Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

a gossip based distributed failure detection and membership protocol. Dynamo is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution.

In the past year, Dynamo has been the underlying storage technology for a number of the core services in Amazon's e-commerce platform. It was able to scale to extreme peak loads efficiently without any downtime during the busy holiday shopping season. For example, the service that maintains shopping cart (Shopping Cart Service) served tens of millions requests that resulted in well over 3 million checkouts in a single day and the service that manages session state handled hundreds of thousands of concurrently active sessions.

The main contribution of this work for the research community is the evaluation of how different techniques can be combined to provide a single highly-available system. It demonstrates that an eventually-consistent storage system can be used in production with demanding applications. It also provides insight into the tuning of these techniques to meet the requirements of production systems with very strict performance demands.

The paper is structured as follows. Section 2 presents the background and Section 3 presents the related work. Section 4 presents the system design and Section 5 describes the implementation. Section 6 details the experiences and insights gained by running Dynamo in production and Section 7 concludes the paper. There are a number of places in this paper where additional information may have been appropriate but where protecting Amazon's business interests require us to reduce some level of detail. For this reason, the intra- and inter-datacenter latencies in section 6, the absolute request rates in section 6.2 and outage lengths and workloads in section 6.3 are provided through aggregate measures instead of absolute details.

2. BACKGROUND
Amazon's e-commerce platform is composed of hundreds of services that work in concert to deliver functionality ranging from recommendations to order fulfillment to fraud detection. Each service is exposed through a well defined interface and is accessible over the network. These services are hosted in an infrastructure that consists of tens of thousands of servers located across many data centers world-wide. Some of these services are stateless (i.e., services which aggregate responses from other services) and some are stateful (i.e., a service that generates its response by executing business logic on its state stored in persistent store).

Traditionally production systems store their state in relational databases. For many of the more common usage patterns of state persistence, however, a relational database is a solution that is far from ideal. Most of these services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS. This excess functionality requires expensive hardware and highly skilled personnel for its operation, making it a very inefficient solution. In addition, the available replication technologies are limited and typically choose consistency over availability. Although many advances have been made in the recent years, it is still not easy to scale-out databases or use smart partitioning schemes for load balancing.

This paper describes Dynamo, a highly available data storage technology that addresses the needs of these important classes of services. Dynamo has a simple key/value interface, is highly available with a clearly defined consistency window, is efficient in its resource usage, and has a simple scale out scheme to address growth in data set size or request rates. Each service that uses Dynamo runs its own Dynamo instances.

2.1 System Assumptions and Requirements
The storage system for this class of services has the following requirements:

Query Model: simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (i.e., blobs) identified by unique keys. No operations span multiple data items and there is no need for relational schema. This requirement is based on the observation that a significant portion of Amazon's services can work with this simple query model and do not need any relational schema. Dynamo targets applications that need to store objects that are relatively small (usually less than 1 MB).

ACID Properties: ACID (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. This has been widely acknowledged by both the industry and academia [5]. Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability. Dynamo does not provide any isolation guarantees and permits only single key updates.

Efficiency: The system needs to function on a commodity hardware infrastructure. In Amazon's platform, services have stringent latency requirements which are in general measured at the 99.9^{th} percentile of the distribution. Given that state access plays a crucial role in service operation the storage system must be capable of meeting such stringent SLAs (see Section 2.2 below). Services must be able to configure Dynamo such that they consistently achieve their latency and throughput requirements. The tradeoffs are in performance, cost efficiency, availability, and durability guarantees.

Other Assumptions: Dynamo is used only by Amazon's internal services. Its operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization. Moreover, since each service uses its distinct instance of Dynamo, its initial design targets a scale of up to hundreds of storage hosts. We will discuss the scalability limitations of Dynamo and possible scalability related extensions in later sections.

2.2 Service Level Agreements (SLA)
To guarantee that the application can deliver its functionality in a bounded time, each and every dependency in the platform needs to deliver its functionality with even tighter bounds. Clients and services engage in a Service Level Agreement (SLA), a formally negotiated contract where a client and a service agree on several system-related characteristics, which most prominently include the client's expected request rate distribution for a particular API and the expected service latency under those conditions. An example of a simple SLA is a service guaranteeing that it will
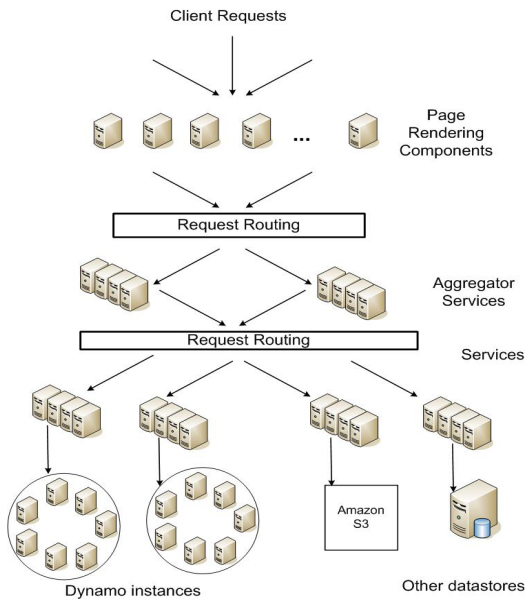
Figure 1: Service-oriented architecture of Amazon's platform

provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.

In Amazon's decentralized service oriented infrastructure, SLAs play an important role. For example a page request to one of the e-commerce sites typically requires the rendering engine to construct its response by sending requests to over 150 services. These services often have multiple dependencies, which frequently are other services, and as such it is not uncommon for the call graph of an application to have more than one level. To ensure that the page rendering engine can maintain a clear bound on page delivery each service within the call chain must obey its performance contract.

Figure 1 shows an abstract view of the architecture of Amazon's platform, where dynamic web content is generated by page rendering components which in turn query many other services. A service can use different data stores to manage its state and these data stores are only accessible within its service boundaries. Some services act as aggregators by using several other services to produce a composite response. Typically, the aggregator services are stateless, although they use extensive caching.

A common approach in the industry for forming a performance oriented SLA is to describe it using average, median and expected variance. At Amazon we have found that these metrics are not good enough if the goal is to build a system where **all** customers have a good experience, rather than just the majority. For example if extensive personalization techniques are used then customers with longer histories require more processing which impacts performance at the high-end of the distribution. An SLA stated in terms of mean or median response times will not address the performance of this important customer segment. To address this issue, at Amazon, SLAs are expressed and measured at the 99.9th percentile of the distribution. The choice for 99.9% over an even higher percentile has been made based on a cost-benefit analysis which demonstrated a significant increase in cost to improve performance that much. Experiences with Amazon's

production systems have shown that this approach provides a better overall experience compared to those systems that meet SLAs defined based on the mean or median.

In this paper there are many references to this 99.9th percentile of distributions, which reflects Amazon engineers' relentless focus on performance from the perspective of the customers' experience. Many papers report on averages, so these are included where it makes sense for comparison purposes. Nevertheless, Amazon's engineering and optimization efforts are not focused on averages. Several techniques, such as the load balanced selection of write coordinators, are purely targeted at controlling performance at the 99.9th percentile.

Storage systems often play an important role in establishing a service's SLA, especially if the business logic is relatively lightweight, as is the case for many Amazon services. State management then becomes the main component of a service's SLA. One of the main design considerations for Dynamo is to give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost-effectiveness.

2.3   Design Considerations

Data replication algorithms used in commercial systems traditionally perform synchronous replica coordination in order to provide a strongly consistent data access interface. To achieve this level of consistency, these algorithms are forced to tradeoff the availability of the data under certain failure scenarios. For instance, rather than dealing with the uncertainty of the correctness of an answer, the data is made unavailable until it is absolutely certain that it is correct. From the very early replicated database works, it is well known that when dealing with the possibility of network failures, strong consistency and high data availability cannot be achieved simultaneously [2, 11]. As such systems and applications need to be aware which properties can be achieved under which conditions.

