Kinesis: A New Approach to Replica Placement in Distributed Storage Systems

JOHN MACCORMICK
Department of Mathematics and Computer Science, Dickinson College, Carlisle, PA 17013
and
NICHOLAS MURPHY
Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195,
and
VENUGOPALAN RAMASUBRAMANIAN
Microsoft Research Silicon Valley, Mountain View, CA 94043
and
UDI WIEDER
Microsoft Research Silicon Valley, Mountain View, CA 94043
and
JUNFENG YANG
Department of Computer Science, Stanford University, Stanford, CA 94305
and
LIDONG ZHOU
Microsoft Research Silicon Valley, Mountain View, CA 94043

Kinesis is a novel data placement model for distributed storage systems. It exemplifies three design principles: `structure` (division of servers into a few failure-isolated segments), `freedom of choice` (freedom to allocate the best servers to store and retrieve data based on current resource availability), and `scattered distribution` (independent, pseudo-random spread of replicas in the system). These design principles enable storage systems to achieve balanced utilization of storage and network resources in the presence of incremental system expansions, failures of single and shared components, and skewed distributions of data size and popularity. In turn, this ability leads to significantly reduced resource provisioning costs, good user-perceived response times, and fast, parallelized recovery from independent and correlated failures.

This paper validates Kinesis through theoretical analysis, simulations, and experiments on a prototype implementation. Evaluations driven by real-world traces show that Kinesis can significantly out-perform the widely-used *Chain* replica-placement strategy; Kinesis saves resource requirements by 25%, improves end-to-end delay by 40%, and recovers from failures orders of magnitude times faster than Chain in our evaluations.

Categories and Subject Descriptors: []:

General Terms:

Additional Key Words and Phrases:

---

1. INTRODUCTION

The design of distributed storage systems involves tradeoffs between three qualities: (i) performance (serving the workload responsively); (ii) scalability (handling increases in workload); (iii) availability and reliability (serving workload continuously without losing data). Imbalance in resource usage often becomes a significant deterrent for achieving these goals. In a poorly balanced system, part of the system tends to become an unnecessary bottleneck leading to longer delays, reduced system capacity, and and diminished availability.

This paper presents Kinesis, a data placement model for local-area storage systems that achieves balanced use of resources through an elegant scheme based on multiple hash functions. In this strategy, storage servers are partitioned into k disjoint *segments* of approximately equal sizes. Servers whose failures might be correlated (e.g., those on the same rack, on the same network switch or on the same power supply) are assigned to the same segments to minimize inter-segment correlated failure. An independent hash function is associated with each segment and maps identifiers for data items to servers in the segment. The hash function adjusts dynamically to accommodate changes in the set of servers due to failures or additions of servers. For each data item, the k hash functions yield k servers, or k *hash locations*, one in each segment.

Kinesis stores the replicas of a data item on a subset of its k hash locations. In particular, if each data item is replicated r times (where $r < k$), the replicas can be stored on any r of the k hash locations. A read request can be served by any of the r hash locations storing the replicas. A write request creates a new version on r of the k hash locations. The system chooses the "best" r locations—a strategy known as the *multi-choice paradigm* [Azar et al. 1999]—to keep resources as balanced as possible. This simple strategy enables Kinesis to achieve the desirable design goals:

High Performance: Balanced use of resources through the multi-choice paradigm enables storage systems using Kinesis to sustain larger workloads than less-balanced systems, thereby reducing the need for over-provisioning.

Scalability: Balanced use of resources also enables Kinesis-enabled systems to scale by removing imbalance-triggered bottlenecks. Hash-based data placement and lookups eliminate any need to maintain a mapping, removing another potential obstacle to good scalability. Furthermore, the dynamically changing hash functions provide efficient, disruption-free system expansions.

Availability and Reliability: Replication ensures data availability despite failures, while physical isolation of segments mitigates the impact of correlated failures. Moreover, dispersed, randomly-distributed replica layout aids in parallel recovery with uniformly spread-out load for re-creating replicas.

This paper validates these strengths of Kinesis through theoretical analysis, simulations, and experiments using a prototype implementation. Our analysis shows that the multi-choice paradigm of resource balancing is sound in the presence of replication and incremental expansion. Our simulations, based on a 10-year Plan-9 file system trace [Quinlan and Dorward 2002], a 6-month log of MSN Video production servers, and an artificial block-level trace, show that Kinesis substantially reduces the number of servers to be provisioned, cuts the recovery time due to parallel and balanced recovery, and reduces client-perceived delays, when compared to

a commonly-used *Chain* data-placement strategy [Hsiao and DeWitt 1990; Rowstron and Druschel 2001; Dabek et al. 2001], where replicas are placed on adjacent, correlated servers. Experiments on a prototype system further confirm the superior performance of Kinesis.

The key contribution of the paper is the novel Kinesis model for replica placement in distributed storage systems. The paper validates the Kinesis model through experiments on a particular storage system (our prototype). However, the presented model is complementary to existing storage systems: it could be used as the replica placement layer in a wide variety of existing or proposed systems to improve resource balance and reduce cost.

The paper has the following organization: Next section gives an overview of other works related to Kinesis. Section 3 describes the Kinesis data placement model in detail, while Section 4 explains the theory behind Kinesis. Section 5 and Section 6 describe our prototype implementation and present an evaluation of Kinesis respectively. Finally, Section 7 discusses the implications of using Kinesis in practice, and Section 8 concludes.

2. RELATED WORK

Surprisingly, there appears to be no distributed storage system with a Kinesis-like replica placement strategy. In this section, we outline current data and replica placement strategies, which provide many, although not all, benefits of Kinesis.

Global-Mapping: Many distributed storage systems maintain a global mapping for locating data items and replicas often replicated on several or all servers or provided by a replicated state-machine service such as Paxos. GFS [Ghemawat et al. 2003] is a recent example, where a single (replicated) master maps data chunks to servers. GFS takes advantage of the freedom to place replicas on any server in the system and achieves a pseudo-random replica distribution which provides tolerance from correlated failures, balanced resource utilization and fast, parallel failure recovery.

Global mapping offers maximum opportunity for perfect resource balancing and correlation-free data placement. However, this flexibility comes at a high cost for making global decisions and consistently maintaining the map on multiple servers. Kinesis shows that we can harness all these benefits even with limited freedom of choice.

Hashing: Hash-based schemes for data placement eliminate the cost of maintaining global maps by providing a deterministic mapping between data and servers. For example, Archipelago [Ji et al. 2000] uses hashing on the directory path for data placement to increase availability. Peer-to-peer storage systems (such as [Rowstron and Druschel 2001], CFS [Dabek et al. 2001], and OceanStore [Kubiatowicz et al. 2000]) based on Distributed Hash Tables (DHTs) also use hashing for placing and locating objects; these systems use multi-hop routing to locate the objects as they operate over WANs and are fully decentralized.

