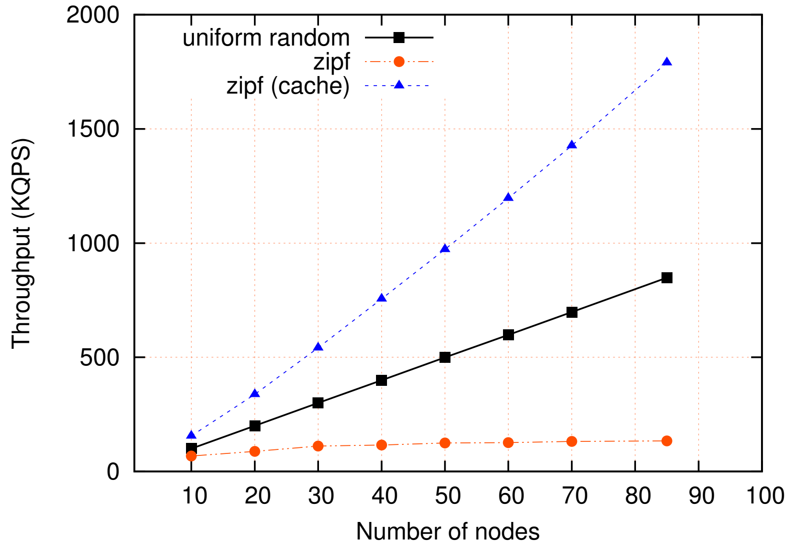

Figure 4.2: Overall throughput as the number of back-end nodes increases from 10 to 85 under different access patterns such as uniformly random, Zipf, and Zipf with a front-end cache

4.2 Impact of Ring Membership Changes

Node joins, leaves, or failures require existing nodes to split, merge, and transfer data while still handling puts and gets. In this section we evaluate the impact of node joins on system query throughput and the impact of maintenance operations such as local splits and compaction on single node query throughput and latency.

Query Throughput During Node Join: In this test, we start a 20-node FAWN-KV Geode cluster populated with 10 GB of key-value pairs and begin issuing get requests uniformly at random to the entire key space. At t=25, we add a node to the ring and continue to issue

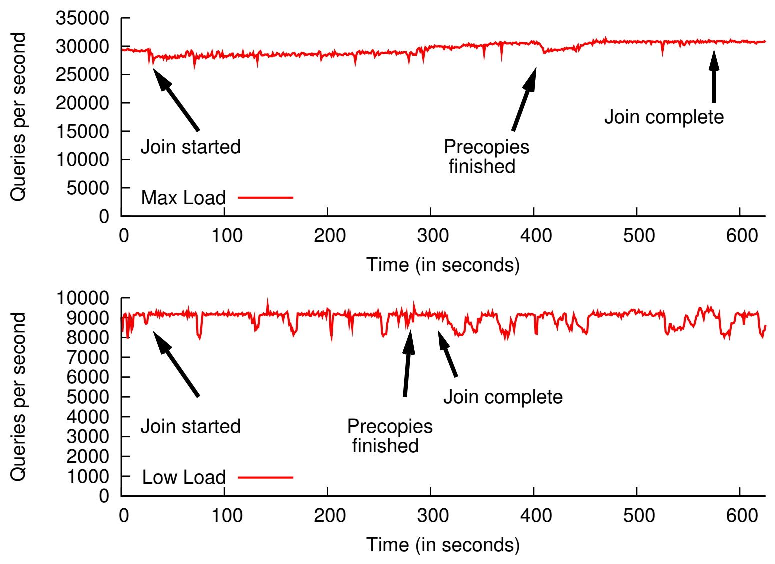

Figure 4.3: Get query rates during node join for max load (top) and low load (bottom).

get requests to the entire cluster. For this experiment, we set $R = 3$ and $V = 1$. Figure 4.3 shows the resulting cluster query throughput during a node join.

The joining node requests pre-copies for $R = 3$ ranges, one range for which it is the tail and two ranges as the head and mid. The three nodes that pre-copy their datastores to the joining node experience a one-third reduction in external query throughput, serving about 1,000 queries/sec. Pre-copying data does not cause significant I/O interference with external requests for data—the pre-copy operation requires only a sequential read of the datastore and bulk sends over the network. The lack of seek penalties for concurrent access on flash together with the availability of spare network capacity results in only a small drop in performance during pre-copying. The other 17 nodes in our cluster are not affected by this join operation and serve queries at their normal rate. The join operation completes

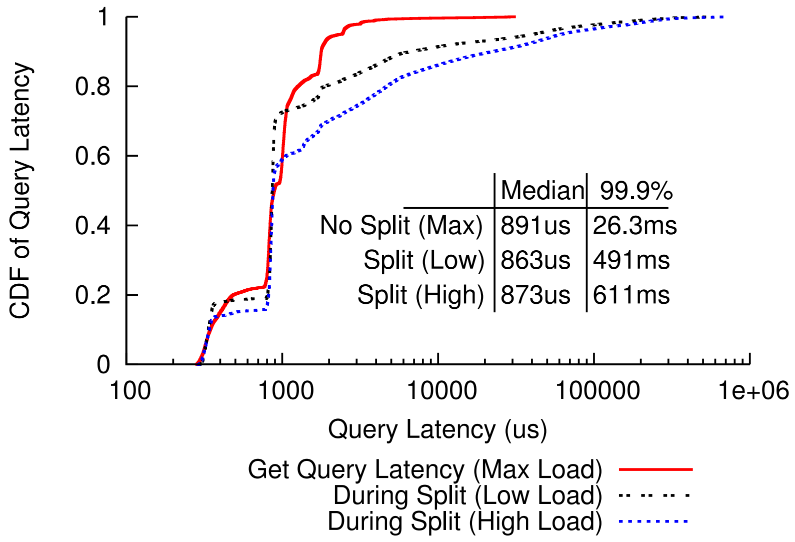

Figure 4.4: Query latency CDF for normal and split workloads.

long after pre-copies finished in this experiment due to the high external query load, and query throughput returns back to the maximum rate.

The experiment above stresses the cluster by issuing requests at the maximum rate the cluster can handle. But most systems offer performance guarantees only for loads below maximum capacity. We run the same experiment above but with an external query load at about 30% of the maximum supported query rate. The three nodes sending pre-copies have enough spare resources available to perform their pre-copy without affecting their ability to serve external queries, so the system's throughput does not drop when the new node is introduced. The join completes shortly after the pre-copies finishes.

Impact of Split on Query Latency: Figure 4.4 shows the distribution of query latency for three workloads: a pure get workload issuing gets at the maximum rate (Max Load), a 500 requests per second workload with a concurrent Split (Split-Low Load), and a 1500 requests per second workload with a Split (Split-High Load). A key range is split into two when a node joins as the head of a chain.

In general, accesses that hit buffer cache are returned in 300 μs including processing and network latency. When the accesses go to flash, the median response time is 800 μs. Even during a split, the median response time remains under 1 ms. The median latency increases with load, so the max load, get-only workload has a slightly higher median latency than the split workloads that have a slightly lower external query load.

Many key-value systems care about 99.9th percentile latency guarantees as well as fast average-case performance. During normal operation, request latency is very low: 99.9% of requests take under 26.3 ms, and 90% take under 2 ms. During a split with low external query load, the additional processing and locking extend 10% of requests above 10 ms. The 99.9%-ile response time during the low-activity split is 491 ms. For a high-rate request workload, the incoming request rate is occasionally higher than can be serviced during the split. Incoming requests are buffered and experience additional queuing delay: the 99.9%-ile response time is 611 ms. Fortunately, these worst-case response times are still on the same order as those worst-case times seen in production key-value systems [39].

Our investigation into this problem identified that this high latency behavior can be attributed to the background garbage collection algorithms implemented in the Flash translation layer in the Flash device. The sequential writes caused by a split, merge, or rewrite in FAWN-DS can trigger the block erasure operation on the CompactFlash, which contains only one programmable Flash chip. During this operation, all read requests to the device are stalled waiting for the device to complete the operation. While individual block erasures often only take 2 ms, the algorithm used on this Flash device performs a bulk block erasure lasting hundreds of milliseconds, causing the 99.9%ile latency behavior of key-value requests to skyrocket. With larger values (1KB), query latency during Split increases further due to a lack of flash device parallelism—a large write to the device blocks concurrent independent reads, resulting in poor worst-case performance. Modern SSDs, in contrast, support and *require* request parallelism to achieve high flash drive performance [84]; assuming they do not perform bulk block erasures that prevent access to the entire device, a future switch to these devices could greatly reduce the effect of background operations on query latency.

We also measured the latency of put requests during normal operation. With R=1, median put latency was about 500μs, with 99.9%ile latency extending to 24.5 ms. With R=3, put requests in chain replication are expected to incur additional latency as the requests get routed down the chain. Median latency increased by roughly three times to 1.58 ms, with 99.9%ile latency increasing only to 30 ms.