For systems prone to server and network failures, availability can be increased by using optimistic replication techniques, where changes are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated. The challenge with this approach is that it can lead to conflicting changes which must be detected and resolved. This process of conflict resolution introduces two problems: when to resolve them and who resolves them. Dynamo is designed to be an eventually consistent data store; that is all updates reach all replicas eventually.

An important design consideration is to decide *when* to perform the process of resolving update conflicts, i.e., whether conflicts should be resolved during reads or writes. Many traditional data stores execute conflict resolution during writes and keep the read complexity simple [7]. In such systems, writes may be rejected if the data store cannot reach all (or a majority of) the replicas at a given time. On the other hand, Dynamo targets the design space of an "always writeable" data store (i.e., a data store that is highly available for writes). For a number of Amazon services, rejecting customer updates could result in a poor customer experience. For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures. This requirement forces us to push the complexity of conflict resolution to the reads in order to ensure that writes are never rejected.

The next design choice is *who* performs the process of conflict resolution. This can be done by the data store or the application. If conflict resolution is done by the data store, its choices are rather limited. In such cases, the data store can only use simple policies, such as "last write wins" [22], to resolve conflicting updates. On the other hand, since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart. Despite this flexibility, some application developers may not want to write their own conflict resolution mechanisms and choose to push it down to the data store, which in turn chooses a simple policy such as "last write wins".

Other key principles embraced in the design are:

Incremental scalability: Dynamo should be able to scale out one storage host (henceforth, referred to as "*node*") at a time, with minimal impact on both operators of the system and the system itself.

Symmetry: Every node in Dynamo should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities. In our experience, symmetry simplifies the process of system provisioning and maintenance.

Decentralization: An extension of symmetry, the design should favor decentralized peer-to-peer techniques over centralized control. In the past, centralized control has resulted in outages and the goal is to avoid it as much as possible. This leads to a simpler, more scalable, and more available system.

Heterogeneity: The system needs to be able to exploit heterogeneity in the infrastructure it runs on. e.g. the work distribution must be proportional to the capabilities of the individual servers. This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.

3. RELATED WORK

3.1   Peer to Peer Systems
There are several peer-to-peer (P2P) systems that have looked at the problem of data storage and distribution. The first generation of P2P systems, such as Freenet and Gnutella[1], were predominantly used as file sharing systems. These were examples of unstructured P2P networks where the overlay links between peers were established arbitrarily. In these networks, a search query is usually flooded through the network to find as many peers as possible that share the data. P2P systems evolved to the next generation into what is widely known as structured P2P networks. These networks employ a globally consistent protocol to ensure that any node can efficiently route a search query to some peer that has the desired data. Systems like Pastry [16] and Chord [20] use routing mechanisms to ensure that queries can be answered within a bounded number of hops. To reduce the additional latency introduced by multi-hop routing, some P2P systems (e.g., [14]) employ O(1) routing where each peer maintains enough routing information locally so that it can route requests (to access a data item) to the appropriate peer within a constant number of hops.

---

[1] http://freenetproject.org/, http://www.gnutella.org

Various storage systems, such as Oceanstore [9] and PAST [17] were built on top of these routing overlays. Oceanstore provides a global, transactional, persistent storage service that supports serialized updates on widely replicated data. To allow for concurrent updates while avoiding many of the problems inherent with wide-area locking, it uses an update model based on conflict resolution. Conflict resolution was introduced in [21] to reduce the number of transaction aborts. Oceanstore resolves conflicts by processing a series of updates, choosing a total order among them, and then applying them atomically in that order. It is built for an environment where the data is replicated on an untrusted infrastructure. By comparison, PAST provides a simple abstraction layer on top of Pastry for persistent and immutable objects. It assumes that the application can build the necessary storage semantics (such as mutable files) on top of it.

3.2   Distributed File Systems and Databases
Distributing data for performance, availability and durability has been widely studied in the file system and database systems community. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus [15] and Coda [19] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. The Farsite system [1] is a distributed file system that does not use any centralized server like NFS. Farsite achieves high availability and scalability using replication. The Google File System [6] is another distributed file system built for hosting the state of Google's internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunkservers. Bayou is a distributed relational database system that allows disconnected operations and provides eventual data consistency [21].

Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level resolution. All of them, however, guarantee eventual consistency. Similar to these systems, Dynamo allows read and write operations to continue even during network partitions and resolves updated conflicts using different conflict resolution mechanisms. Distributed block storage systems like FAB [18] split large size objects into smaller blocks and stores each block in a highly available manner. In comparison to these systems, a key-value store is more suitable in this case because: (a) it is intended to store relatively small objects (size < 1M) and (b) key-value stores are easier to configure on a per-application basis. Antiquity is a wide-area distributed storage system designed to handle multiple server failures [23]. It uses a secure log to preserve data integrity, replicates each log on multiple servers for durability, and uses Byzantine fault tolerance protocols to ensure data consistency. In contrast to Antiquity, Dynamo does not focus on the problem of data integrity and security and is built for a trusted environment. Bigtable is a distributed storage system for managing structured data. It maintains a sparse, multi-dimensional sorted map and allows applications to access their data using multiple attributes [2]. Compared to Bigtable, Dynamo targets applications that require only key/value access with primary focus on high availability where updates are not rejected even in the wake of network partitions or server failures.
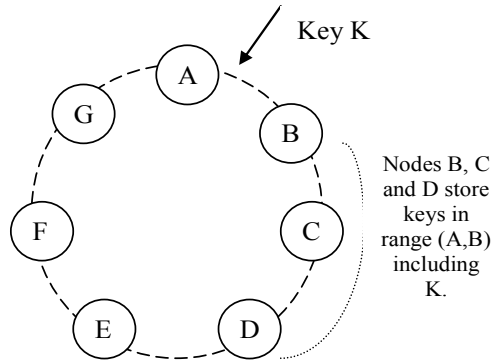
Figure 2: Partitioning and replication of keys in Dynamo ring.

Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency to replicated data. Although strong consistency provides the application writer a convenient programming model, these systems are limited in scalability and availability [7]. These systems are not capable of handling network partitions because they typically provide strong consistency guarantees.

3.3   Discussion

Dynamo differs from the aforementioned decentralized storage systems in terms of its target requirements. First, Dynamo is targeted mainly at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes. This is a crucial requirement for many Amazon applications. Second, as noted earlier, Dynamo is built for an infrastructure within a single administrative domain where all nodes are assumed to be trusted. Third, applications that use Dynamo do not require support for hierarchical namespaces (a norm in many file systems) or complex relational schema (supported by traditional databases). Fourth, Dynamo is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds. To meet these stringent latency requirements, it was imperative for us to avoid routing requests through multiple nodes (which is the typical design adopted by several distributed hash table systems such as Chord and Pastry). This is because multi-hop routing increases variability in response times, thereby increasing the latency at higher percentiles. Dynamo can be characterized as a zero-hop DHT, where each node maintains enough routing information locally to route a request to the appropriate node directly.

4. SYSTEM ARCHITECTURE

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is not possible, so this paper focuses on the core distributed systems techniques used in Dynamo: partitioning, replication, versioning, membership, failure handling and scaling.

Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

4.1   System Interface

Dynamo stores objects associated with a key through a simple interface; it exposes two operations: get() and put(). The get(*key*) operation locates the object replicas associated with the *key* in the storage system and returns a single object or a list of objects with conflicting versions along with a *context*. The put(*key, context, object*) operation determines where the replicas of the *object* should be placed based on the associated *key*, and writes the replicas to disk. The *context* encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object. The context information is stored along with the object so that the system can verify the validity of the context object supplied in the put request.