There are two key issues in hashing-based schemes: (1) how to dynamically adjust the hash function in face of server failures and server additions, and (2) how to achieve load and storage balance.

Hashing schemes that require minimal data relocation during server failures and

expansions include Consistent Hashing [Litwin 1980], Extendable Hashing [Ji et al. 2000], and Linear Hashing [Karger et al. 1997]. The peer-to-peer storage systems above use consistent hashing while Archipelago uses extendable hashing. Linear hashing is an alternative that is well-suited for a LAN environment, where servers are added into the space in a controlled, predetermined fashion (Section 3.1.1. Moreover, linear hashing has inherently less imbalance in the way it allocates the key space between servers compared to consistent hashing. We use linear hashing while evaluating Kinesis in this paper. However, the design principles are equally applicable for other hashing schemes are expected to provide similar benefits.

An important concern with hashing is that it provides poor balance in resource utilization. Its imbalance typically corresponds to a single-choice paradigm (see Section 4) and increases with the number of data items. One well-known technique for load balance servers is proposed in [Dabek et al. 2001; Godfrey et al. 2004], where each server emulates $\log n$ *virtual servers* by holding $\log n$ small segments. Resource balance is then achieved when highly loaded servers pass virtual servers to lightly loaded ones. However, this approach tends to induce correlated failures (as multiple virtual servers map to one physical machine, and the mapping is not exposed to the application) and requires additional maintenance overhead.

Byers *et al.* [Byers et al. 2003] analyze the use of the multi-choice paradigm for storage balancing for DHTs but do not address storage system issues such as replication, failure recovery, and system expansions.

Chaining: A commonly-used replica placement scheme, which we call *chain placement*, first chooses a primary server through any data placement scheme and then places replicas on servers adjacent to the primary. Peer-to-Peer storage systems, such as PAST [Rowstron and Druschel 2001] and CFS [Dabek et al. 2001], as well as LAN stroage systems, such as Chained declustering [Hsiao and DeWitt 1990], Petal [Lee and Thekkath 1996], and Boxwood [MacCormick et al. 2004] use chain placement. Much of this paper discusses the relative merits of Kinesis over the Chain strategy. Simulations and experiments presented in this paper clearly quantify the advantages of Kinesis over Chaining.

Scattering: While chaining distributes replicas in a correlated manner, a recently proposed distributed file system called Ceph [Weil et al. 2006] uses a psuedorandom replica placement strategy. This strategy called CRUSH [Weil et al. 2006] maps a data item to a set of servers on which the replicas are placed in a random yet deterministic manner taking as input cluster organization and server capacities. While Crush provides similar advantages as Kinesis, namely hash-based data placement and tolerance to correlated, shared component failures, it lacks the freedom in placing replicas. Consequently, resource utilization is likely poorly-balanced (similar to single-choice placement discussed in Section 4), and requires data relocation to improve balance in the presence of dynamic changes in server load or spare capacity.

3. KINESIS

We consider systems consisting of a (large) number of back-end *servers*, each with one or more CPUs, local memory, and locally attached disks for storing data. We refer to each logical data unit as an *object*; objects might be of variable sizes.

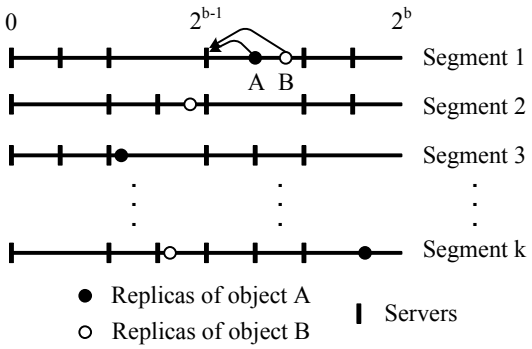


Fig. 1. Kinesis Data Placement Model: Kinesis divides the servers into k segments, employs independent hash functions to map an object to a unique server in each segment, and replicates the object in r servers chosen from the k mapped servers based on resource availability.

There are also one or more *front-ends* that take client requests and distribute the requests to the back-end servers. Both the servers and the front-ends reside in the same administrative domain, connected through one or multiple high-speed network switches in a local area network. Clients of the system can read, create and update objects.

3.1 Hash-Based Organization

Kinesis partitions servers into k disjoint *segments* of approximately the same size, as illustrated in Figure 1. Servers whose failures might be correlated are placed in the same segment when possible, meaning that most single-component failures impact only a single segment.

Within each segment, Kinesis employs a hash function that maps each object to a unique server in the segment; hash functions for different segments are independent. Hash-based data placement eliminates the expense for maintaining large mapping tables by obtaining potential locations of an object through efficient and deterministic computations. The k servers, one in each segment, to which an object is mapped are referred to as the *hash locations* for the object.

For each object, Kinesis creates $r < k$ replicas and places them on r of its k hash locations. The r locations are chosen to minimize resource imbalance following the multiple-choice paradigm, as described in Section 3.2. Because there are $k > r$ hash locations to choose from, the scheme offers freedom. Due to the use of independent hash functions across segments, replicas are distributed in a scattered manner: replicas of objects that belong to the same server on one segment may be stored on different servers in another segment or on completely different segments altogether, as shown in Figure 1.

3.1.1 *Linear Hashing.* A hash function for a segment needs to accommodate the addition of new servers so that some objects are mapped to those servers. As previously discussed, we can use any dynamic hashing scheme, such as linear hashing, extendable hashing, or consistent hashing. These schemes use a hash

function to map objects to a key space and assign portions of the key space to servers. For example, in Figure 1, objects are hashed to a b-bit key space and mapped to the closest server to their left in the key space.

We use the well-known linear hashing scheme because it is deterministic, has low key-space imbalance (bounded by a factor of 2 in the absence of failures), and requires minimal reassignment of objects when servers are added or removed. In contrast, the popular consistent-hashing scheme has a higher imbalance (factor $O(\log n)$ for n servers in key-space allocation).

We describe our precise implementation for concreteness. In linear hashing, a primary, fixed-base hash function h first maps an object d to a unique location in a b-bit key space $[0, 2^b]$, larger than the maximum number of nodes expected in the system. A *secondary* hash function h_e then maps this key to one of the n nodes in the segment as follows:

$$h_e(d) := \begin{cases} h(d) \ mod \ 2^{l+1} & if \ h(d) \ mod \ 2^{l+1} < n \\ h(d) \ mod \ 2^{l} & \text{otherwise} \end{cases}$$

Here, $l = \lfloor log_2(n) \rfloor$.