Chapter 5

Flex-KV

We are witnessing an explosion of "NoSQL" storage systems, used by companies ranging from startups to industry giants including Amazon, Facebook, Google, and Twitter [39, 72, 85]. Their popularity has resulted in cloud service providers offering NoSQL key-value (KV) systems as building blocks for applications. Each system provides slightly different semantics or is optimized for subtly different use cases. This situation is tragic: It impedes the flexibility of cloud providers and developers by forcing them to commit to a particular model, which they can change only by switching to an entirely different system. It furthermore misses numerous opportunities for worthwhile designs that fall "in-between" existing storage system design choices. We argue that it is possible to create *one* storage system that can meet the needs of all of these applications.

5.1 Motivation

Some applications or operations demand synchronous, durable replication; others favor availability over consistency; and for yet others, the cost of such safety is orders of magnitude too expensive, making it impossible to meet latency or throughput requirements. These requirements sit on a multi-dimensional continuum, with the breadth of NoSQL KV systems testifying to the value of finding a design and implementation well matched to one's requirements. Flash memory and purely in-memory datastores add yet more tradeoffs between sequential and random read/write performance, durability, and power consumption, which further complicates the design space for data storage systems.

Unfortunately, this demand places system designers in a bind: Do they run multiple stores, each operating at maximum efficiency, or do they optimize instead for system complexity by avoiding the need for multiple codebases, vendors, configurations, and so on? We argue that placing systems designers in this bind is unreasonable and, our work suggests, unnecessary. Instead, we show that a KV architecture designed right can easily be configured to support many points along this continuum, from weakly-consistent, non-replicated caches [72] to strongly-consistent, durable disk-backed key-value stores [48].

We argue that a design based on simple chain-based replication enables such a flexible architecture. Flex-KV [83] is a configurable key-value storage system that uses chain-based replication to effectively support a wide range of application requirements. Flex-KV can support DRAM, disk, or Flash-based storage; can support homogeneous or heterogeneous replica chains that can act as an unreliable cache or a durable store; and can trade strong data consistency for higher performance by varying the communication protocols between the replicas in the chain and selecting the query replica. The value of such a system goes beyond ease-of-use: While exploring these dimensions of durability, consistency, and availability, we find new choices for system designs supported by replica chains, such as a cache-consistent memcached, that offer some applications a better balance of performance and cost than was previously available. The schemes and results we discuss here use a simple hash-style key-value system, but we believe that core design ideas apply to other storage systems as well.

5.2 KV Design Space

Systems designers today must pick a particular implementation to meet their application's needs. Current key-value systems differ in three major ways:

Durability What happens to data when the entire KV system is rebooted? Many key-value systems are used as a DRAM *cache* backed by relational databases or storage systems, e.g., the popular *memcached* system. On a cache miss, data is fetched from the backend and is then cached in the key-value system for future use. Updates (puts and deletes) are committed to the backend storage to guarantee data durability.

Other key-value systems act as the primary persistent store without a backend database, e.g., *MemcacheDB* [2] or Redis [87]. There exist important differences in what data these systems may lose upon failures: Some sacrifice performance to write data synchronously to disk, and others favor a higher-performing asynchronous model.

Consistency Some applications may tolerate trading consistency semantics for performance and availability, e.g. Dynamo [39]. A strongly consistent system has the same value across replicas for all keys. Weakly consistent systems allow replicas to return older or different values for any key. For notational clarity, we permit a strongly consistent system that does not guarantee durability, in the face of failures, to either return "failure" or an older value for a key, *if* it correctly informs the client that the value is stale.

Availability in the presence of failures Failures affect data recoverability, system response time, and throughput. We classify availability as:

1. Data Availability (DA): What fraction of nodes can fail before data loss, with a given replication factor?

2. Performance Availability (PA): On a failure, how long does it take until performance is back to that when there were no failures?

5.2.1 New options in the design space: A Memory-efficient Alternative

Existing KV caches offer two extreme options: non-replicated configurations are memory efficient but suffer from poor performance availability; in-memory replicated schemes have higher memory overhead but better performance availability. To bridge this extreme divide, we propose a new design choice: A DRAM-based key-value store that provides high *performance availability* in the face of failures without the memory overhead of the simple replication strategies used today.

Sites such as LiveJournal, Facebook, and Twitter use *memcached* to support a read-mostly workload of millions of page views every day. Because of the huge performance gap between the cache (100,000s of queries per second) and the backing database (1,000s of queries per second), they devote terabytes of DRAM so that nearly all queries are served from cache. Writes invalidate entries in *memcached*, and directly update the database for persistence. Subsequent queries are then cached in *memcached* after being fetched from the database.

The large gap between cache and DB performance means that a cache node failure imposes a sudden high load on the backend database—higher than it can handle, degrading performance or even causing an entire site failure [42, 86]. These sites require high *performance-availability*: they must continue to serve queries at in-memory speeds in the presence of failures.

Non-replicated and in-memory replicated systems offer two extreme options, with trade-offs between recovery time and memory overhead, shown as the "M" and "M-M" configurations in Figure 5.1. In-memory replication, supported by systems such as memcached [71], repcached [88], and Gear6 [47], improves performance availability at the cost of at least twice as much DRAM, already measured in terabytes. The non-replicated system suffers long recovery time because it must read all data into cache from the backend database, potentially requiring random reads from disk.

Disk-backed cache with fast restore (M–D). Instead of naively replicating in-memory, an alternate design logs updates to disk on the replica ("M-D" configurations in Figure 5.1). If the primary fails, the system can rapidly stream the logged cache contents from disk to memory. Synchronous updates can be sped up by buffering updates at the replica before flushing them to disk asynchronously, giving rise to a variant M–($M_b \ldots D$) – a mechanism used in RAMCloud [79]. Alternatively, when used merely as a cache, it is acceptable to lose a small recent window of writes, and so updates can be propagated asynchronously to the disk replicas. To avoid inconsistency, however, it is necessary to synchronously invalidate entries on the replica (in-memory for speed). This combination is memory efficient while still serving both reads and writes at memory speeds – a cache consistent memcache.

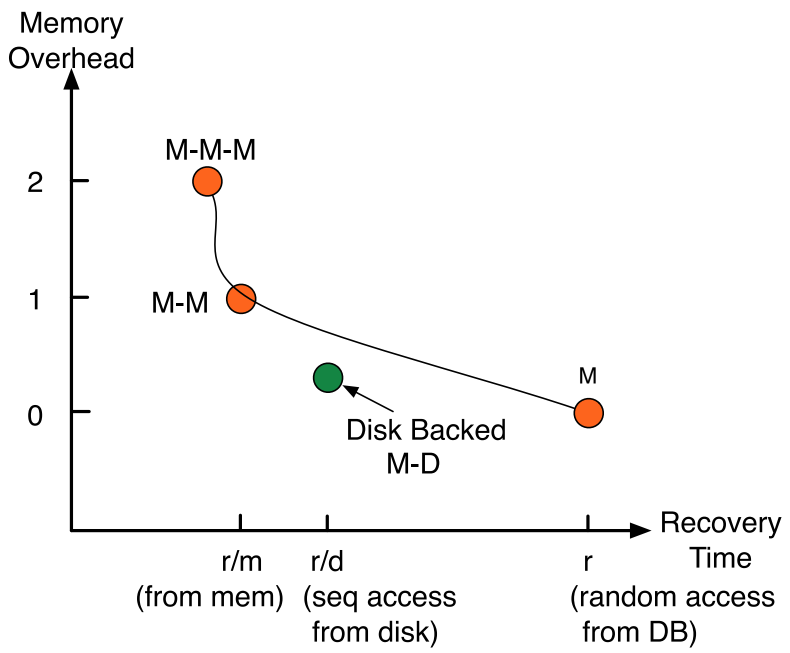

Figure 5.1: Disk backed replicas offer better tradeoffs between memory overhead and performance availability compared to options available today.

5.3 Flex-KV – A Flexible KV System

To implement all the configurations described in the previous section, a KV architecture must be able to: support heterogeneous replicas (e.g., disk, Flash, memory, etc.); direct queries and inserts appropriately (e.g., both to the in-memory replica for high performance, or queries to the tail and updates to the head for strong consistency); send invalidations and updates as configured; flexibly choose whether to send them synchronously or asynchronously; and optionally consult an invalidation table while applying updates on recovery.