Dynamo treats both the key and the object supplied by the caller as an opaque array of bytes. It applies a MD5 hash on the key to generate a 128-bit identifier, which is used to determine the storage nodes that are responsible for serving the key.

4.2   Partitioning Algorithm

One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes (i.e., storage hosts) in the system. Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing [10], the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position.

Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Dynamo uses a variant of consistent hashing (similar to the one used in [10, 20]): instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of "virtual nodes". A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions (henceforth, "tokens") in the ring. The process of fine-tuning Dynamo's partitioning scheme is discussed in Section 6.

Using virtual nodes has the following advantages:

- If a node becomes unavailable (due to failures or routine maintenance), the load handled by this node is evenly dispersed across the remaining available nodes.

- When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.

- The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

4.3 Replication

To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured *"per-instance"*. Each key, *k*, is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its Nth predecessor. In Figure 2, node B replicates the key *k* at nodes C and D in addition to storing it locally. Node D will store the keys that fall in the ranges (A, B], (B, C], and (C, D].

The list of nodes that is responsible for storing a particular key is called the *preference list*. The system is designed, as will be explained in Section 4.8, so that every node in the system can determine which nodes should be in this list for any particular key. To account for node failures, preference list contains more than N nodes. Note that with the use of virtual nodes, it is possible that the first N successor positions for a particular key may be owned by less than N distinct physical nodes (i.e. a node may hold more than one of the first N positions). To address this, the preference list for a key is constructed by skipping positions in the ring to ensure that the list contains only distinct physical nodes.

4.4 Data Versioning

Dynamo provides eventual consistency, which allows for updates to be propagated to all replicas asynchronously. A put() call may return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent get() operation may return an object that does not have the latest updates.. If there are no failures then there is a bound on the update propagation times. However, under certain failure scenarios (e.g., server outages or network partitions), updates may not arrive at all replicas for an extended period of time.

There is a category of applications in Amazon's platform that can tolerate such inconsistencies and can be constructed to operate under these conditions. For example, the shopping cart application requires that an "*Add to Cart*" operation can never be forgotten or rejected. If the most recent state of the cart is unavailable, and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved. But at the same time it shouldn't supersede the currently unavailable state of the cart, which itself may contain changes that should be preserved. Note that both "*add to cart*" and "*delete item from cart*" operations are translated into put requests to Dynamo. When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later.

In order to provide this kind of guarantee, Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version (syntactic reconciliation). However, version branching may happen, in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to *collapse* multiple branches of data evolution back into one (semantic reconciliation). A typical example of a collapse operation is "merging" different versions of a customer's shopping cart. Using this reconciliation mechanism, an "add to cart" operation is never lost. However, deleted items can resurface.

It is important to understand that certain failure modes can potentially result in the system having not just two but several versions of the same data. Updates in the presence of network partitions and node failures can potentially result in an object having distinct version sub-histories, which the system will need to reconcile in the future. This requires us to design applications that explicitly acknowledge the possibility of multiple versions of the same data (in order to never lose any updates).

Dynamo uses vector clocks [12] in order to capture causality between different versions of the same object. A vector clock is effectively a list of (node, counter) pairs. One vector clock is associated with every version of every object. One can determine whether two versions of an object are on parallel branches or have a causal ordering, by examine their vector clocks. If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation.

In Dynamo, when a client wishes to update an object, it must specify which version it is updating. This is done by passing the context it obtained from an earlier read operation, which contains the vector clock information. Upon processing a read request, if
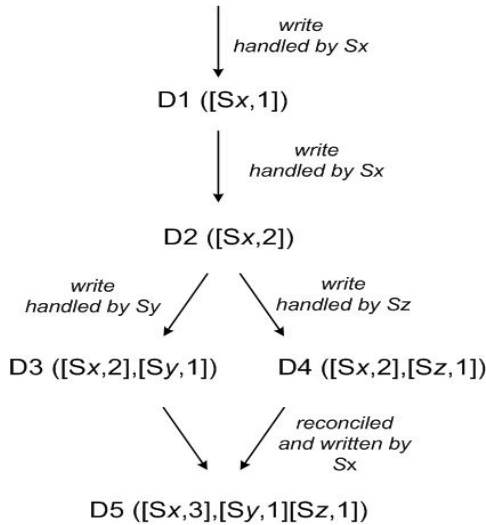
Figure 3: Version evolution of an object over time.

Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context. An update using this context is considered to have reconciled the divergent versions and the branches are collapsed into a single new version.

To illustrate the use of vector clocks, let us consider the example shown in Figure 3. A client writes a new object. The node (say Sx) that handles the write for this key increases its sequence number and uses it to create the data's vector clock. The system now has the object D1 and its associated clock [(Sx, 1)]. The client updates the object. Assume the same node handles this request as well. The system now also has object D2 and its associated clock [(Sx, 2)]. D2 *descends* from D1 and therefore over-writes D1, however there may be replicas of D1 lingering at nodes that have not yet seen D2. Let us assume that the same client updates the object again and a different server (say Sy) handles the request. The system now has data D3 and its associated clock [(Sx, 2), (Sy, 1)].

Next assume a different client reads D2 and then tries to update it, and another node (say Sz) does the write. The system now has D4 (descendant of D2) whose version clock is [(Sx, 2), (Sz, 1)]. A node that is aware of D1 or D2 could determine, upon receiving D4 and its clock, that D1 and D2 are overwritten by the new data and can be garbage collected. A node that is aware of D3 and receives D4 will find that there is no causal relation between them. In other words, there are changes in D3 and D4 that are not reflected in each other. Both versions of the data must be kept and presented to a client (upon a read) for semantic reconciliation.

Now assume some client reads both D3 and D4 (the context will reflect that both values were found by the read). The read's context is a summary of the clocks of D3 and D4, namely [(Sx, 2), (Sy, 1), (Sz, 1)]. If the client performs the reconciliation and node Sx coordinates the write, Sx will update its sequence number in the clock. The new data D5 will have the following clock: [(Sx, 3), (Sy, 1), (Sz, 1)].

A possible issue with vector clocks is that the size of vector clocks may grow if many servers coordinate the writes to an object. In practice, this is not likely because the writes are usually handled by one of the top N nodes in the preference list. In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow. In these scenarios, it is desirable to limit the size of vector clock. To this end, Dynamo employs the following clock truncation scheme: Along with each (node, counter) pair, Dynamo stores a timestamp that indicates the last time the node updated the data item. When the number of (node, counter) pairs in the vector clock reaches a threshold (say 10), the oldest pair is removed from the clock. Clearly, this truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. However, this problem has not surfaced in production and therefore this issue has not been thoroughly investigated.

4.5 Execution of get () and put () operations

Any storage node in Dynamo is eligible to receive client get and put operations for any key. In this section, for sake of simplicity, we describe how these operations are performed in a failure-free environment and in the subsequent section we describe how read and write operations are executed during failures.

Both get and put operations are invoked using Amazon's infrastructure-specific request processing framework over HTTP. There are two strategies that a client can use to select a node: (1) route its request through a generic load balancer that will select a node based on load information, or (2) use a partition-aware client library that routes requests directly to the appropriate coordinator nodes. The advantage of the first approach is that the client does not have to link any code specific to Dynamo in its application, whereas the second strategy can achieve lower latency because it skips a potential forwarding step.

A node handling a read or write operation is known as the *coordinator*. Typically, this is the first among the top N nodes in the preference list. If the requests are received through a load balancer, requests to access a key may be routed to any random node in the ring. In this scenario, the node that receives the request will not coordinate it if the node is not in the top N of the requested key's preference list. Instead, that node will forward the request to the first among the top N nodes in the preference list.