Intuitively, linear hashing positions new servers deterministically on the key space. Starting with a state with 2^{b-1} servers, it positions the $(2^{b-1} + i)^{th}$ server to split the key space assigned to server i into two equal portions. This splitting continues until there are 2^b servers and then restarts at the next level with 2^b servers.

Figure 2 illustrates linear hashing: the first line shows the addition of servers 0 and 1, where server 1 splits the key space of server 0; the second line shows the further addition of servers 2 and 3, where server 2 splits the key space for server 0, and server 3 splits the key space for server 1; the key space is even when there are 4 servers. The last line shows the addition of servers 4 and 5, where server 4 splits the key space for server 0 and server 5 splits that of server 1. With 6 servers, the key space of the un-split servers, namely servers 2 and 3 are twice as large as the others.

Linear hashing can be extended to handle server failures through a simple strategy: map an object to the closest non-faulty server to its left. For example, if server 2 fails, all objects mapped to 2 will now be mapped to 4. Note that such adjustment could lead to excessive hash-space imbalance. Because failed servers are often repaired or replaced in a data center, and Kinesis has spare hash locations to store extra replicas before the replacement, hash-space imbalance is not a major issue for Kinesis. Moreover, Kinesis' resource balancing algorithms handle any inherent hash-space imbalance.

3.2 Resource-Balanced Request Processing

Kinesis attempts to achieve both *storage balance*, which is concerned with the balanced use of storage space on servers, and *load balance*, which is concerned with the balanced use of other transient resources, such as network bandwidth, disk bandwidth, CPU, and memory, that are utilized when processing client requests.

Storage balance and load balance are crucial to reduce the overall cost for provisioning a distributed storage system. If the system has high utilization but poor

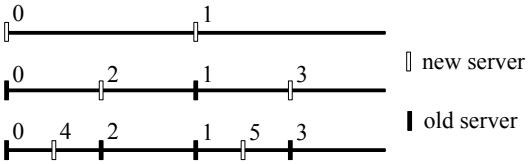


Fig. 2. Key-Space Distribution in Linear Hashing

balance, then some part of the system will become an unnecessary bottleneck and require over-provisioning to remove such bottlenecks. Imbalance in the use of resources may arise due to a number of reasons: inherent imbalance in the key space, high variability in object sizes, and skewed distribution of request rate for popular and unpopular objects all contribute to high imbalance in the resource utilization.

3.2.1 *Read Requests:.* Reads affect only load balance, and a storage system using the Kinesis model for replica placement can use the straightforward approach of picking the least-loaded server from the r replicas storing the requested object.

Even though this is a simple load balancing strategy commonly used in many systems, the dispersed, pseudo-random distribution of replicas in Kinesis provides subtle advantages over other correlated replica placement strategies. In Kinesis, the probability that all replicas of two objects are mapped to the same set of nodes is low; consequently, even if two highly popular objects happen to be stored on the same node, other replicas are likely to be distributed on different nodes, leading to increased opportunities for balancing load.

Server-side caching may influence the performance of read requests under certain circumstances. If the disk bandwidth is lower than the network bandwidth, then server-side caching is an effective technique to improve performance. In the presence of caching, a uniform distribution of read requests to many servers may lead to poorer utilization of the cache as the same object may get cached at different nodes and lead to the waste of memory. A locality-aware request distribution strategy, such as LARD [Pai et al. 1998] or D-SPTF [Lumb et al. 2004], may be more suitable to such systems. However, we do not investigate the effect of caching on load balance as using state-of-the-art disks or combining a sufficient number of disks can usually provide disk bandwidth that is comparable to the network bandwidth on each server.

3.3 Write Requests

Write requests include creating new objects and updating existing objects. When a client issues a request to create an object, Kinesis is free to choose r of the k possible hash locations. Unlike reads, in which only one resource requires balancing, writes require two resources to be balanced: storage and load. We advocate a combined approach called *Kinesis S+L*, which balances load in the short term and storage in the long term by using the following algorithm: choose up to r servers with lowest storage utilization from among those servers with empty I/O queues; if any more servers are required, select those with the shortest I/O queues. It is intuitively clear that Kinesis S+L will keep the servers load-balanced at times of heavy load, and permit storage balance to be regained during periods of low load. Section 6.2.3

confirms this intuition empirically and examines naïve strategies that balance only one of these two resources: a strategy termed *Kinesis S* chooses the r servers with lowest current storage utilization, whereas *Kinesis L* chooses the r servers with shortest I/O queues.

Updates raise the issue of consistency in addition to resource balance. Maintaining consistency while performing updates has been comprehensively studied elsewhere and is not a major theme of this work. We simply mention two simple strategies that could be employed. In the first strategy, version numbers are associated with objects; an update creates a new version for the object while deleting any old one. A server overwrites its current version only when it receives one with a higher version number; including the client id can ensure that version numbers are unique. Polling $k - r + 1$ hash locations is sufficient to encounter a server that processed the last completed update because there are only k hash locations. An appropriate read/write register semantics can thus be provided.

Alternatively, the application uses the Kinesis-enabled storage system as a block store and employs copy-on-write to maintain multiple versions. Updates are thus transformed to creations of new blocks. The application is assumed to maintain metadata that keeps track of the relationship between different versions of data and the associated blocks.

3.4 Failure Recovery

When a server fails, new replicas need to be created for lost replicas in order to reduce the probability of data loss. Note that Kinesis minimizes the impact of correlated failures as replicas are distributed on different segments. Thus, the replicas of an object are unlikely to all become unavailable due to typical causes of correlated failures, such as the failure of a rack's power supply or network switch.

Restoring the replicas on the servers taking over the key space assigned to failed ones is likely to introduce key-space imbalance and overload those servers. Instead, we restore the replicas on the as-yet-unused hash locations; re-creating replicas at spare hash locations preserves the invariant that each replica of a data item is placed at one of its k hash locations, thereby eliminating the need for any bookkeeping and for consistent meta-data updates.

Due to the pseudo-random nature of the hash functions, as well as their independence, any two data items on a failed server are likely to have their remaining replicas and their unused hash locations spread across different servers of the other segments. (Recall that the other hash locations are by definition in other segments.) This leads to fast parallel recovery that involves many different pairs of servers, which has been shown effective in reducing recovery time [Ghemawat et al. 2003; van Renesse and Schneider 2004].

Recreating a replica involves copying data from a *source* server that has a replica to a *destination* server on which a new replica is to be created. This process consumes storage space on the destination server, while imposing load on both the source and the destination servers. Freedom of choice can be exercised for load balance and storage balance during failure recovery: when a replica fails, a suitable source server can be picked from $r - 1$ remaining replicas and a destination server from $k - r$ unused hash locations.