Chain-based replication provides an effective mechanism to implement these options. Flex-KV uses replica chains on a consistent hashing ring. Consistent hashing with virtual nodes balances load across the backends and reduces repair time when nodes fail or new nodes join the system. Our prior work, FAWN-KV, uses a similar approach, but it supports only synchronous, durable, and consistent updates to Flash-based replicas, while routing queries to the tail of a replica chain. Flex-KV supports the range of application requirements, listed in the previous section, by supporting:

Replica types: Flex-KV supports different replica types that expose a common storage interface; examples include in-memory replicas (M), disk based logs (D), and buffered logs ($M_b \ldots D$). Flex-KV supports the addition of new datastores as long as they adhere to the storage interface. On each node, it is easy to combine different types of datastores by chaining their interfaces together, as is done in Anvil [69]. All update operations in Flex-KV are log-structured thereby ensuring high performance on different storage devices.

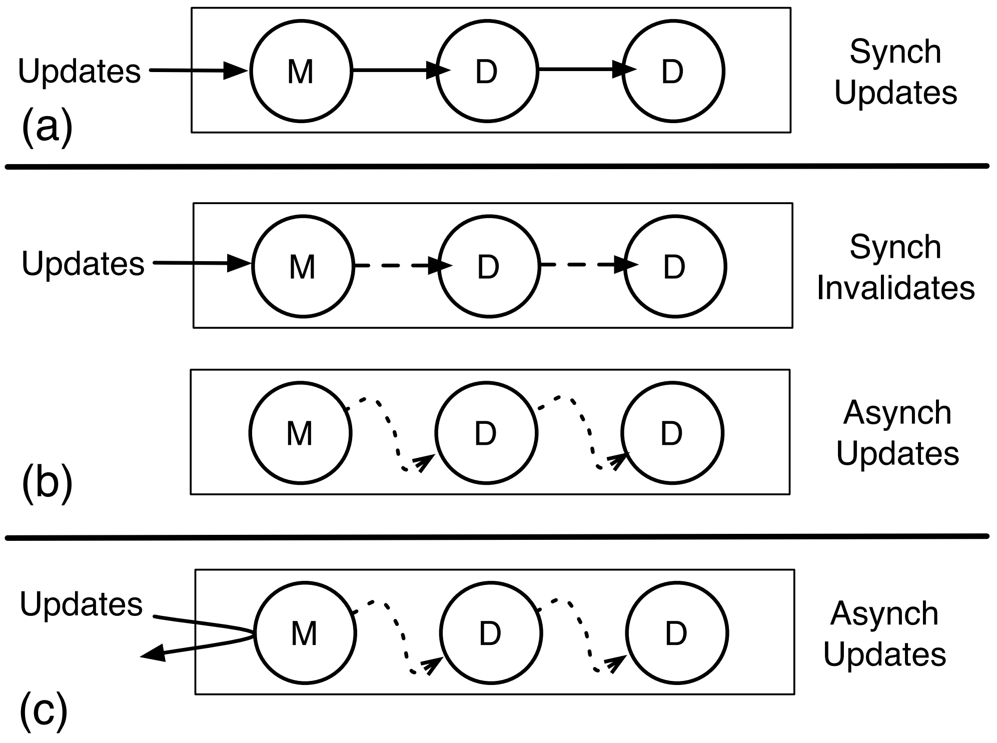

Figure 5.2: Three options for propagating updates through a chain of replicas.

Heterogeneous replica chains: Chaining of replicas [102] provides the basis for a variety of system options (e.g., creating M–D replica chains). Figure 5.2 shows several example configurations. Updates arrive at the head of the chain and propagate through the chain to the tail, as in chain replication. For performance, queries may need to be served not from the tail, but instead from the highest-performing replica (e.g., a memory replica). Flex-KV allows queries to go to different nodes in each replica chain. For example, a high-performance configuration may wish to direct reads and writes to a memory-based replica irrespective of its position in the replica chain. This flexibility "breaks" the simple structure of chain replication to also support a more conventional primary-backup replication style.

Supporting synchronous insertions with read-from-head behavior requires more flexible configuration of when nodes will respond to queries they have already processed, marking un-acknowledged but received writes as tentative. Flex-KV hides the work of allocating replicas and managing the topology, and separates these functions from, e.g., the implementation of the replica's per-node storage.

Flexible update "plumbing" between replicas: The system separates update propagation and invalidation, and allows each to be delivered synchronously or asynchronously. We examine three ways to "plumb" replicas together. Key to these options are the ability to add asynchrony between purely memory-based replicas and disk replicas, to allow the system to operate with memory latency, not disk latency. Flex-KV can send updates using:

- **Synchronous Updates (SU)**: Figure 5.2(a) shows synchronous update propagation through a chain, creating three consistent replicas. Updates succeed only if all replicas are updated.

- **Asynchronous Updates with Synchronous Invalidations (AUSI)**: An update succeeds only if the primary commits the updated value and all secondary replicas receive invalidations (Figure 5.2(b)). Secondary replicas maintain an in-memory invalidation map. Updates are sent in batches from the primary to secondary replicas. Secondary replicas can clear their invalidation map after applying a batch of updates. This scheme enables coherent memory caches that recover from disk (e.g., the example in the previous section).

- **Asynchronous Updates (AU)**: An update succeeds if the primary replica commits the updated value. Secondary replicas receive either individual updates or a batch of updates asynchronously. (Figure 5.2(c))

Replication factor: Flex-KV allows configuring the system with an arbitrary replication factor. Flex-KV maintains this replication factor as long as it is possible to do so. On a node addition the replication factor of the chain it joins goes up by one and it is restored by relinquishing the current tail replica. On a node failure, the replication factor of the affected chain goes down by one and it is restored by recruiting a new tail for this chain. To ensure high performance, node additions and removals are non-blocking operations.

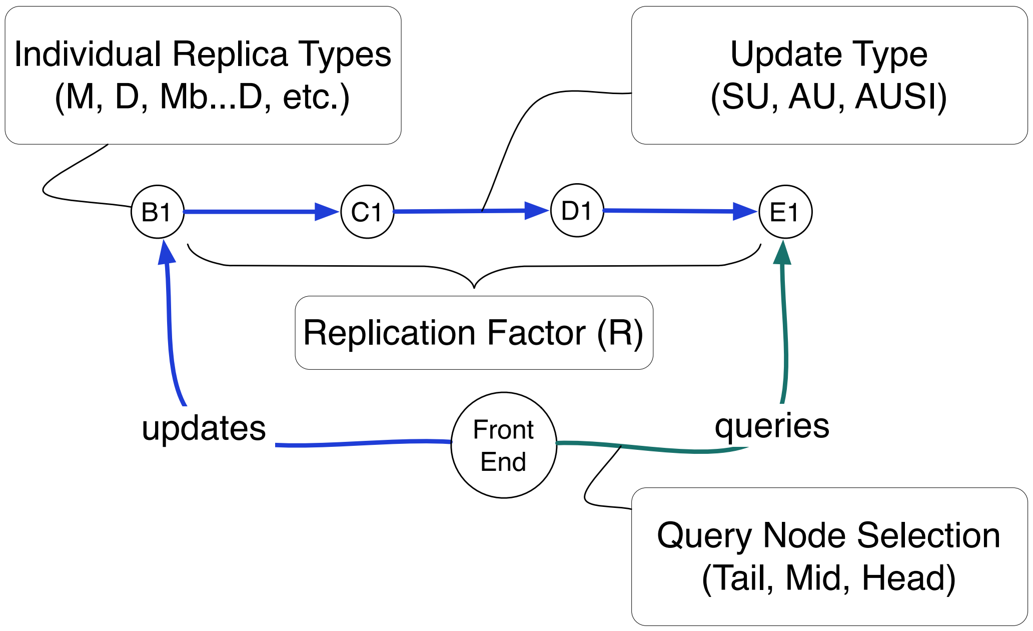

Figure 5.3: Flex-KV supports many different key-value system configurations using four simple knobs.

Ouroboros+ is this new generalization of chain-based replication, which extends Ouroboros to effectively support a wide range of application requirements by (a) selecting from different update mechanisms between replicas, and, (b) selecting a query node in a replica chain. Flex-KV uses Ouroboros+ with different datastores that expose a common storage interface to form homogeneous or heterogeneous replica chains. Flex-KV supports creating many different key-value system configurations using four simple knobs:

1. Replication Factor;
2. Replica Type: Memory, Disk, Flash, Buffered Log, etc.;
3. Update mechanism: SU, AUSI, and AU;
4. Query node: Replica to issue a read request to.

Chapter 6

Flex-KV Evaluation

In the next two sections, we systematically vary the knobs exposed by Flex-KV (Tables 6.1, 6.2). Each cell in those tables represents a unique KV design, to illustrate the coverage of design options provided by Flex-KV's configurability—some choices are similar to currently available point solutions, and some offer new tradeoffs. Furthermore, we compare these design options to highlight the tradeoffs they offer. In Section 6.3 we evaluate Flex-KV's performance on different hardware configurations and compare its performance to a popular existing system, *memcached*.