Read and write operations involve the first N healthy nodes in the preference list, skipping over those that are down or inaccessible. When all nodes are healthy, the top N nodes in a key's preference list are accessed. When there are node failures or network partitions, nodes that are lower ranked in the preference list are accessed.

To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems. This protocol has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that R + W > N yields a quorum-like system. In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Upon receiving a put() request for a key, the coordinator generates the vector clock for the new version and writes the new version locally. The coordinator then sends the new version (along with

the new vector clock) to the N highest-ranked reachable nodes. If at least W-1 nodes respond then the write is considered successful.

Similarly, for a get() request, the coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key, and then waits for R responses before returning the result to the client. If the coordinator ends up gathering multiple versions of the data, it returns all the versions it deems to be causally unrelated. The divergent versions are then reconciled and the reconciled version superseding the current versions is written back.

4.6 Handling Failures: Hinted Handoff

If Dynamo used a traditional quorum approach it would be unavailable during server failures and network partitions, and would have reduced durability even under the simplest of failure conditions. To remedy this it does not enforce strict quorum membership and instead it uses a "sloppy quorum"; all read and write operations are performed on the first N *healthy* nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring.

Consider the example of Dynamo configuration given in Figure 2 with N=3. In this example, if node A is temporarily down or unreachable during a write operation then a replica that would normally have lived on A will now be sent to node D. This is done to maintain the desired availability and durability guarantees. The replica sent to D will have a hint in its metadata that suggests which node was the intended recipient of the replica (in this case A). Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically. Upon detecting that A has recovered, D will attempt to deliver the replica to A. Once the transfer succeeds, D may delete the object from its local store without decreasing the total number of replicas in the system.

Using hinted handoff, Dynamo ensures that the read and write operations are not failed due to temporary node or network failures. Applications that need the highest level of availability can set W to 1, which ensures that a write is accepted as long as a single node in the system has durably written the key it to its local store. Thus, the write request is only rejected if all nodes in the system are unavailable. However, in practice, most Amazon services in production set a higher W to meet the desired level of durability. A more detailed discussion of configuring N, R and W follows in section 6.

It is imperative that a highly available storage system be capable of handling the failure of an entire data center(s). Data center failures happen due to power outages, cooling failures, network failures, and natural disasters. Dynamo is configured such that each object is replicated across multiple data centers. In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers. These datacenters are connected through high speed network links. This scheme of replicating across multiple datacenters allows us to handle entire data center failures without a data outage.

4.7 Handling permanent failures: Replica synchronization

Hinted handoff works best if the system membership churn is low and node failures are transient. There are scenarios under which hinted replicas become unavailable before they can be returned to the original replica node. To handle this and other threats to durability, Dynamo implements an anti-entropy (replica synchronization) protocol to keep the replicas synchronized.

To detect the inconsistencies between replicas faster and to minimize the amount of transferred data, Dynamo uses Merkle trees [13]. A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children. The principal advantage of Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the entire data set. Moreover, Merkle trees help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas. For instance, if the hash values of the root of two trees are equal, then the values of the leaf nodes in the tree are equal and the nodes require no synchronization. If not, it implies that the values of some replicas are different. In such cases, the nodes may exchange the hash values of children and the process continues until it reaches the leaves of the trees, at which point the hosts can identify the keys that are "out of sync". Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during the anti-entropy process.

Dynamo uses Merkle trees for anti-entropy as follows: Each node maintains a separate Merkle tree for each key range (the set of keys covered by a virtual node) it hosts. This allows nodes to compare whether the keys within a key range are up-to-date. In this scheme, two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common. Subsequently, using the tree traversal scheme described above the nodes determine if they have any differences and perform the appropriate synchronization action. The disadvantage with this scheme is that many key ranges change when a node joins or leaves the system thereby requiring the tree(s) to be recalculated. This issue is addressed, however, by the refined partitioning scheme described in Section 6.2.

4.8 Membership and Failure Detection

4.8.1 Ring Membership

In Amazon's environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. A node outage rarely signifies a permanent departure and therefore should not result in rebalancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Dynamo nodes. For these reasons, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring. An administrator uses a command line tool or a browser to connect to a Dynamo node and issue a membership change to join a node to a ring or remove a node from a ring. The node that serves the request writes the membership change and its time of issue to persistent store. The membership changes form a history because nodes can be removed and added back multiple times. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership. Each node contacts a peer chosen at random every second and the two nodes efficiently reconcile their persisted membership change histories.

When a node starts for the first time, it chooses its set of tokens (virtual nodes in the consistent hash space) and maps nodes to their respective token sets. The mapping is persisted on disk and

initially contains only the local node and token set. The mappings stored at different Dynamo nodes are reconciled during the same communication exchange that reconciles the membership change histories. Therefore, partitioning and placement information also propagates via the gossip-based protocol and each storage node is aware of the token ranges handled by its peers. This allows each node to forward a key's read/write operations to the right set of nodes directly.

4.8.2   External Discovery

The mechanism described above could temporarily result in a logically partitioned Dynamo ring. For example, the administrator could contact node A to join A to the ring, then contact node B to join B to the ring. In this scenario, nodes A and B would each consider itself a member of the ring, yet neither would be immediately aware of the other. To prevent logical partitions, some Dynamo nodes play the role of seeds. Seeds are nodes that are discovered via an external mechanism and are known to all nodes. Because all nodes eventually reconcile their membership with a seed, logical partitions are highly unlikely. Seeds can be obtained either from static configuration or from a configuration service. Typically seeds are fully functional nodes in the Dynamo ring.

4.8.3   Failure Detection

Failure detection in Dynamo is used to avoid attempts to communicate with unreachable peers during get() and put() operations and when transferring partitions and hinted replicas. For the purpose of avoiding failed attempts at communication, a purely local notion of failure detection is entirely sufficient: node A may consider node B failed if node B does not respond to node A's messages (even if B is responsive to node C's messages). In the presence of a steady rate of client requests generating inter-node communication in the Dynamo ring, a node A quickly discovers that a node B is unresponsive when B fails to respond to a message; Node A then uses alternate nodes to service requests that map to B's partitions; A periodically retries B to check for the latter's recovery. In the absence of client requests to drive traffic between two nodes, neither node really needs to know whether the other is reachable and responsive.

Decentralized failure detection protocols use a simple gossip-style protocol that enable each node in the system to learn about the arrival (or departure) of other nodes. For detailed information on decentralized failure detectors and the parameters affecting their accuracy, the interested reader is referred to [8]. Early designs of Dynamo used a decentralized failure detector to maintain a globally consistent view of failure state. Later it was determined that the explicit node join and leave methods obviates the need for a global view of failure state. This is because nodes are notified of permanent node additions and removals by the explicit node join and leave methods and temporary node failures are detected by the individual nodes when they fail to communicate with others (while forwarding requests).

4.9   Adding/Removing Storage Nodes

When a new node (say X) is added into the system, it gets assigned a number of tokens that are randomly scattered on the ring. For every key range that is assigned to node X, there may be a number of nodes (less than or equal to N) that are currently in charge of handling keys that fall within its token range. Due to the allocation of key ranges to X, some existing nodes no longer have to some of their keys and these nodes transfer those keys to X. Let us consider a simple bootstrapping scenario where node X is added to the ring shown in Figure 2 between A and B. When X is added to the system, it is in charge of storing keys in the ranges (F, G], (G, A] and (A, X]. As a consequence, nodes B, C and D no longer have to store the keys in these respective ranges. Therefore, nodes B, C, and D will offer to and upon confirmation from X transfer the appropriate set of keys. When a node is removed from the system, the reallocation of keys happens in a reverse process.