3.5 Incremental Expansion

New machines may be added to the system in order to increase its capacity. Kinesis adds new servers to the segments (logically) in a round-robin fashion so that the sizes of segments remain approximately the same.

Linear hashing accommodates the newly added servers by having them take over half of the key spaces of some existing servers. Objects mapped to the re-assigned key spaces must now be copied to the new servers in order to maintain the invariant that an object is stored only on its hash locations. These relocations consume systems resources and therefore might impact performance at times of high utilization.

Performing relocations in a lazy fashion preferably at times of low system utilization avoids any performance impact. Instead, *location pointers* can be placed at the new location pointing to the new location. A data item can be moved when requested. Cold data items do not have to move at all unless the original server gets close to its capacity. Note that location pointers do not necessarily hamper the availability of the data when servers storing the pointers are unavailable: because of our use of linear hashing, the targets of the pointers are deterministic and easily computable. Therefore, a front-end can always try the servers that could be pointed to and locate the data items if they are indeed stored there.

Incremental expansion might also lead to imbalance in resource utilization. For instance, a storage imbalance is created the moment new servers are added to a segment. Moreover, the presence of fewer objects as well as newer object that might be more popular just because they are newer may lead to load imbalance. Next section discusses scenarios under which this imbalance can be corrected eventually.

4. THEORETICAL UNDERPINNING

Data placement in distributed storage systems is closely related to the classic *balls-and-bins* problem. The balls-and-bins problem involves sequentially placing m balls into n bins. In a storage system, servers correspond to the "bins"; objects are "balls" for storage balancing, whereas for load balancing, read/write requests are "balls".

In this section, we describe the well-known theoretical bounds for the *weight imbalance* between the bins; that is, the difference between the maximum weight of any bin and their average weight, where the weight of a bin is the sum of weights of the balls assigned to it. We extend the bounds to Kinesis and discuss their practical implications.

4.1 Balanced Allocations

For the *single-choice* case, where all balls are of uniform weight and each ball is placed into a bin picked uniformly at random, the weight imbalance is bounded by $\sqrt{m \ln n/n}$ [Berenbrink et al. 2000]. Azar et al. analyzed the weight imbalance for the k-choice scheme (a.k.a. the *multiple-choice paradigm*) with n balls ($m = n$): for each ball, this scheme chooses $k \geq 2$ bins independently and uniformly at random (with replacement) and places the ball in the least full of the k bins at the time of placement breaking ties arbitrarily. They prove that the weight imbalance is bounded by $\ln \ln n / \ln k + O(1)$ with probability $1 - o(1/n)$. Berenbrink *et al.* [Berenbrink et al. 2000] further generalizes this result for $m \gg n$ balls, showing

that for every m, the imbalance is bounded by $\ln n / \ln k + O(1)$.

The key advantage of the multi-choice paradigm is that the weight imbalance is independent of the number of balls m in the system unlike the single-choice case, where the imbalance increases with m. Moreover, Berenbrink *et al.* [Berenbrink et al. 2000] also show that the multi-choice paradigm corrects any existing weight imbalance significantly faster than a single-choice strategy. Wieder [Wieder 2007] extends the above result to the case where bins are not sampled uniformly and have heterogeneous capacity and shows that the imbalance caused by the heterogeneity of the bins is alleviated by increasing the number of choices. Finally, Talwar and Wieder [Talwar and Wieder 2007] extend the same result to the case where balls are weighted with weights coming from a distribution with finite variance.

Resource balancing in Kinesis differs from the above balls-and-bins models in several ways:

Segmentation: Kinesis servers are divided into k equal-sized segments and a target server is chosen from each segment independently. Vöcking [Vöcking 1999] confirms that the above bounds hold for this case.

Replication: Kinesis employs r-fold replication, and, hence, picks r out of k servers during writes. Most of the results above can be easily extended to include replication.

Deletions: The schemes outlined so far did not consider the case where balls may be removed from the bins. However, objects in Kinesis may get deleted during updates. Moreover, read and write requests consume network bandwidth only temporarily and disappear from the system once competed. These scenarios correspond to a balls-and-bins problem with deletions. Fortunately, the analytical bounds for balls-and-bins continue to hold even during deletions as long as the deletions are random and independent from the random choices of the insertion algorithm [Vöcking 1999].

Heterogeneity: Finally, heterogeneity exists at different levels in a practical storage system; objects often have different size, disks vary in capacity and performance, and network bandwidth is variable too. As we noted earlier, the multiple-choice paradigm works for variable-sized objects (weighted balls) and heterogeneous disks and networks (weighted bins) as shown in [Talwar and Wieder 2007] and [Wieder 2007] respectively.

4.2 Incremental Expansion

Incremental addition of new servers to the system could introduce both key-space imbalance and storage imbalance.

4.2.1 *Storage Balance.* A key concern during incremental expansion is whether or not the storage imbalance created during the expansion is overcome during further insertions of objects. In this section, we analyze the conditions under which the storage imbalance can be overcome.

Consider a fully-balanced Kinesis system with kn servers, where each of the k segments contains n servers, and each server stores m unit-sized objects. Assume that initially, every server is responsible for an equal portion of the key space, that is, each server is sampled by the hash function with equal probability. Let expansions happen at rate α, that is, an expansion adds αn $(0 < \alpha \leq 1)$ new

servers to each segment. Then, each new server "takes over" half of the key space of an existing one, amounting to a transfer of roughly half of the objects from an existing server to the new one.

The system is now imbalanced in two respects. Let L be the set of new servers and the servers they split (hence $|L| = 2k\alpha n$), and H be the set of remaining $(1-\alpha)kn$ servers. Then:

(1) Each server in L holds roughly $m/2$ items while each server in H holds m items.

(2) The key space for a server in L is half of that for H. Consequently, the hash function samples a server in L with probability $1/2n$ but a server in H with probability $1/n$.

In order to analyze whether the servers in L catch up and the system gains its balance over time we need to calculate the expected number of replicas each server in L receives upon inserting a new element, denoted by E_L. We can compute E_L as follows. When a new data item is inserted the number of copies which fall in L is determined by the number of times a hash function samples a server from L. The probability a hash function samples a server in L is α. Let X be a random variable distributed according to the binomial distribution with parameters (k, α). It implies that X counts how many time the set L is sampled. The expected number of copies in L is therefore $C_L := \Pr[X = 1] + 2\Pr[X = 2] + \ldots + (r-1)\Pr[X = r-1] + r\Pr[X \geq r]$. Now $E_L = C_L/|L|$.

Let $E := r/(kn(1+\alpha))$ denote the expected number of replicas each server would have received, had the system been perfectly balanced. An important measure is the ratio between those two quantities $R_L := \frac{E_L}{E} = \frac{C_L(1+\alpha)}{2r\alpha}$.