6.1 Key-Value systems as *caches*

We start with KV systems with a backing database. When used with an external database, a key-value storage system (e.g., memcached) does not need to write synchronously to disk for persistence. It may, however, need replication for high *performance-availability*. Ta-

Configuration	Synchronous Updates (SU) ***Consistent***	Async Updates, Synch Invalidations (AUSI) ***Consistent***	Asynchronous Updates (AU) ***Weakly Consistent***
In-memory Replication M–M +Backing Database(D) ***Memory inefficient***	Updates: Fast (slower than non-replicated systems) PA: Instant recovery Example: Gear6	Updates: Faster for large values PA: Nearly Instant (some cache misses)	Updates: Fast PA: Instant Example: repcached
Disk backed cache M–D +Backing Database(D) ***Memory efficient***	Updates: Slow due to disk flush at replica (buffer for speed: M–($M_b \ldots D$)) PA: Disk scan	Updates: Faster for large values (quick in-memory invalidations) PA: Disk scan (some cache misses)	Updates: Fast PA: Disk scan

Table 6.1: KV configurations with a backing database providing durability. We show configurations with one secondary replica, but the characteristics hold true for similar configurations with n secondary replicas.

ble 6.1 compares systems constructed with different replica types and update propagation mechanisms using four metrics: Read speed, update speed, memory overhead, and performance availability.

The horizontal axis in table 6.1 compares the results of using different plumbing between replicas. In general, synchronous updates provide consistency: A replica can fail and the data is still available in cache, but they bound the system performance to that of the slowest replica in the chain. Asynchronous updates lack consistency, but allow the system to perform at the speed of the fastest replica. AUSI updates provide consistency without data-loss, and decouple performance, making them the best choice when acting as a cache.

Durability: A backing database in all configurations ensures data durability across the board.

Memory overhead: All configurations with in-memory replication have high overhead. M–D configurations with synchronous invalidations need only store invalidations in memory, and so their overhead is determined by the frequency of the asynchronous updates and the workload's update rate.

Update performance: Asynchronous updates are faster than synchronous updates, but this speed advantage also depends on the write cost at the next replica; even memory-to-memory replicas may be faster using AUSI updates if the updates are large. Disk-based replicas benefit more from asynchrony.

Performance Availability (PA): Configurations with both SU and AU recover almost instantaneously on failure. M–M recovers almost instantaneously. M–D is slower than M–M but is still much faster than random queries to the backing DB. The reason these configurations are not as rapid as M–M during recovery is because M–D involves a sequential scan of the log on the disk. The performance availability of configurations with AUSI are sightly worse than the corresponding configurations of SU or AU, because they involve applying invalidations and might incur cache misses for accesses of those key-value pairs that are invalidated. Figure 6.6, for example, shows the best case recovery time for the AUSI scheme where there are hardly any cache misses on recovery.

6.2 Key-Value systems as *stores*

Without a backing DB, most configurations retain the same properties, with one crucial difference: Durability. Configurations using only DRAM are not durable (Table 6.2), but neither does a configuration with a disk replica guarantee durability: In the table, only

Configuration	Synchronous Updates (SU) ***Consistent***	Async Updates, Synch Invalidations (AUSI) ***Consistent***	Asynchronous Updates (AU) ***Weakly Consistent***
M–M ***Memory Inefficient***	Not Durable Example: Gear6, scalaris	Not Durable	Not Durable Example: repcached
M–($M_b \ldots D$) ***Memory Efficient***	Window-loss Durable Example: RAMCloud	Window-loss Durable (Cognizant of loss)	Window-Loss Durable
M–D ***Memory Efficient***	Durable	Window-loss Durable (Cognizant of loss)	Window-Loss Durable Example: Redis
Disk or Flash based *D–D*	Durable Example: FAWN-KV, Hibari	Window Loss Durable (Cognizant of loss)	Window Loss Durable Example: Tokyo Tyrant

Table 6.2: Comparison of different KV configurations without a backing database, all supported by Flex-KV.

configurations with synchronous disk writes, either by starting with a disk, e.g., D–D, or using SU propagation to disk, e.g., M–D, are fully durable.

Configurations using AU and AUSI schemes with a disk replica have *window loss durability*: the system might have an older version of the value for some key, or no value at all, if there is a failure of a primary before updates are propagated to replicas. AUSI invalidations only provide *correct* durability if the invalidations are written synchronously to disk; this does not matter in the cache case, because if both the memory and disk replica fail, the system can recover from the database with some loss of performance. Configurations using AUSI can inform clients that the system lacks the latest update in case of such a failure. Using fully asynchronous updates risks undetected inconsistency.

Window loss durability may suffice for situations in which rare instance of stale data *could* be acceptable, e.g., for data such as web counters or "last visitors" lists, but where complete data loss over all time would not.

6.3 Flex-KV Performance Evaluation

Memcached Baseline Evaluation. Before we evaluate the performance of the different configurations of Flex-KV, we first benchmark *memcached* (version 1.4) so that we can put the performance on Flex-KV in perspective. In the following experiments we try to extract the maximum query throughput from a single memcache server. Our benchmarking uses a modified *memslap* [73] client that in turn uses the standard *libMemcached* [1] client library. Our modifications to memslap ensure query request randomization and additional reporting of performance statistics.

We ran memcached on 3 clusters. Here we report results from the cluster on which memcached performed the best; the 'Core2' cluster nodes have dual core 2.4 GHz CPUs, with 1 GB DRAM, and 500 GB Hard Disks. We use 100-byte values to be certain that network bandwidth is not a bottleneck in our experiments. The memcache client is closed-loop, waiting make the next query until it receives a response to the current request. We increase the concurrency of requests at the server by increasing the number of client instances and by increasing the number of client threads issuing requests in parallel. Our tests do no batch requests; we request individual key-value pairs using TCP as the transport layer and avoid memcached's multiget call interface.

Figure 6.1 shows that increasing the number of concurrent requests to the server, by increasing the number of client threads, steadily improves the memcache server's lookup throughput until it peaks at 100,000 to 110,000.

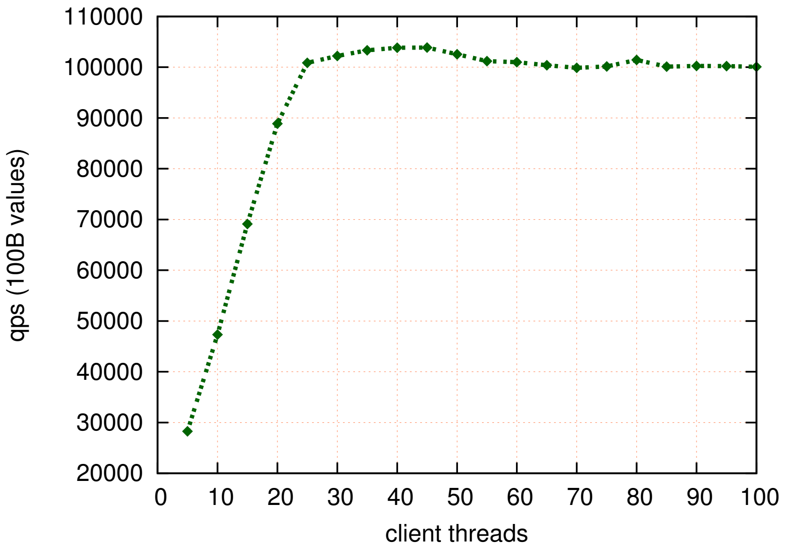

Figure 6.1: Memcached query throughput at a single server instance as we vary the number of clients threads issuing requests.

A single memcached server instance can perform 100,000 to 110,000 lookups per second irrespective of the number of server threads we use (Figure 6.2). Figure 6.3 shows that increasing the number of outstanding requests by increasing the number of clients, each with 50 threads, querying the server does not change the lookup throughput of the server. Memcached is CPU bound at this stage and this can be attributed to the locking overhead on the read path in memcached's implementation.

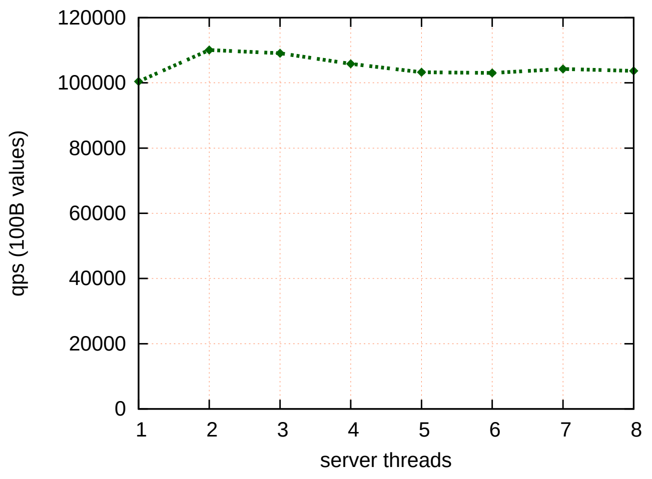

Figure 6.2: Memcached query throughput at a single server instance as we vary the number of server threads. We use a single client with 50 concurrent threads for this test.