Operational experience has shown that this approach distributes the load of key distribution uniformly across the storage nodes, which is important to meet the latency requirements and to ensure fast bootstrapping. Finally, by adding a confirmation round between the source and the destination, it is made sure that the destination node does not receive any duplicate transfers for a given key range.

5.   IMPLEMENTATION

In Dynamo, each storage node has three main software components: request coordination, membership and failure detection, and a local persistence engine. All these components are implemented in Java.

Dynamo's local persistence component allows for different storage engines to be plugged in. Engines that are in use are Berkeley Database (BDB) Transactional Data Store[2], BDB Java Edition, MySQL, and an in-memory buffer with persistent backing store. The main reason for designing a pluggable persistence component is to choose the storage engine best suited for an application's access patterns. For instance, BDB can handle objects typically in the order of tens of kilobytes whereas MySQL can handle objects of larger sizes. Applications choose Dynamo's local persistence engine based on their object size distribution. The majority of Dynamo's production instances use BDB Transactional Data Store.

The request coordination component is built on top of an event-driven messaging substrate where the message processing pipeline is split into multiple stages similar to the SEDA architecture [24]. All communications are implemented using Java NIO channels. The coordinator executes the read and write requests on behalf of clients by collecting data from one or more nodes (in the case of reads) or storing data at one or more nodes (for writes). Each client request results in the creation of a state machine on the node that received the client request. The state machine contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies and packaging the response to the client. Each state machine instance handles exactly one client request. For instance, a read operation implements the following state machine: (i) send read requests to the nodes, (ii) wait for minimum number of required responses, (iii) if too few replies were received within a given time bound, fail the request, (iv) otherwise gather all the data versions and determine the ones to be returned and (v) if versioning is enabled, perform syntactic reconciliation and generate an opaque write context that contains the vector clock that subsumes all the remaining versions. For the sake of brevity the failure handling and retry states are left out.

After the read response has been returned to the caller the state

---

[2] http://www.oracle.com/database/berkeley-db.html
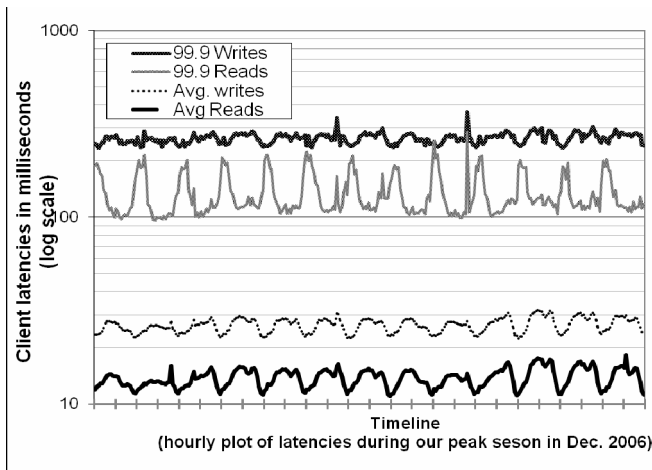
Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

machine waits for a small period of time to receive any outstanding responses. If stale versions were returned in any of the responses, the coordinator updates those nodes with the latest version. This process is called *read repair* because it repairs replicas that have missed a recent update at an opportunistic time and relieves the anti-entropy protocol from having to do it.

As noted earlier, write requests are coordinated by one of the top N nodes in the preference list. Although it is desirable always to have the first node among the top N to coordinate the writes thereby serializing all writes at a single location, this approach has led to uneven load distribution resulting in SLA violations. This is because the request load is not uniformly distributed across objects. To counter this, any of the top N nodes in the preference list is allowed to coordinate the writes. In particular, since each write usually follows a read operation, the coordinator for a write is chosen to be the node that replied fastest to the previous read operation which is stored in the context information of the request. This optimization enables us to pick the node that has the data that was read by the preceding read operation thereby increasing the chances of getting "read-your-writes" consistency. It also reduces variability in the performance of the request handling which improves the performance at the 99.9 percentile.

6. EXPERIENCES & LESSONS LEARNED

Dynamo is used by several services with different configurations. These instances differ by their version reconciliation logic, and read/write quorum characteristics. The following are the main patterns in which Dynamo is used:

- *Business logic specific reconciliation:* This is a popular use case for Dynamo. Each data object is replicated across multiple nodes. In case of divergent versions, the client application performs its own reconciliation logic. The shopping cart service discussed earlier is a prime example of this category. Its business logic reconciles objects by merging different versions of a customer's shopping cart.
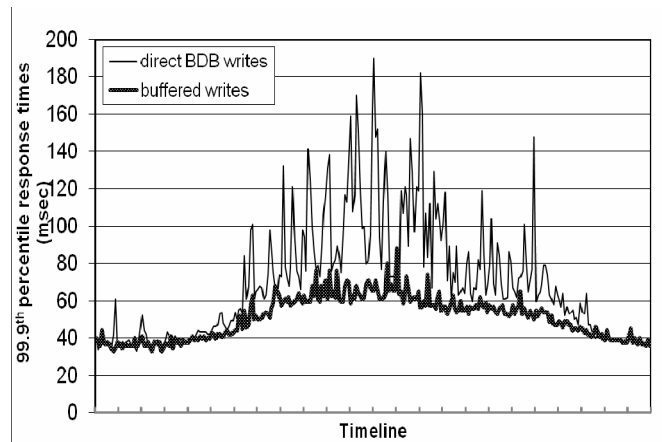
- *Timestamp based reconciliation:* This case differs from the previous one only in the reconciliation mechanism. In case of divergent versions, Dynamo performs simple timestamp based reconciliation logic of "last write wins"; i.e., the object with the largest physical timestamp value is chosen as the correct version. The service that maintains customer's session information is a good example of a service that uses this mode.

- *High performance read engine:* While Dynamo is built to be an "always writeable" data store, a few services are tuning its quorum characteristics and using it as a high performance read engine. Typically, these services have a high read request rate and only a small number of updates. In this configuration, typically R is set to be 1 and W to be N. For these services, Dynamo provides the ability to partition and replicate their data across multiple nodes thereby offering incremental scalability. Some of these instances function as the authoritative persistence cache for data stored in more heavy weight backing stores. Services that maintain product catalog and promotional items fit in this category.

The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability. For instance, the value of N determines the durability of each object. A typical value of N used by Dynamo's users is 3.

The values of W and R impact object availability, durability and consistency. For instance, if W is set to 1, then the system will never reject a write request as long as there is at least one node in the system that can successfully process a write request. However, low values of W and R can increase the risk of inconsistency as write requests are deemed successful and returned to the clients even if they are not processed by a majority of the replicas. This also introduces a vulnerability window for durability when a write request is successfully returned to the client even though it has been persisted at only a small number of nodes.

214
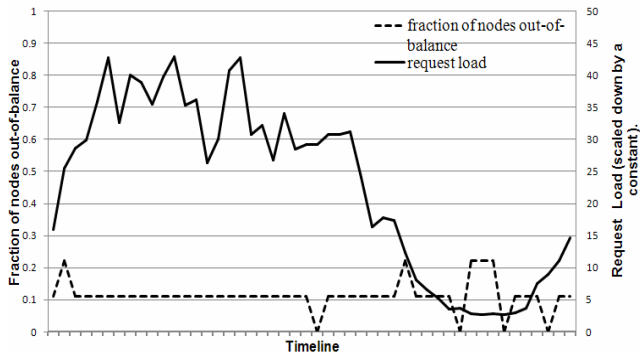
Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

Traditional wisdom holds that durability and availability go hand-in-hand. However, this is not necessarily true here. For instance, the vulnerability window for durability can be decreased by increasing W. This may increase the probability of rejecting requests (thereby decreasing availability) because more storage hosts need to be alive to process a write request.