When $R_L = 1$, all servers get the same number of new replicas on expectation. However, when $R_L < 1$, servers in L receive fewer replicas than servers in H; that is, the bias towards under-utilized servers does not overweigh the low sampling probability, and the servers in L will never catch up the servers in H in storage consumption. Whereas, when $R_L > 1$, servers in L get more replicas than servers in R on average and will eventually balance their storage utilization as new objects are inserted.

Table I summarizes the values of R_L for interesting cases. When $\alpha = 1$ (i.e., all servers are split), $R_L = 1$. In contrast, when $\alpha = 1/n$ (i.e., only one server is added to each segment), $R_L \approx \frac{k}{2r}$ since the probability of a server in L being sampled is about $1/2n$. In general, the value of R_L is a concave function of α and is maximized between these two extreme cases.

The above implies that split servers can always catch up with the un-split servers as long as $k \geq 2r$. Otherwise, the new servers may permanently lag behind the old servers in storage consumption. Note that a replica placement scheme with a single choice scheme belongs to the latter case and incurs the risk of high resource imbalance as a result of expansions. For Kinesis, we recommend a choice of $k \geq 2r$ based on the above analysis.

Note that, the above analysis assumes continues writes to the system (either due to creation of new objects or updates to existing objects) to achieve storage balance eventually. Naturally, achieving storage balance might take a long time if the workload is not write intensive. Moving objects proactively after each system

Table I. Imbalance (R_L) in the expected ratio of the number of new replicas received at less-utilized server to the system wide average during incremental expansion. α is the fraction of new servers added to the system during an expansion.

$\mathbf{k, r}$	$\mathbf{R_L}$ ($\alpha = \mathbf{1/n}$)	$\mathbf{R_L}$ ($\alpha = \mathbf{1}$)	$\mathbf{R_L}$ (**Max**)
$4, 3$	0.66	1	1.05 ($\alpha = 0.81$)
$6, 3$	1	1	1.27 ($\alpha = 0.44$)
$7, 3$	1.16	1	1.4 ($\alpha = 0.34$)
$10, 3$	1.66	1	1.84 ($\alpha = 0.18$)

expansions can help speed up storage balance at the expense of some additional overhead.

4.2.2 *Network Load Balance.* In contrast to storage balance, the key-space imbalance introduced by incremental expansion has a modest impact on netwrok load balance: For write requests, even if the r locations are chosen based on their storage consumption, the load balance is bounded by R_L, derived previously. For read requests, it is possible that new objects are more popular than old objects, and this popularity difference leads to a higher load on servers with more new objects. However, as shown in Table I, the unevenness in the distribution of new objects is small for suitable choices of k and r. Furthermore, the work of Wieder[Wieder 2007] implies that the theoretical bounds hold even when the sampling distribution are not perfectly uniform.

5. PROTOTYPE IMPLEMENTATION

We implemented a prototype distributed storage system in order to evaluate the Kinesis replica placement algorithm. However, it should be emphasized that we view Kinesis as a module that could be plugged into many existing or future distributed storage systems. Our prototype was used to evaluate this module—it is by no means a fully-featured large-scale distribute storage system.

The Kinesis prototype consists of storage servers that provide a basic service for storing and retrieving objects and front ends that manage placement of replicas, handle user requests, and balance resource utilization. Our implementation supports `Kinesis`(k, r) for different values of k and r and provides a *Read* and *Write* interface for mutable, variable-sized objects.

The prototype implements linear hashing as described in Section 3.1 using a 160-bit SHA-1 hash function. It generates k independent keys for an object by appending the object name with integers from 1 to k. An alternative, more efficient way to compute independent keys is to use k disjoint regions of a single hash value. For example, we can divide the 160-bit SHA-1 hash value into 10 segments and allocate 16 bits in the hash to each segment; this approach can support 10 segments with a maximum of 2^{16} servers in each.

5.1 Storage Servers

A storage server manages objects locally on a single machine and provides a remote interface for basic operations. The operations supported include *Append* to create a new object or append content to an already existing object, *Read* to read portions of content from an object, *Exists* to check the existence of an object, *Size* to provide the total storage consumed, and *Load* to provide the current network load in terms

of the number of outstanding Append and Read requests. Storage servers perform Append and Read operations only in fixed-sized blocks; thus, a read or write request from a client may involve several Read and Append operations between the server and the front end. In addition, a storage server maintains a list of objects it stores in-memory.

Storage servers also perform failure recovery. When a storage server fails, other servers create new replicas of the lost objects. Each object has a unique *primary* server responsible for its recovery. The primary server of an object is the first non-faulty server obtained by applying the hash functions in a fixed order. The prototype does not currently implement a failure detector. Instead, a failure is simulated by the front end by marking some storage server as faulty, never performing any operation on this storage server, and notifying other servers of the failure after a fixed failure-detection interval.

5.2 Front Ends

Front ends handle read and write requests from the clients. They redirect requests to the storage servers chosen based on the preferred resource balancing policy. For read operations, a front end first queries the k possible locations to determine the replica holders. Since our system is targeted at high-speed, LAN environments, the round trip time to check whether a server has an object is typically negligible (less than a millisecond). It is possible to cache the fetched results for a short period of time to improve efficiency. However, caching leads to consistency problems during write operations. If a read request does not complete due to a server failure, the request is redirected to a different replica-holding server.

The write operations include creations of new objects as well as updates. While in-place updates are suitable for fixed-sized objects, updates to variable-sized objects may aggravate storage imbalance. So, our implementation creates a new version of the object during each update and replicates it independently. When a front end receives a write request, it queries the k potential locations of the object to determine the current version number and creates r replicas with a higher version number. Collisions in version numbers in the presence of multiple control servers is avoided by using the identifier of the front end to break ties. If desired, older versions of objects may be deleted from the system in the background.

Our implementation efficiently handles several front ends at the same time. Front ends operate independently and do not have to coordinate with other front ends. They do not store persistent state but learn current replica locations and resource usage by directly querying the storage servers. Thus, front-end failures do not require expensive recovery mechanisms.

Front ends communicate with storage servers through a remote procedure call (RPC). At startup, a front end initiates connections with each storage server for performing RPCs. Our system devotes one connection for faster operations such as Exists, Size, and Load, and multiple connections for the disk operations Append and Read so that object transfers can proceed in parallel. Additionally, each storage server makes two RPC connections (one each for fast and slow operations) with every other storage server for failure recovery.

Table II. Workload Object-Size Characteristics.