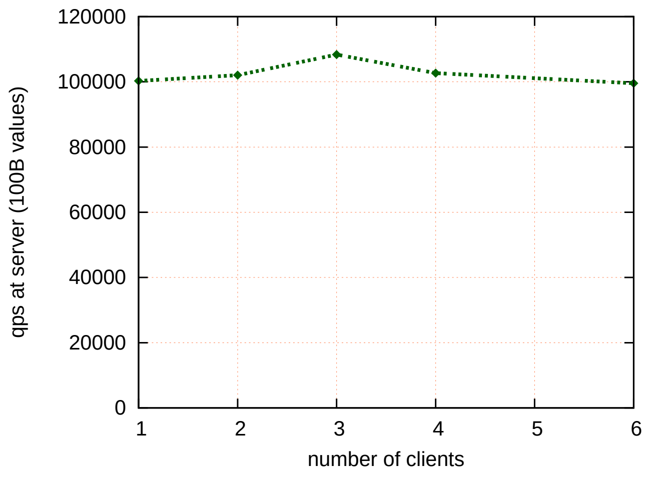

Figure 6.3: Memcached query throughput at a single server instance as we vary the number of clients issuing requests. Individual clients issue 50 outstanding query requests.

Evaluating Flex-KV on Different Hardware Platforms. We evaluate Flex-KV on 3 different test clusters. Table 6.3 shows the specification of individual backend nodes on these three clusters. $Geode_{21}$ is our 2008-era FAWN cluster with 21 nodes, $Atom_{85}$ is a 2010-era FAWN cluster with 85 Atom-based nodes, and Core2 is small cluster of server-class machines. We use 128-byte values in our experiments. Queries requests are asynchronous and are sent in an open loop. We log the end-to-end latency of each request to report median and 99%-ile latencies where appropriate.

Memcached comparison: We first benchmark Flex-KV on the Core2 cluster to compare its performance to memcached. Flex-KV is configured to match memcached by using only memory-based backends with no replication ("M" configuration). Flex-KV achieves a maximum of 100,000 $queries/second$, closely matching memcache throughput on the Core2 cluster. The limit here is primarily due to network, kernel, and Thrift RPC processing overheads; benchmarking a "null" RPC call using Flex-KV results in a limit of 100,000 $calls/second$.

	$Geode_{21}$	$Atom_{85}$	Core2
CPU:	AMD Geode LX 500 MHz	Intel Atom D510 1.66 GHz	Intel Core2 2.40 GHz
# cores:	1	2	2
CPU cache:	128 KB (L2)	512 KB (L2)	4096 KB
DRAM:	256 MB SDRAM at 400 MHz	1 GB	1 GB
Storage:	4 GB Sandisk Extreme IV CF	Intel SSD (520 Series)	500GB Hard Disk

Table 6.3: Comparison of specifications of individual backend nodes of 3 different test clusters that Flex-KV runs on.

Figure 6.4 shows that at low load the median latency is between 250 to 300 μ-seconds, but at high load the median latency shoots up close to 600 μ-seconds. Because the measurement is open-loop, latency increases at high load; Figure 6.5 shows the 99%-ile query latency is 6.5ms when operating at the highest query throughput of 100,000 $lookups/second$.

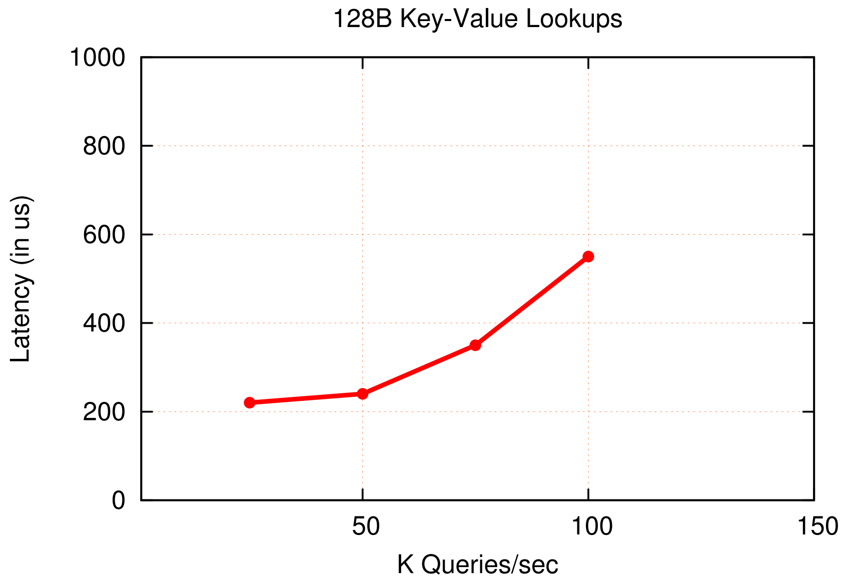

Figure 6.4: Median query latencies when using Flex-KV in the "M" configuration on the Core2 cluster.

Flex-KV on different hardware configurations: Next, we evaluate two simple Flex-KV configurations, M and D, on the three different hardware configurations. Table 6.4 shows the query throughput on these clusters averaged across 5 different runs. All the M configurations achieve a query throughput that closely match the "null" RPC call using Flex-KV on the respective clusters. The disk and Flash based configurations are limited by the performance of the storage device: the slowest configuration is the hard disk based Core2s that incur a high seek latency; the Compact Flash configuration on $Geode_{21}$ is around 6x

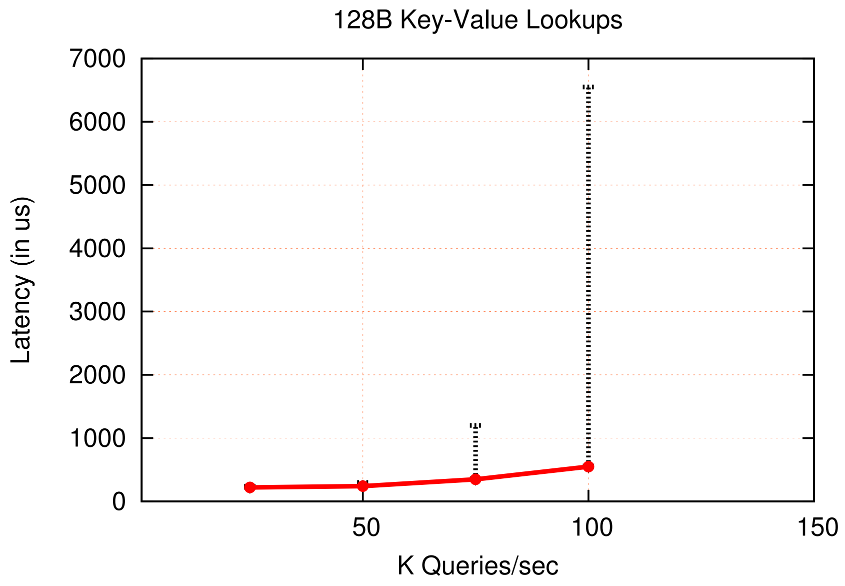

Figure 6.5: Median and 99%-ile query latencies when using Flex-KV in the "M" configuration on the Core2 cluster.

faster than the Core2s; and the Intel SSD-based configurations on $Atom_{85}$ offer a further order of magnitude higher throughput.

Comparing the tradeoffs of different Flex-KV configurations: Each configuration of our Flex-KV implementation trades between durability, memory overhead, performance, and recovery time. Figure 6.6 shows this tradeoff for five different Flex-KV configurations. In this experiment each Core2 backend node stores 15,000 KV pairs with 1KByte values. In-memory replication (M–M) uses twice as much memory as its unreplicated counterpart, but recovers instantly from a node failure. Heterogeneous replica chains (e.g., M–D) are memory efficient and recover more rapidly from node failures than an unreplicated node, and their recovery time is bound by sequential disk scan speeds. Schemes with synchronous updates (SU) are slow when they involve synchronous disk writes.

Configuration	$Geode_{21}$	$Atom_{85}$	Core2
M	14,028	36,000	100,000
Disk or Flash	1,850	10,000	289

Table 6.4: $Queries/second$ for two different Flex-KV configurations on 3 hardware platforms.