The common (N,R,W) configuration used by several instances of Dynamo is (3,2,2). These values are chosen to meet the necessary levels of performance, durability, consistency, and availability SLAs.

All the measurements presented in this section were taken on a live system operating with a configuration of (3,2,2) and running a couple hundred nodes with homogenous hardware configurations. As mentioned earlier, each instance of Dynamo contains nodes that are located in multiple datacenters. These datacenters are typically connected through high speed network links. Recall that to generate a successful get (or put) response R (or W) nodes need to respond to the coordinator. Clearly, the network latencies between datacenters affect the response time and the nodes (and their datacenter locations) are chosen such that the applications target SLAs are met.

6.1   Balancing Performance and Durability

While Dynamo's principle design goal is to build a highly available data store, performance is an equally important criterion in Amazon's platform. As noted earlier, to provide a consistent customer experience, Amazon's services set their performance targets at higher percentiles (such as the 99.9[th] or 99.99[th] percentiles). A typical SLA required of services that use Dynamo is that 99.9% of the read and write requests execute within 300ms.

Since Dynamo is run on standard commodity hardware components that have far less I/O throughput than high-end enterprise servers, providing consistently high performance for read and write operations is a non-trivial task. The involvement of multiple storage nodes in read and write operations makes it even more challenging, since the performance of these operations is limited by the slowest of the R or W replicas. Figure 4 shows the average and 99.9[th] percentile latencies of Dynamo's read and write operations during a period of 30 days. As seen in the figure, the latencies exhibit a clear diurnal pattern which is a result of the diurnal pattern in the incoming request rate (i.e., there is a

significant difference in request rate between the daytime and night). Moreover, the write latencies are higher than read latencies obviously because write operations always results in disk access. Also, the 99.9[th] percentile latencies are around 200 ms and are an order of magnitude higher than the averages. This is because the 99.9[th] percentile latencies are affected by several factors such as variability in request load, object sizes, and locality patterns.

While this level of performance is acceptable for a number of services, a few customer-facing services required higher levels of performance. For these services, Dynamo provides the ability to trade-off durability guarantees for performance. In the optimization each storage node maintains an object buffer in its main memory. Each write operation is stored in the buffer and gets periodically written to storage by a *writer thread*. In this scheme, read operations first check if the requested key is present in the buffer. If so, the object is read from the buffer instead of the storage engine.

This optimization has resulted in lowering the 99.9[th] percentile latency by a factor of 5 during peak traffic even for a very small buffer of a thousand objects (see Figure 5). Also, as seen in the figure, write buffering smoothes out higher percentile latencies. Obviously, this scheme trades durability for performance. In this scheme, a server crash can result in missing writes that were queued up in the buffer. To reduce the durability risk, the write operation is refined to have the coordinator choose one out of the N replicas to perform a "durable write". Since the coordinator waits only for W responses, the performance of the write operation is not affected by the performance of the durable write operation performed by a single replica.

6.2   Ensuring Uniform Load distribution

Dynamo uses consistent hashing to partition its key space across its replicas and to ensure uniform load distribution. A uniform key distribution can help us achieve uniform load distribution assuming the access distribution of keys is not highly skewed. In particular, Dynamo's design assumes that even where there is a significant skew in the access distribution there are enough keys in the popular end of the distribution so that the load of handling popular keys can be spread across the nodes uniformly through partitioning. This section discusses the load imbalance seen in Dynamo and the impact of different partitioning strategies on load distribution.

To study the load imbalance and its correlation with request load, the total number of requests received by each node was measured for a period of 24 hours - broken down into intervals of 30 minutes. In a given time window, a node is considered to be "in-balance", if the node's request load deviates from the average load by a value a less than a certain threshold (here 15%). Otherwise the node was deemed "out-of-balance". Figure 6 presents the fraction of nodes that are "out-of-balance" (henceforth, "imbalance ratio") during this time period. For reference, the corresponding request load received by the entire system during this time period is also plotted. As seen in the figure, the imbalance ratio decreases with increasing load. For instance, during low loads the imbalance ratio is as high as 20% and during high loads it is close to 10%. Intuitively, this can be explained by the fact that under high loads, a large number of popular keys are accessed and due to uniform distribution of keys the load is evenly distributed. However, during low loads (where load is 1/8th
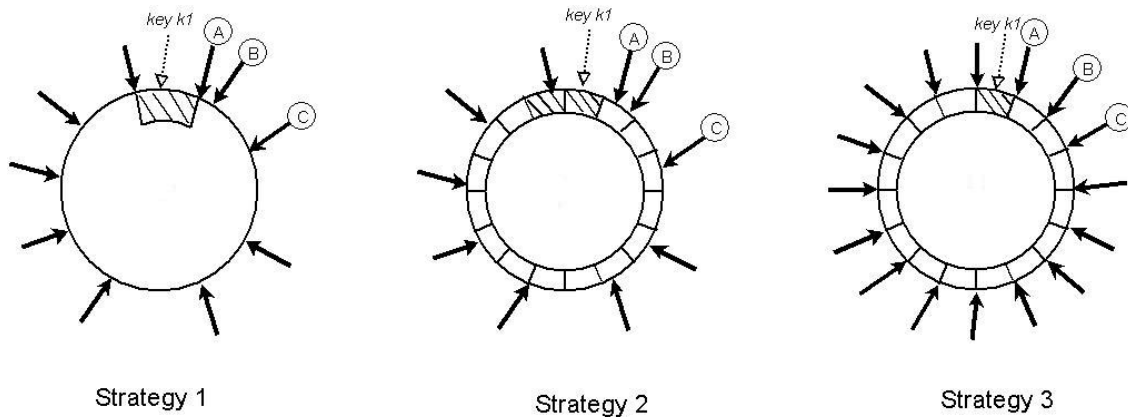
Figure 7: Partitioning and placement of keys in the three strategies. A, B, and C depict the three unique nodes that form the preference list for the key k1 on the consistent hashing ring (N=3). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows indicate the token locations for various nodes.

of the measured peak load), fewer popular keys are accessed, resulting in a higher load imbalance.

This section discusses how Dynamo's partitioning scheme has evolved over time and its implications on load distribution.

Strategy 1: T random tokens per node and partition by token value: This was the initial strategy deployed in production (and described in Section 4.2). In this scheme, each node is assigned T tokens (chosen uniformly at random from the hash space). The tokens of all nodes are ordered according to their values in the hash space. Every two consecutive tokens define a range. The last token and the first token form a range that "wraps" around from the highest value to the lowest value in the hash space. Because the tokens are chosen randomly, the ranges vary in size. As nodes join and leave the system, the token set changes and consequently the ranges change. Note that the space needed to maintain the membership at each node increases linearly with the number of nodes in the system.

While using this strategy, the following problems were encountered. First, when a new node joins the system, it needs to "steal" its key ranges from other nodes. However, the nodes handing the key ranges off to the new node have to scan their local persistence store to retrieve the appropriate set of data items. Note that performing such a scan operation on a production node is tricky as scans are highly resource intensive operations and they need to be executed in the background without affecting the customer performance. This requires us to run the bootstrapping task at the lowest priority. However, this significantly slows the bootstrapping process and during busy shopping season, when the nodes are handling millions of requests a day, the bootstrapping has taken almost a day to complete. Second, when a node joins/leaves the system, the key ranges handled by many nodes change and the Merkle trees for the new ranges need to be recalculated, which is a non-trivial operation to perform on a production system. Finally, there was no easy way to take a snapshot of the entire key space due to the randomness in key ranges, and this made the process of archival complicated. In this scheme, archiving the entire key space requires us to retrieve the keys from each node separately, which is highly inefficient.