Workload	Count	Max	Mean	Median
Block	10M	10MB	10MB	10MB
MSN	30,656	160.8MB	7.97MB	5.53MB
Plan 9	1.9M	2GB	93.4KB	3215B

6. EVALUATION

We compare Kinesis parameterized by k segments and r replicas, `Kinesis`(k, r), with a commonly-used replica placement scheme we characterize as the *Chain* scheme. Chain uses a single hash function to chose a *primary* location for an object and places r replicas on servers obtained through a deterministic function of the primary. Chain represents the variants of the single-choice placement strategy commonly used in Chained Declustering [Hsiao and DeWitt 1990], Petal [Lee and Thekkath 1996], PAST [Rowstron and Druschel 2001], CFS [Dabek et al. 2001], and Boxwood [MacCormick et al. 2004]. Note that Chain is highly correlated and does not provide freedom of choice for distributing write requests. However, it does provide choice for distributing read requests (1 out of r), and we take advantage of this choice for balancing network load during reads.

We implemented the `Chain`(r) scheme using a single hash function h (SHA1) with linear hashing. Given a data item d, with n servers in the system, d will be placed on the servers in $\{(h(d)+i) \bmod n \mid 0 \leq i < r\}$. Intuitively, `Chain`$(r)$ places the data item based on a single hash function, with replicas on the following $r-1$ server locations in a circular key space.

6.1 Workloads

We evaluate Kinesis using the three different workload traces summarized in Table II. The *block trace* consists of 10 million objects of the same size (10MB).[1] This is a synthetic workload of a block storage system. We assign unique numbers as identifiers to the objects and use them as hash keys.

The *MSN trace* is based on a 6-month access log from the MSN Video production servers from mid February of 2006; this trace contains a record for each client request to the video files on the servers. The trace contains 30,656 video files with a total size of 2.33TB. Further statistics are shown in Table II. The URLs for video files serve as hash keys.

The *Plan 9 trace* is based on the `bootes` Plan 9 file server history used in Venti [Quinlan and Dorward 2002]. From the provided block information, we reconstruct the file system trace consisting of file creation and modification events with size information. The trace contains about 2,757,441 events with 1,943,372 unique files. Based on the final file sizes after the last modifications, the trace has a total of about 173GB. Further statistics are shown in Table II. File identifiers are used as hash keys.

[1]We chose 10M because it is close to the average in the MSN Trace. We found the evaluation results insensitive to block size.

6.2 Simulations

Our simulations compare `Kinesis`(k, r) with `Chain`(r) in terms of storage balancing, load balancing, and failure recovery. The following sections show how balanced resource utilization affects the provisioning requirements to meet client workload, end-to-end performance, and availability of the system.

6.2.1 *Storage Balancing.* We first evaluate the amount of resources needed to meet the storage requirements of the application. Note that storage imbalance in the system implies that we may need more capacity than the actual storage requirements of an application. We define the metric *over-provisioning percentage* β to express the fraction of additional storage capacity needed to meet application demand.

We measure the over-provisioning percentage in simulations as follows: let ρ be the threshold fraction of the total system capacity that triggers incremental expansion; that is, a new server is added whenever the total amount of storage currently utilized exceeds ρ. We define $\rho_{\max}$ as the largest ρ-value that is permissible before some server exceeds its individual capacity at some time during a given workload. We can then compute the over-provisioning percentage β as $1/\rho_{\max} - 1$. The better balanced the storage, the lower the β and the less server over-provisioning is required.

We use the block trace for this experiment, start with 100 servers with capacity 64 GB each. We only show the results for adding 10 servers at a time, as we found that varying the number of servers added during each expansion does not change the quality of the results. We determine β experimentally by varying the values of ρ at which incremental expansion is triggered.

Figure 3 shows the values for β for `Kinesis`$(k, 3)$ with $5 \leq k \leq 10$ and for `Chain`(3). Note that $\beta = 0$ means that storage is perfectly balanced in the system and no additional storage capacity is required to meet application demand. Figure 3 shows that Kinesis comes much closer to this optimal value of 0 compared to Chain, the difference for $k = 7$ is by 23%, and for $k = 10$ by 32%. This means that for about every 150 servers used by `Chain`(3), `Kinesis`$(10, 3)$ may require only 120 servers, a 25% saving. Of course, in practice, these absolute numbers may depend on the properties of the application workload; nevertheless, the over-provisioning results presented here indicate that Kinesis can substantially reduce the provisioning cost in practice.

Figure 3 also shows that β decreases for `Kinesis` when k increases; increased freedom of choice leads to better storage balance and reduced over-provisioning. Based on this observation, in the rest of the paper, we use `Kinesis`$(7, 3)$ as the default setting for Kinesis.

The reduced over-provisioning is the result of better storage balance for Kinesis. To verify that `Kinesis` provides better storage balance across different traces, we simulate the process of adding objects in each of the three traces into a fixed set of 32 servers. (Using a power of 2 is favorable to `Chain`(3), since it will not suffer from hash-space imbalance. In contrast, because `Kinesis`$(7, 3)$ uses 7 segments, hash-space imbalance does appear.) Figure 4 shows the storage imbalance for all three traces, measured by the difference between the maximum number of bytes on any server and the average, as files are added into the system or modified. Only the

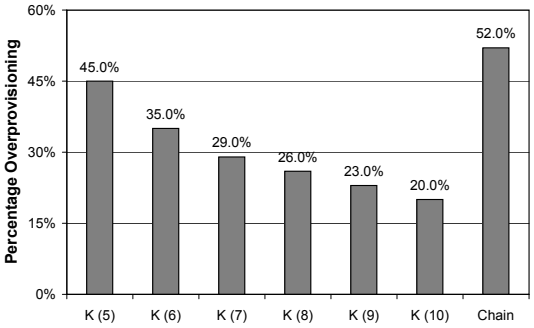


Fig. 3. `Kinesis`$(k, 3)$ vs. `Chain`(3): Over-Provisioning Percentage: `K(k)` corresponds to `Kinesis`$(k, 3)$ and `Chain` to `Chain`(3). Kinesis requires substantially less additional storage capacity than Chain to handle the same workload.

Plan 9 trace contains file modification events; for such an event, the new version is inserted with the same key and the old version deleted.

The results for all traces indicate clearly better storage balance for `Kinesis`$(7, 3)$. The different size distributions are mainly responsible for the different behavior for the three traces. The closer it is to the uniform distribution, the better the storage balance for all schemes. In particular, the spikes in the figure for the Plan 9 trace are due to some exceedingly large files in the trace. We tried different values of k and found that a larger k is better, but the differences are small.