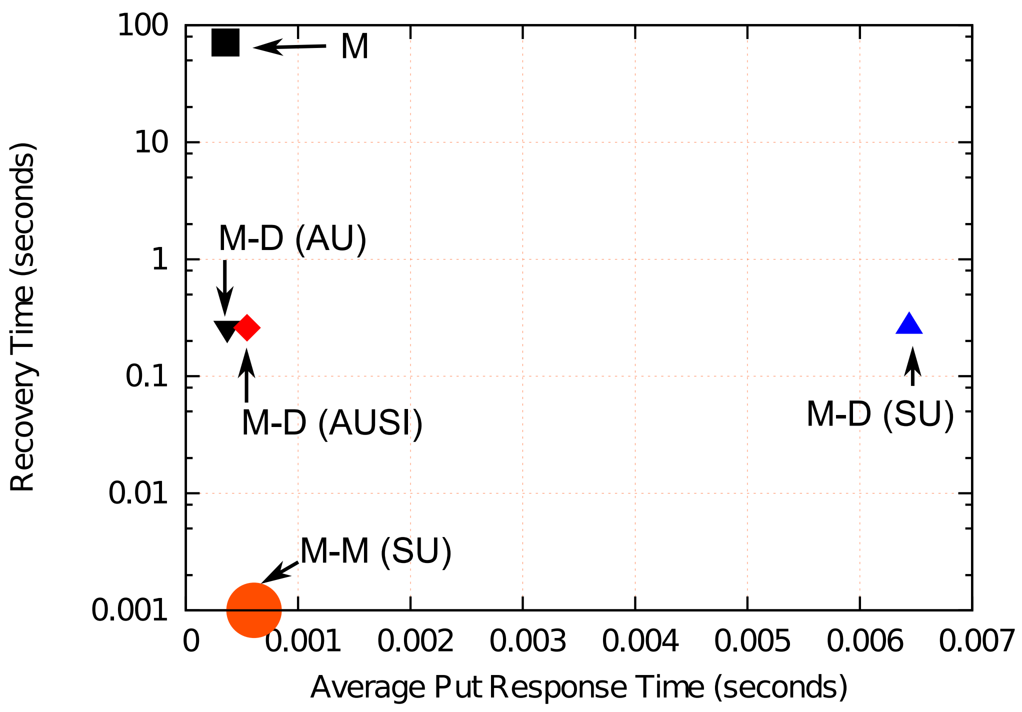

Figure 6.6: Memory overhead, put latency, and recovery time for different key-value configurations when using Flex-KV. The size of the points indicate memory overhead: M–M uses twice as much memory as its unreplicated counterpart.

Chapter 7

Related Work

This thesis work build upon a broad set of research work in distributed systems, storage and database systems, and networking. In this section we summarize the prior work that this dissertation builds upon.

7.1 Flash in Databases and Filesystems

Much prior work is examining the use of flash in databases, examining how database data structures and algorithms can be modified to account for flash storage strengths and weaknesses [64, 65, 75, 77, 101]. this work concluded that NAND flash might be appropriate in "read-mostly, transaction-like workloads", but that flash was a poor fit for high-update databases [75]. Our work, like FlashDB [77] and FD-Trees [65], also notes the benefits of a log structure on flash; however, in their environments, using a log-structured approach slowed query performance by an unacceptable degree. Prior work in sensor networks [36, 70] has employed flash in resource-constrained sensor applications to provide energy-efficient filesystems and single node object stores. In contrast to the above work, FAWN-KV sacrifices range queries by providing only primary-key queries, which eliminates complex indexes and can having separate data and index can therefore support log-structured access without reduced query performance. speed maintenance and failover operations in a clustered, datacenter environment.

Several filesystems are specialized for use on flash. Most are partially log-structured [89], such as the popular JFFS2 (Journaling Flash File System) for Linux. Our observations about flash's performance characteristics follow a long line of research [40, 75, 77, 84, 111]. Past solutions to these problems include the eNVy filesystem's use of battery-backed SRAM to buffer copy-on-write log updates for high performance [108], followed closely by purely flash-based log-structured filesystems [60].

7.2 High-throughput Storage and Analysis

Recent work such as Hadoop [5, 12] or MapReduce [38] running on GFS [49] has examined techniques for scalable, high-throughput computing on massive datasets. More specialized examples include SQL-centric options such as the massively parallel data-mining appliances from Netezza [78], AsterData [11], and others [6, 7, 13]. As opposed to the random-access workloads we examine for FAWN-KV, these systems provide bulk throughput for massive datasets with low selectivity or where indexing in advance is difficult.

7.3 Distributed Hash Tables

Related cluster and wide-area hash table-like services include Distributed data structure (DDS) [51], a persistent data management layer designed to simplify cluster-based Internet services. FAWN-KV's major points of differences with DDS are a result of FAWN's hardware architecture, and the protocols for strong consistency with minimal blocking during churn. These same differences apply to systems such as Dynamo [39] and Voldemort [85] which trade consistency for partition tolerance. Grapevine [24] was an important early example of trading consistency for simplicity, and Bayou [82, 100] later explored trading consistency for availability in application-specific ways. FAWN-KV trades partition tolerance for availability and strong consistency [25, 50]. Systems such as Boxwood [68] focus on the higher level primitives necessary for managing storage clusters. Our focus was on a simple key-value abstraction.

The replica selection strategy in Ouroborous is similar to that used in systems like CFS [32, 33], DHash++ [34], Dynamo [39], and PAST & Pastry [41, 90, 91]. However, none of these systems use chain based replication and they provide weaker consistency guarantees. In Base DHash, each data block is stored as 14 erasure-coded fragments using the IDA coding algorithm, one on each successors of the key, and any seven of which are sufficient to reconstruct the block. Base DHash, used in CFS, was designed for the wide-area, and does not guarantee strong consistency. Similarly, Dynamo, trades consistency for partition tolerance of data stored across datacenters. Dynamo is optimized for write-availability, and offers eventual data consistency with conflict resolution, if any, on a read.

Ouroboros, generalizes chain replication to allow node additions to any position in the chain, to offer provably strong consistency with minimal blocking.

7.4 Examples of systems designed to provide flexibility to end users

Click [61] is a flexible software router composed of configurable elements responsible for packet processing. Ensemble and JGroups are high-performance modularized protocol architectures for replicated services that allow the construction of reliable multicast protocols using basic building blocks [52, 103]. Similarly, PRACTI [35] and PADS [21] provide a policy based architecture for building distributed storage protocols. TACT [109, 110], designed for small deployments of computers in the WAN, dynamically trades consistency for availability (and performance) based on system, network, and client characteristics, but assumes that all nodes store all of the data and receive all updates. To provide strong consistency, TACT switches from an anti-entorpy based model to a two phase commit protocol. Unlike these systems, Flex-KV uses a single mechanism, Ouroboros+, which can be configured to support the different application requirements of consistency, durability, performance, and cost.

WheelFS [97] is a wide-area user-level distributed file system that allows applications to choose a tradeoff between performance and consistency. WheelFS allows these adjustments via semantic cues, which provide application control over consistency, failure handling, and file and replica placement. WheelFS uses primary-backup replication and provides close-to-open consistency, while also supporting eventual consistency if requested. Yahoo!'s data storage platform, PNUTS [31], offers a read-write interface to records of database like tables. Records are replicated across multiple data centers, and all replicas of a given record apply all updates in the same order. PNUTS supports control of wide-area tradeoffs by allowing applications to choose between reading the latest version of a record, any version of a record, or a version of a record newer than a specified version. The tradeoffs offered by the update mechanisms and selection of query nodes, in both WheelFS and PNUTS, is somewhat similar to Ouroboros+. Unlike WheelFS and PNUTS, Flex-KV can support different replica types as part of the same replica group offering better tradeoffs between memory overhead and recovery time. Flex-KV currently does not support cues to offer applications different runtime guarantees; we leave this for future explorations.

7.5 Logging in Distributed Databases.

Distributed databases have long used logging for recovery to ensure that the database state is not corrupted as a result of software, system, or media failures, and to respect ACID properties [46, 66, 74]. These systems have to scan the log and perform undo and redo operations on recovery. FAWN-KV and Flex-KV use a log as the data store for high performance writes and replication; they don't provide ACID guarantees and do not have to rescan the log to undo or redo operations on recovery.

7.6 Cluster-based “NoSQL” systems.

We covered a wide range of key-value stores in Section 5. But NoSQL systems are not just restricted to key-value store. Systems such as Redis [87] (a data structure server supporting strings, hashes, lists, sets and sorted sets); BigTable [27] (column-oriented storage); and MongoDB [28]) (document-based storage) all demonstrate the value of richer data models. Extending Flex-KV beyond key-value storage without crossing a cliff of complexity is left as a future undertaking; it is not the focus of this thesis.