The fundamental issue with this strategy is that the schemes for data partitioning and data placement are intertwined. For instance, in some cases, it is preferred to add more nodes to the system in order to handle an increase in request load. However, in this scenario, it is not possible to add nodes without affecting data partitioning. Ideally, it is desirable to use independent schemes for partitioning and placement. To this end, following strategies were evaluated:

Strategy 2: T random tokens per node and equal sized partitions: In this strategy, the hash space is divided into Q equally sized partitions/ranges and each node is assigned T random tokens. Q is usually set such that $Q \gg N$ and $Q \gg S*T$, where S is the number of nodes in the system. In this strategy, the tokens are only used to build the function that maps values in the hash space to the ordered lists of nodes and not to decide the partitioning. A partition is placed on the first N unique nodes that are encountered while walking the consistent hashing ring clockwise from the end of the partition. Figure 7 illustrates this strategy for N=3. In this example, nodes A, B, C are encountered while walking the ring from the end of the partition that contains key k1. The primary advantages of this strategy are: (i) decoupling of partitioning and partition placement, and (ii) enabling the possibility of changing the placement scheme at runtime.

Strategy 3: Q/S tokens per node, equal-sized partitions: Similar to strategy 2, this strategy divides the hash space into Q equally sized partitions and the placement of partition is decoupled from the partitioning scheme. Moreover, each node is assigned Q/S tokens where S is the number of nodes in the system. When a node leaves the system, its tokens are randomly distributed to the remaining nodes such that these properties are preserved. Similarly, when a node joins the system it "steals" tokens from nodes in the system in a way that preserves these properties.

The efficiency of these three strategies is evaluated for a system with S=30 and N=3. However, comparing these different strategies in a fair manner is hard as different strategies have different configurations to tune their efficiency. For instance, the load distribution property of strategy 1 depends on the number of tokens (i.e., T) while strategy 3 depends on the number of partitions (i.e., Q). One fair way to compare these strategies is to
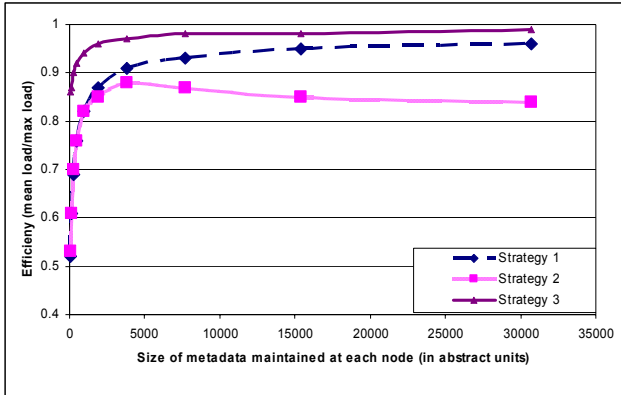
Figure 8: Comparison of the load distribution efficiency of different strategies for system with 30 nodes and N=3 with equal amount of metadata maintained at each node. The values of the system size and number of replicas are based on the typical configuration deployed for majority of our services.

evaluate the skew in their load distribution while all strategies use the same amount of space to maintain their membership information. For instance, in strategy 1 each node needs to maintain the token positions of all the nodes in the ring and in strategy 3 each node needs to maintain the information regarding the partitions assigned to each node.

In our next experiment, these strategies were evaluated by varying the relevant parameters (T and Q). The load balancing efficiency of each strategy was measured for different sizes of membership information that needs to be maintained at each node, where *Load balancing efficiency* is defined as the ratio of average number of requests served by each node to the maximum number of requests served by the hottest node.

The results are given in Figure 8. As seen in the figure, strategy 3 achieves the best load balancing efficiency and strategy 2 has the worst load balancing efficiency. For a brief time, Strategy 2 served as an interim setup during the process of migrating Dynamo instances from using Strategy 1 to Strategy 3. Compared to Strategy 1, Strategy 3 achieves better efficiency and reduces the size of membership information maintained at each node by three orders of magnitude. While storage is not a major issue the nodes gossip the membership information periodically and as such it is desirable to keep this information as compact as possible. In addition to this, strategy 3 is advantageous and simpler to deploy for the following reasons: (i) *Faster bootstrapping/recovery:* Since partition ranges are fixed, they can be stored in separate files, meaning a partition can be relocated as a unit by simply transferring the file (avoiding random accesses needed to locate specific items). This simplifies the process of bootstrapping and recovery. (ii) *Ease of archival*: Periodical archiving of the dataset is a mandatory requirement for most of Amazon storage services. Archiving the entire dataset stored by Dynamo is simpler in strategy 3 because the partition files can be archived separately. By contrast, in Strategy 1, the tokens are chosen randomly and, archiving the data stored in Dynamo requires retrieving the keys from individual nodes separately and is usually inefficient and slow. The disadvantage of strategy 3 is that changing the node membership requires coordination in order to preserve the properties required of the assignment.

6.3 Divergent Versions: When and How Many?

As noted earlier, Dynamo is designed to tradeoff consistency for availability. To understand the precise impact of different failures on consistency, detailed data is required on multiple factors: outage length, type of failure, component reliability, workload etc. Presenting these numbers in detail is outside of the scope of this paper. However, this section discusses a good summary metric: the number of divergent versions seen by the application in a live production environment.

Divergent versions of a data item arise in two scenarios. The first is when the system is facing failure scenarios such as node failures, data center failures, and network partitions. The second is when the system is handling a large number of concurrent writers to a single data item and multiple nodes end up coordinating the updates concurrently. From both a usability and efficiency perspective, it is preferred to keep the number of divergent versions at any given time as low as possible. If the versions cannot be syntactically reconciled based on vector clocks alone, they have to be passed to the business logic for semantic reconciliation. Semantic reconciliation introduces additional load on services, so it is desirable to minimize the need for it.

In our next experiment, the number of versions returned to the shopping cart service was profiled for a period of 24 hours. During this period, 99.94% of requests saw exactly one version; 0.00057% of requests saw 2 versions; 0.00047% of requests saw 3 versions and 0.00009% of requests saw 4 versions. This shows that divergent versions are created rarely.

Experience shows that the increase in the number of divergent versions is contributed not by failures but due to the increase in number of concurrent writers. The increase in the number of concurrent writes is usually triggered by busy robots (automated client programs) and rarely by humans. This issue is not discussed in detail due to the sensitive nature of the story.

6.4 Client-driven or Server-driven Coordination

As mentioned in Section 5, Dynamo has a request coordination component that uses a state machine to handle incoming requests. Client requests are uniformly assigned to nodes in the ring by a load balancer. Any Dynamo node can act as a coordinator for a read request. Write requests on the other hand will be coordinated by a node in the key's current preference list. This restriction is due to the fact that these preferred nodes have the added responsibility of creating a new version stamp that causally subsumes the version that has been updated by the write request. Note that if Dynamo's versioning scheme is based on physical timestamps, any node can coordinate a write request.

An alternative approach to request coordination is to move the state machine to the client nodes. In this scheme client applications use a library to perform request coordination locally. A client periodically picks a random Dynamo node and downloads its current view of Dynamo membership state. Using this information the client can determine which set of nodes form the preference list for any given key. Read requests can be coordinated at the client node thereby avoiding the extra network hop that is incurred if the request were assigned to a random Dynamo node by the load balancer. Writes will either be forwarded to a node in the key's preference list or can be

Table 2: Performance of client-driven and server-driven coordination approaches.

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

coordinated locally if Dynamo is using timestamps based versioning.