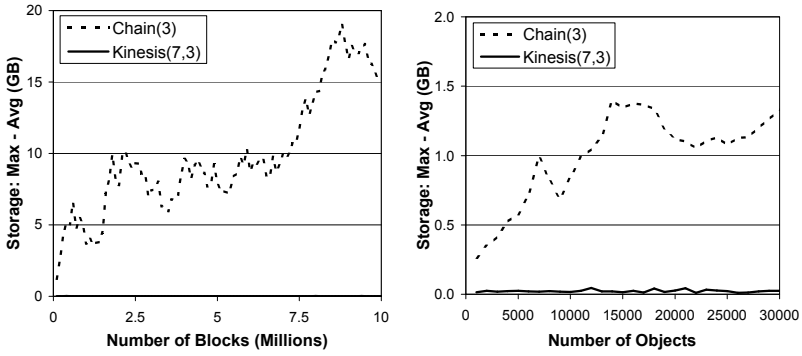
6.2.2 *Load Balancing.* Second, we investigate whether superior load balancing leads to the tangible benefit of reduced provisioning requirements, given a fixed performance target in terms of client-observed delay.

To simulate request processing, we use the following model. Each server provides p pipes, each with bandwidth B. That is, a server can process up to p concurrent client requests. Requests will be queued when all pipes are in use. A request to read an object of size s is assumed to take s/B time units to complete. Each request is sent to exactly one of the servers that stores a replica of the requested object; the particular server selected is the first one to have a pipe available, with ties broken arbitrarily.

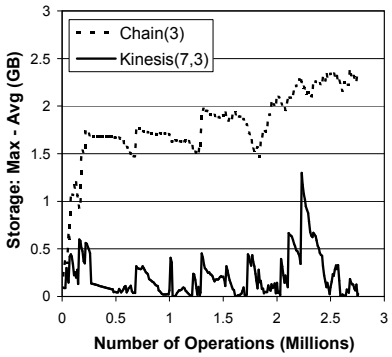
Note that the bandwidth provided by a single server to a client remains B even if only a single request is processed. This is reasonable because we assume that the bottleneck is on the client's network—certainly a realistic assumption for consumer clients on a DSL line, and probably also reasonable for corporate clients. We also ignore caching—as explained in Section 3.2, caching has negligible effect on performance for the workloads we consider.

The experiment assumes that each server has 2 Gbits per second of outgoing bandwidth available, clients have 5 Mbit per second of incoming bandwidth, and therefore, servers have $p = 410$ pipes each of bandwidth $B = 0.63$ MB per second.

The load balancing simulation examines the load imposed by read requests in the MSN Video trace. The simulation assumes that all the MSN Video files in our trace have first been inserted into the system, in a fixed order, using either the `Kinesis`$(7, 3)$ or `Chain`(3) schemes. We then simulate all read requests from



(a) Block Trace (b) MSN Trace



(c) Plan 9 Trace

Fig. 4. Storage Imbalance: `Kinesis`$(k, 3)$ vs. `Chain`(3)

30 consecutive days of the MSN Video trace. The relative timing of requests is taken directly from the trace, but the absolute rate at which requests arrive is a tunable parameter of the experiment. This enables us to increase the load on the system until it is reasonably saturated.[2] The results given below are for a request rate that averages just under 3 GB per second. Note that if read requests were spread constantly in time and served in a perfectly balanced fashion, only 12 servers would be required to keep up with this load (assuming 2 Gbits per second of outgoing bandwidth on each server, as explained above). In practice, requests are very bursty and are skewed towards a small number of popular files, so many more than 12 servers would be needed.

We measure the "90th percentile queuing delay" of read requests in the workload. The queuing delay of a request is the time it spends waiting in a queue before it

[2]Obviously, schemes tend to perform the same when the system is lightly loaded or completely overloaded.

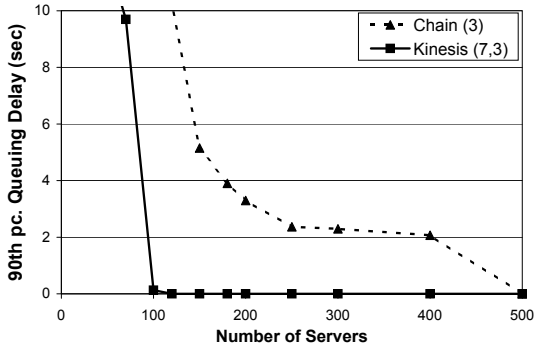


Fig. 5. `Kinesis`$(7,3)$ vs. `Chain`(3): 90th percentile queuing delay for read requests on MSN Video trace, as the number of servers is varied with average read request rate fixed at 2.9 GB per second.

begins to be serviced. The queuing delay is a more useful metric than the total delay since it excludes the time taken to actually transfer the file, which even a lightly-loaded system would incur. Because our trace is strongly skewed towards a small number of popular objects, requests for the popular objects inevitably incur some queuing delay regardless of how well-balanced this system is. Therefore, a robust measure of the queuing delay—in this case, the 90th percentile—is more useful than a non-robust measure (such as the average or maximum), for comparing the performance of Kinesis with a less well-balanced approach such as Chain. Clearly, if it is desired to eliminate queuing delay, then the system should create additional replicas of popular objects. In fact, it is easy for Kinesis to do this, but it would take us too far afield to investigate the creation of additional replicas here and we instead settle for a like-for-like comparison of the 90th percentile queuing delay between standard Kinesis and Chain.

Figure 5 shows the simulation results, comparing the 90th percentile queuing delay of `Kinesis`$(7,3)$ with `Chain`(3) for the workload described above, as the number of servers in each system is varied. Obviously, increasing the number of servers in a system provides more resources and thus decreases the delays. But from a provisioning point of view, the key question is: how many servers are required in order to achieve a given level of performance? It is clear from Figure 5 that `Chain`(3) requires many more servers than `Kinesis`$(7,3)$ to achieve the same performance. For example, to achieve a 90th percentile queuing delay of zero, `Kinesis`$(7,3)$ requires 120 servers, whereas `Chain`(3) requires over 400 servers—an additional provisioning requirement (and presumably cost) of over 230% to achieve this particular goal.

Figure 6 shows a similar scalability result, this time, fixing the number of servers at 128 and varying the read request rate. `Kinesis`$(7,3)$ can sustain a request rate of nearly 3 GB per second before its 90th percentile queuing delay becomes positive, whereas `Chain`(3) withstands only about 2.5 GB/s. Thus, according to this metric, Kinesis can serve a load about 20% higher than Chain.