Chapter 8

Conclusion

This dissertation demonstrates that it possible for a key-value architecture to be easily configured to support many points along the KV system design continuum, from weakly-consistent, non-replicated caches to strongly-consistent, durable disk-backed key-value stores. Our work made the following contributions:

- First, we presented a new replication protocol, Ouroboros, which generalized chain-based replication to allow node additions to any part of the replica chain. Ouroboros is designed to minimize blocking during node additions and deletions while guaranteeing strong data consistency (per-key linearizability [53]). We proved the correctness of Ouroboros with regard to Query, Update, and Replication guarantees. We also presented the design, implementation, and evaluation of a distributed key-value storage system, FAWN-KV, with the goal of supporting the three key properties of fault tolerance, high performance, and generality. FAWN-KV achieved these goals using four principles: (a) sequential writes for high performance and generality; (b) replication for fault tolerance; (c) use of Ouroboros with minimal blocking for high performance when nodes are added or fail; and (d) load balancing for high performance;

- Second, we presented a generalization of chain-based replication, Ouroboros+, which extends Ouroboros to effectively support a wide range of application requirements by (a) selecting from different update protocols between replicas, and, (b) selecting a query node in a replica chain. We described Flex-KV, which uses Ouroboros+ with different datastores that expose a common storage interface to form homogeneous or heterogeneous replica chains. Flex-KV can support DRAM, Flash, and disk-based storage; can act as an unreliable cache or a durable store; and can offer strong or weak data consistency. The value of such a system goes beyond ease-of-use: We enabled new choices for system designs, such as a cache-consistent memcached, that offer some applications a better balance of performance and cost than was previously available. Finally, we empirically evaluated Flex-KV on three different hardware configurations to show its effectiveness.

The battle for dominance in the "Big data" space rages on [9, 14, 96]; on one side are the "NoSQL" adherents, on the other are "NewSQL" proponents. The core design ideas described in this dissertation, though in the context of hash-style key-value systems, apply to replicated storage systems in general, irrespective of the camp they are in. In this context, our work raises some important questions for the future.

A first question our work raises is how users of storage systems should choose a configuration. Although orthogonal to the arguments of this dissertation, this question needs to be addressed in the future, not only for Flex-KV, but also for storage systems that provide different guarantees and tradeoffs at large.

A second important question is how to extend the Flex-KV idea to encompass more NoSQL designs. Two important axes to consider include partition tolerance (as in Dynamo [39], which trades consistency for partition tolerance), and a richer data model. Systems such as Redis [87] (a data structure server supporting strings, hashes, lists, sets and sorted sets); BigTable [27] (column-oriented storage); and MongoDB [28]) (document-based storage) all demonstrate the value of richer data models.

Creating "one store for all" is difficult, and it is likely that no one system can truly meet the needs of all users. However, our progress designing Flex-KV suggests that the right set of configuration and coupling primitives can make structured storage systems able to satisfy a wide variety of performance, consistency, and durability requirements.

Bibliography

[1] libMemcached. http://libmemcached.org/libMemcached.html, . [Cited on page 83.]

[2] memcachedb. http://memcachedb.org/, . [Cited on page 69.]

[3] Ouroboros. http://en.wikipedia.org/wiki/Ouroboros, . [Cited on page 2.]

[4] Eventually Consistent - Revisited. http://www.allthingsdistributed.com/2008/12/eventually_consistent.html, 2008. URL retrieved August 2012. [Cited on pages 2 and 14.]

[5] Cloudera. http://www.cloudera.com/, 2009. URL retrieved August 2009. [Cited on page 92.]

[6] Greenplum: the petabyte-scale database for data warehousing and business intelligence. http://www.greenplum.com/, 2009. URL retrieved August 2009. [Cited on page 92.]

[7] Teradata. http://www.teradata.com/, 2009. URL retrieved August 2009. [Cited on page 92.]

[8] Velocity 2010. http://perspectives.mvdirona.com/2010/07/01/Velocity2010.aspx, 2010. URL retrieved August 2012. [Cited on page 2.]

[9] Why Enterprises Are Uninterested in NoSQL. http://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext/, 2010. URL retrieved August 2012. [Cited on page 98.]

[10] Twitter's New Search Architecture. http://engineering.twitter.com/2010/10/twitters-new-search-architecture.html, 2010. URL retrieved August 2012. [Cited on page 2.]

[11] Aster data. http://www.asterdata.com, 2011. [Cited on page 92.]

[12] Hadoop. http://hadoop.apache.org/, 2011. [Cited on page 92.]

[13] Paraccel analytic platform. http://www.paraccel.com, 2011. [Cited on page 92.]

[14] Stonebraker trapped in Stonebraker 'fate worse than death'. http://dom.as/2011/07/08/stonebraker-trapped/, 2011. URL retrieved August 2012. [Cited on page 98.]

[15] Apache Thrift. https://thrift.apache.org/, 2011. [Cited on page 10.]

[16] Amazon DynamoDB. http://aws.amazon.com/dynamodb/, 2012. URL retrieved August 2012. [Cited on page 3.]

[17] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc.*

22nd ACM Symposium on Operating Systems Principles (SOSP), Big Sky, MT, October 2009. [Cited on pages 9 and 58.]

[18] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS Performance*, London, UK, June 2012. [Cited on pages 2 and 4.]

[19] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *Trans. Storage*, 4(3):8:1–8:28, November 2008. ISSN 1553-3077. doi: 10.1145/1416944.1416947. URL `http://doi.acm.org/10.1145/1416944.1416947`. [Cited on page 29.]

[20] Joel F. Bartlett. A nonstop kernel. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP '81, pages 22–29, New York, NY, USA, 1981. ACM. ISBN 0-89791-062-1. doi: 10.1145/800216.806587. URL `http://doi.acm.org/10.1145/800216.806587`. [Cited on page 8.]

[21] N. Belaramani, J. Zheng, A. Nayte, M. Dahlin, and R. Grimm. PADS: A Policy Architecture for building Distributed Storage systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2009. [Cited on page 94.]

[22] M. Berezecki, Eitan Frachtenberg, Mike Paleczny, and K. Steele. Many-core key-value store. In *Proceedings of the Second International Green Computing Conference*, Orlando, FL, USA, Aug. 2011. [Cited on page 4.]

[23] Kenneth P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, December 1993. ISSN 0001-0782. doi: 10.1145/163298.163303. URL `http://doi.acm.org/10.1145/163298.163303`. [Cited on page 12.]

[24] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. Grapevine: an exercise in distributed computing. *Commun. ACM*, 25(4):260–274, April 1982. ISSN 0001-0782. doi: 10.1145/358468.358487. URL `http://doi.acm.org/10.1145/358468.358487`. [Cited on page 93.]

[25] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. doi: 10.1145/343477.343502. URL `http://doi.acm.org/10.1145/343477.343502`. [Cited on page 93.]

[26] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In *ACM SIGFIDET (now SIGMOD) workshop on Data description, access, and control*, pages 249––264, New York, NY, USA, 1974. ACM. [Cited on page 1.]

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. 7th USENIX OSDI*, Seattle, WA, November 2006. [Cited on pages 96 and 99.]

[28] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, 2010. [Cited on pages 96 and 99.]

[29] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251203.1251223`. [Cited on page 20.]

[30] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. 13: 377––387, June 1970. [Cited on page 1.]

[31] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008. ISSN 2150-8097. doi: 10.1145/1454159.1454167. URL `http://dx.doi.org/10.1145/1454159.1454167`. [Cited on page 95.]

[32] Frank Dabek. A distributed hash table. *Ph.D. Thesis, MIT*, 2000. [Cited on pages 15 and 93.]

[33] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001. [Cited on pages 15 and 93.]

[34] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a dht for low latency and high throughput. In *Proceedings of*

the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1, NSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251175.1251182. [Cited on pages 15 and 93.]

[35] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006. [Cited on page 94.]

[36] Hui Dai, Michael Neufeld, and Richard Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Baltimore, MD, November 2004. [Cited on page 92.]

[37] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining (WSDM)*, 2009. [Cited on page 3.]

[38] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX OSDI*, San Francisco, CA, December 2004. [Cited on page 92.]

[39] Guiseppe DeCandia, Deinz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007. [Cited on pages 1, 3, 15, 64, 67, 70, 93 and 99.]

[40] Fred Douglis, Frans Kaashoek, Brian Marsh, Ramon Caceres, Kai Li, and Joshua Tauber. Storage alternatives for mobile computers. In *Proc. 1st USENIX OSDI*, pages 25–37, Monterey, CA, November 1994. [Cited on page 92.]

[41] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proc. HotOS VIII*, pages 75–80, Schloss-Elmau, Germany, May 2001. [Cited on pages 15 and 93.]

[42] Facebook Outage. More Details on Today's Outage | Facebook, Sept. 2010. `http://www.facebook.com/note.php?note_id=431441338919`. [Cited on pages 2 and 71.]

[43] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proc. 2nd ACM Symposium on Cloud Computing (SOCC)*, Cascais, Portugal, October 2011. [Cited on page 60.]

[44] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924943.1924948`. [Cited on pages 2 and 27.]

[45] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Cluster-based Scalable Network Services. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malô, France, October 1997. [Cited on page 1.]