An important advantage of the client-driven coordination approach is that a load balancer is no longer required to uniformly distribute client load. Fair load distribution is implicitly guaranteed by the near uniform assignment of keys to the storage nodes. Obviously, the efficiency of this scheme is dependent on how fresh the membership information is at the client. Currently clients poll a random Dynamo node every 10 seconds for membership updates. A pull based approach was chosen over a push based one as the former scales better with large number of clients and requires very little state to be maintained at servers regarding clients. However, in the worst case the client can be exposed to stale membership for duration of 10 seconds. In case, if the client detects its membership table is stale (for instance, when some members are unreachable), it will immediately refresh its membership information.

Table 2 shows the latency improvements at the 99.9th percentile and averages that were observed for a period of 24 hours using client-driven coordination compared to the server-driven approach. As seen in the table, the client-driven coordination approach reduces the latencies by at least 30 milliseconds for 99.9th percentile latencies and decreases the average by 3 to 4 milliseconds. The latency improvement is because the client-driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node. As seen in the table, average latencies tend to be significantly lower than latencies at the 99.9th percentile. This is because Dynamo's storage engine caches and write buffer have good hit ratios. Moreover, since the load balancers and network introduce additional variability to the response time, the gain in response time is higher for the 99.9th percentile than the average.

6.5 Balancing background vs. foreground tasks

Each node performs different kinds of background tasks for replica synchronization and data handoff (either due to hinting or adding/removing nodes) in addition to its normal foreground put/get operations. In early production settings, these background tasks triggered the problem of resource contention and affected the performance of the regular put and get operations. Hence, it became necessary to ensure that background tasks ran only when the regular critical operations are not affected significantly. To this end, the background tasks were integrated with an admission control mechanism. Each of the background tasks uses this controller to reserve runtime slices of the resource (e.g. database),

shared across all background tasks. A feedback mechanism based on the monitored performance of the foreground tasks is employed to change the number of slices that are available to the background tasks.

The admission controller constantly monitors the behavior of resource accesses while executing a "foreground" put/get operation. Monitored aspects include latencies for disk operations, failed database accesses due to lock-contention and transaction timeouts, and request queue wait times. This information is used to check whether the percentiles of latencies (or failures) in a given trailing time window are close to a desired threshold. For example, the background controller checks to see how close the 99th percentile database read latency (over the last 60 seconds) is to a preset threshold (say 50ms). The controller uses such comparisons to assess the resource availability for the foreground operations. Subsequently, it decides on how many time slices will be available to background tasks, thereby using the feedback loop to limit the intrusiveness of the background activities. Note that a similar problem of managing background tasks has been studied in [4].

6.6 Discussion

This section summarizes some of the experiences gained during the process of implementation and maintenance of Dynamo. Many Amazon internal services have used Dynamo for the past two years and it has provided significant levels of availability to its applications. In particular, applications have received successful responses (without timing out) for 99.9995% of its requests and no data loss event has occurred to date.

Moreover, the primary advantage of Dynamo is that it provides the necessary knobs using the three parameters of (N,R,W) to tune their instance based on their needs.. Unlike popular commercial data stores, Dynamo exposes data consistency and reconciliation logic issues to the developers. At the outset, one may expect the application logic to become more complex. However, historically, Amazon's platform is built for high availability and many applications are designed to handle different failure modes and inconsistencies that may arise. Hence, porting such applications to use Dynamo was a relatively simple task. For new applications that want to use Dynamo, some analysis is required during the initial stages of the development to pick the right conflict resolution mechanisms that meet the business case appropriately. Finally, Dynamo adopts a full membership model where each node is aware of the data hosted by its peers. To do this, each node actively gossips the full routing table with other nodes in the system. This model works well for a system that contains couple of hundreds of nodes. However, scaling such a design to run with tens of thousands of nodes is not trivial because the overhead in maintaining the routing table increases with the system size. This limitation might be overcome by introducing hierarchical extensions to Dynamo. Also, note that this problem is actively addressed by O(1) DHT systems(e.g., [14]).

7. CONCLUSIONS

This paper described Dynamo, a highly available and scalable data store, used for storing state of a number of core services of Amazon.com's e-commerce platform. Dynamo has provided the desired levels of availability and performance and has been successful in handling server failures, data center failures and network partitions. Dynamo is incrementally scalable and allows service owners to scale up and down based on their current

request load. Dynamo allows service owners to customize their storage system to meet their desired performance, durability and consistency SLAs by allowing them to tune the parameters N, R, and W.

The production use of Dynamo for the past year demonstrates that decentralized techniques can be combined to provide a single highly-available system. Its success in one of the most challenging application environments shows that an eventual-consistent storage system can be a building block for highly-available applications.

ACKNOWLEDGEMENTS

REFERENCES

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

[2] Bernstein, P.A., and Goodman, N. An algorithm for concurrency control and recovery in replicated distributed databases. ACM Trans. on Database Systems, 9(4):596-615, December 1984

[3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA, November 06 - 08, 2006). USENIX Association, Berkeley, CA, 15-15.

[4] Douceur, J. R. and Bolosky, W. J. 2000. Process-based regulation of low-importance processes. *SIGOPS Oper. Syst. Rev.* 34, 2 (Apr. 2000), 26-27.

[5] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France, October 05 - 08, 1997). W. M. Waite, Ed. SOSP '97. ACM Press, New York, NY, 78-91.

[6] Ghemawat, S., Gobioff, H., and Leung, S. 2003. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM Press, New York, NY, 29-43.

[7] Gray, J., Helland, P., O'Neil, P., and Shasha, D. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data* (Montreal, Quebec, Canada, June 04 - 06, 1996). J. Widom, Ed. SIGMOD '96. ACM Press, New York, NY, 173-182.

[8] Gupta, I., Chandra, T. D., and Goldszmidt, G. S. 2001. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, Rhode Island, United States). PODC '01. ACM Press, New York, NY, 170-179.

[9] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. 2000. OceanStore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News* 28, 5 (Dec. 2000), 190-201.

[10] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing* (El Paso, Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663.

[11] Lindsay, B.G., et. al., "Notes on Distributed Databases", Research Report RJ2571(33471), IBM Research, July 1979

[12] Lamport, L. Time, clocks, and the ordering of events in a distributed system. ACM Communications, 21(7), pp. 558-565, 1978.

[13] Merkle, R. A digital signature based on a conventional encryption function. Proceedings of CRYPTO, pages 369–378. Springer-Verlag, 1988.

[14] Ramasubramanian, V., and Sirer, E. G. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation,* San Francisco, CA, March 29 - 31, 2004.

[15] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. 1994. Resolving file conflicts in the Ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (Boston, Massachusetts, June 06 - 10, 1994). USENIX Association, Berkeley, CA, 12-12..

[16] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Proceedings of Middleware, pages 329-350, November, 2001.

[17] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. Proceedings of Symposium on Operating Systems Principles, October 2001.

[18] Saito, Y., Frølund, S., Veitch, A., Merchant, A., and Spence, S. 2004. FAB: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.* 38, 5 (Dec. 2004), 48-58.

[19] Satyanarayanan, M., Kistler, J.J., Siegel, E.H. Coda: A Resilient Distributed File System. IEEE Workshop on Workstation Operating Systems, Nov. 1987.

[20] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California, United States). SIGCOMM '01. ACM Press, New York, NY, 149-160.

[21] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, United States, December 03 - 06, 1995). M. B. Jones, Ed. SOSP '95. ACM Press, New York, NY, 172-182.

[22] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems 4 (2): 180-209, 1979.

[23] Weatherspoon, H., Eaton, P., Chun, B., and Kubiatowicz, J. 2007. Antiquity: exploiting a secure log for wide-area distributed storage. *SIGOPS Oper. Syst. Rev.* 41, 3 (Jun. 2007), 371-384.

[24] Welsh, M., Culler, D., and Brewer, E. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada, October 21 - 24, 2001). SOSP '01. ACM Press, New York, NY, 230-243.