[46] Michael Franklin. Concurrency control and recovery. *In A. B. TUCKER Ed., The Computer Science and Engineering Handbook*, pages 1058–1077, 1997. [Cited on page 95.]

[47] Gear6. Cache replication with Gear6 Web Cache. http://www.gear6.com/sites/gear6.com/files/Cache%20Replication%20Solution%20Brief%20Final.pdf. [Cited on page 71.]

[48] Inc Gemini Mobile Technologies. Hibari: A Whitepaper. http://www.geminimobile.com/developers-center/white-papers/hibari-whitepaper-v1.0.pdf. [Cited on pages 4 and 68.]

[49] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, October 2003. [Cited on page 92.]

[50] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL http://doi.acm.org/10.1145/564585.564601. [Cited on page 93.]

[51] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for Internet service construction. In *Proc. 4th USENIX OSDI*, San Diego, CA, November 2000. [Cited on page 93.]

[52] Bela Ban (Red Hat). JGroups - A Toolkit for Reliable Multicast Communication. http://www.jgroups.org/. [Cited on page 94.]

[53] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL http://doi.acm.org/10.1145/78969.78972. [Cited on pages 2, 14 and 97.]

[54] Urs Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 2010. [Cited on page 4.]

[55] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10, Berkeley, CA, USA, 2010. USENIX Association. [Cited on page 12.]

[56] Intel. Atom Processor D510. http://ark.intel.com/Product.aspx?id=43098. [Cited on pages 33 and 60.]

[57] Intel Solid-State Drive. 520 Series. http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-520-series.html. [Cited on page 59.]

[58] JFFS2. The Journaling Flash File System. http://sources.redhat.com/jffs2/. [Cited on page 10.]

[59] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-*

ninth annual ACM symposium on Theory of computing, pages 654–663, New York, NY, USA, 1997. ACM. [Cited on page 11.]

[60] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proc. USENIX Annual Technical Conference*, New Orleans, LA, January 1995. [Cited on pages 10 and 92.]

[61] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000. [Cited on page 94.]

[62] Michael Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, WMCSA '02, pages 40–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1647-5. URL `http://dl.acm.org/citation.cfm?id=832315.837557`. [Cited on page 20.]

[63] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998. ISSN 0734-2071. [Cited on page 12.]

[64] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory SSD in enterprise database applications. In *Proc. ACM SIGMOD*, Vancouver, BC, Canada, June 2008. [Cited on page 91.]

[65] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. Tree indexing on flash disks. In *Proceedings of 25th International Conference on Data Engineering*, March 2009. [Cited on page 91.]

[66] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorieand, T. G. Price, F. Putzolu, and B. W. Wade. Notes on distributed databases (Draffan and Poole, Eds.). In *Distributed Data Bases*, pages 247–284. Cambridge Univ. Press, Cambridge, U.K., 1980. [Cited on page 95.]

[67] David B. Lomet. High speed on-line backup when using logical log operations. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 34–45, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4. doi: 10.1145/342009.335378. URL http://doi.acm.org/10.1145/342009.335378. [Cited on page 20.]

[68] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proc. 6th USENIX OSDI*, San Francisco, CA, December 2004. [Cited on page 93.]

[69] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. Modular data storage with Anvil. In *SOSP*, pages 147–160, 2009. [Cited on page 73.]

[70] Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Boulder, CO, October 2006. [Cited on page 92.]

[71] Memcache Failover FAQ. How does memcached handle failover? http://code.google.com/p/memcached/wiki/FAQ. [Cited on page 71.]

[72] Memcached. A distributed memory object caching system. http://memcached.org/, 2011. [Cited on pages 1, 4, 67 and 68.]

[73] memslap. Load testing and benchmarking a server. http://docs.libmemcached.org/memslap.html. [Cited on page 83.]

[74] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992. ISSN 0362-5915. doi: 10.1145/128765.128770. URL http://doi.acm.org/10.1145/128765.128770. [Cited on page 95.]

[75] Daniel Myers. On the use of NAND flash memory in high-performance relational databases. M.S. Thesis, MIT, February 2008. [Cited on pages 10, 91 and 92.]

[76] Suman Nath and Phillip B. Gibbons. Online maintenance of very large random samples on flash storage. In *Proc. VLDB*, Auckland, New Zealand, August 2008. [Cited on page 10.]

[77] Suman Nath and Aman Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *Proceedings of ACM/IEEE International Conference on Information Processing in Sensor Networks*, Cambridge, MA, April 2007. [Cited on pages 10, 91 and 92.]

[78] Netezza. Business intelligence data warehouse appliance. http://www.netezza.com/, 2006. [Cited on page 92.]

[79] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011. [Cited on page 72.]

[80] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, December 2002. ISSN 0163-5980. doi: 10.1145/844128.844162. URL http://doi.acm.org/10.1145/844128.844162. [Cited on page 20.]

[81] PCEngines. PC Engines Alix3c2. http://pcengines.ch/alix3c2.htm. [Cited on pages 33 and 57.]

[82] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, USA, 1997. ACM. [Cited on page 93.]

[83] Amar Phanishayee, David Andersen, Himabindu Pucha, Anna Povzner, and Wendy Belluomini. Flex-KV: Enabling High-performance and Flexible KV Systems. In *Proceedings of the First Workshop on Management of Big Data Systems*, San Jose, CA, USA, Sept. 2012. [Cited on page 68.]

[84] Milo Polte, Jiri Simsa, and Garth Gibson. Enabling enterprise solid state disks performance. In *Proc. Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, Washington, DC, March 2009. [Cited on pages 10, 65 and 92.]

[85] Project Voldemort. A distributed key-value storage system. http://project-voldemort.com. [Cited on pages 1, 67 and 93.]

[86] reddit. May 2010 ”State of the Servers” report. `http://blog.reddit.com/2010/05/reddits-may-2010-state-of-servers.html`. [Cited on pages 2 and 71.]

[87] Redis. A data structure server. `http://redis.io/documentation`. [Cited on pages 69, 96 and 99.]

[88] repcached. Add data replication to memcached. `http://repcached.lab.klab.org/`. [Cited on page 71.]

[89] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992. [Cited on pages 10 and 92.]

[90] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP ’01, pages 188–201, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. doi: 10.1145/502034.502053. URL `http://doi.acm.org/10.1145/502034.502053`. [Cited on pages 15 and 93.]

[91] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. 18th IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001. [Cited on pages 15 and 93.]

[92] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984. ISSN 0734-2071. [Cited on pages 27 and 34.]

[93] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. ISSN 0360-0300. doi: 10.1145/98163.98167. URL `http://doi.acm.org/10.1145/98163.98167`. [Cited on page 8.]

[94] SeaMicro. Seamicro. `http://www.seamicro.com`, 2010. [Cited on page 4.]

[95] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, San Diego, CA, August 2001. [Cited on page 11.]

[96] Michael Stonebraker. Sql databases v. nosql databases. *Commun. ACM*, 53(4):10–11, April 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721659. URL `http://doi.acm.org/10.1145/1721654.1721659`. [Cited on pages 1 and 98.]

[97] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Morris. Flexible, Wide-Area Storage for Distributed Systems with WheelFS. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2009. [Cited on page 94.]

[98] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proc. USENIX Annual Technical Conference*, San Diego, CA, June 2009. [Cited on page 59.]

[99] Doug Terry. Replicated data consistency explained through baseball. *MSR Technical Report*. [Cited on pages 2 and 14.]

[100] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, USA, 1995. ACM. [Cited on page 93.]

[101] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proc. ACM SIGMOD*, Providence, RI, June 2009. [Cited on page 91.]

[102] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. 6th USENIX OSDI*, San Francisco, CA, December 2004. [Cited on pages 2, 14 and 74.]

[103] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. *Softw. Pract. Exper.*, 28:963–979, July 1998. ISSN 0038-0644. [Cited on page 94.]

[104] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009. ISSN 0001-0782. doi: 10.1145/1435417.1435432. URL `http://doi.acm.org/10.1145/1435417.1435432`. [Cited on pages 2 and 14.]

[105] Matt Welsh. A retrospective on SEDA. `http://matt-welsh.blogspot.com/2010/07/retrospective-on-seda.html`. [Cited on page 10.]

[106] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001. [Cited on page 10.]

[107] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251175.1251188`. [Cited on page 20.]

[108] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proc. 6th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 1994. [Cited on page 92.]

[109] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association. [Cited on page 94.]

[110] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. *SIGOPS Oper. Syst. Rev.*, 35:29–42, October 2001. ISSN 0163-5980. [Cited on page 94.]

[111] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. MicroHash: An efficient index structure for flash-based sensor devices. In *Proc. 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, December 2005. [Cited on page 92.]

[112] Zookeeper. Apache ZooKeeper. http://hadoop.apache.org/zookeeper/. [Cited on page 12.]