6.2.3 *Resource Balancing during Writes.* This section evaluates the performance of Kinesis' resource balancing policy in the presence of write loads. We compare

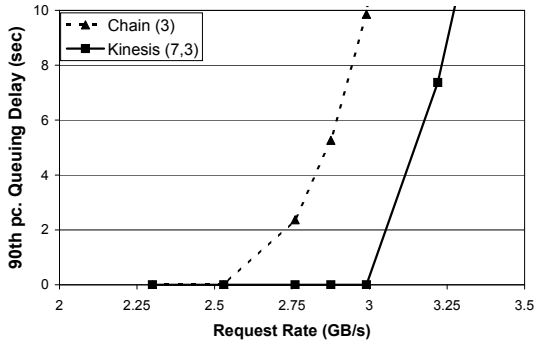


Fig. 6. `Kinesis`$(7,3)$ vs. `Chain`(3): 90th percentile queuing delay for read requests on MSN Video trace, as average read request rate is varied with a number of servers fixed at 128.

the three replica-location strategies described in Section 3.2: `Kinesis` S+L, which balances load when the load is heavy and storage otherwise; `Kinesis` S, which balances storage only; and `Kinesis` L, which balances load only. We also compare with `Chain`, which has no freedom of choice for write operations, and therefore balances neither storage nor load during writes.

Since we did not have a real-world workload with write operations, we simulated writes by converting some percentages of read operations in the MSN Video trace into writes. This preserves the overall object request distribution, while allowing us to study the effect of varying *update-to-read ratios*. We treat writes to an item already present in the system as updates and handle it by deleting the old replicas and inserting fresh ones at independently chosen locations.

We present simulations for one day of the MSN Video trace on a 14-node, comparing `Kinesis`(7,3) with `Chain`(3). Figure 7 shows the extent of load imbalance in different schemes as the update-to-read ratio is varied. The effect of load imbalance is once again plotted in terms of queuing delay (i.e., the delay incurred while the request is queued up at the network.) As expected, increased update activity results in worse load balance for `Chain` and `Kinesis` S because neither scheme takes into account network load during writes, while `Kinesis` S+L performs as well as `Kinesis` L. This is not surprising since `Kinesis` S+L gives clear priority for network load over storage. Note that load balance improves for both `Kinesis` S+L and `Kinesis` L as update activity is increased because updates place load on more nodes (due to replication) than reads.

Figure 8 shows the impact on storage balance as the percentage by which maximum storage consumption exceeds the average; the update-to-read ratio is set to 100%. It also plots the total load generated by update activity on a separate Y-axis. As expected, storage balance worsens in `Kinesis` S+L as more update load is imposed on the system. However, the storage balance improves substantially once the peak update activity is past and comes close to the earlier levels. This result backs up our intuition that short-term imbalances in storage caused by giving higher priority to network load balance can be compensated during lower levels of activity.

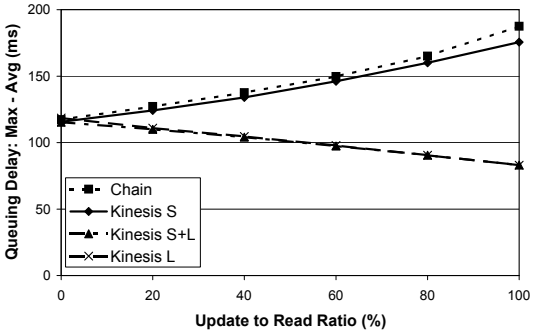


Fig. 7. Network-Load Imbalance vs. Update-to-Read Ratio: `Kinesis` S+L provides comparable load balance to `Kinesis` L despite the influence of storage consumption while balancing load.

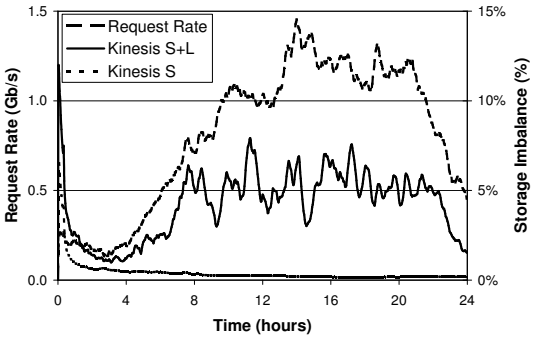


Fig. 8. Storage Imbalance vs. Time: `Kinesis` S+L compensates for temporary decreases in storage balance when client activity is low.

Overall, S+L is a reasonable scheme for simultaneously balancing both storage and network resources during writes.

6.2.4 *Failure Recovery.* Finally, we investigate the performance of failure recovery. In the following simulations, we compare `Kinesis`$(7, 3)$ with `Chain`(3). For `Kinesis`$(7, 3)$, when a replica fails, there are two remaining replicas that could serve as the source of data transfer and four unused hash locations as the location for holding a new replica. We experiment with two strategies: the storage-balancing recovery strategy `Kinesis` *SR* and the load-balancing recovery strategy `Kinesis` *LR*. In the storage-balancing strategy, we fix the source to be the *primary* of the object to replicate and, for Kinesis, pick the destination that has the most spare capacity. Note that for `Chain`(3), there is a single choice for the destination due to the chain structure. For Kinesis, the primary is defined as the non-faulty server with the lowest identifer holding a replica; for `Chain`, the primary is the first non-faulty server starting from the single hash location for the object. In the load-balancing recovery strategy, both the source and the destination are picked greedily as the ones with lower current recovery loads as the decisions are made one by one on all the objects stored on the failed servers. The load-balancing recovery strategy

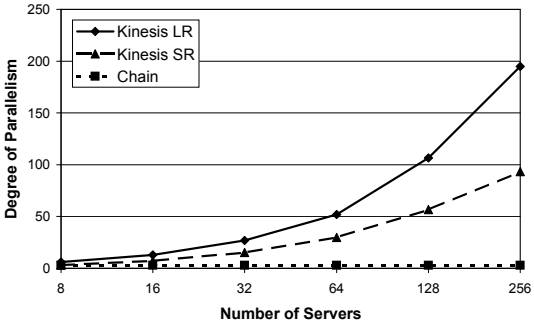


Fig. 9. `Kinesis`$(7,3)$ vs. `Chain`(3): Degree of Parallelism with Varying Numbers of Servers.

might require a central entity to pick the source and destination servers, whereas the storage-balancing recovery strategy can more easily be implemented in a distributed fashion.

For any given recovery instance, we compute the *degree of parallelism* π in the recovery process as follows. For each server s involved in the recovery process, compute the number of bytes b_s transferred due to recovery. Now define the degree of parallelism $\pi = (\sum_s b_s)/\max_s(b_s)$; that is, the ratio of all recovery bytes to the number of bytes transferred by the most-burdened recovery server. A large value of π implies a rapid, highly parallel recovery; for example, if $\pi = 50$, the recovery process was equivalent to a perfectly-balanced recovery on 50 servers.

For `Kinesis`(k,r), when a server fails, the number of servers that could be involved is the total number of servers in the other segments, which is about $n(k-1)/k$ for a total number n of servers. For `Chain`(r), that number is fixed at $2r-1$. Those are upper bounds for π.

We use the block trace with varying numbers of servers since it has a large number of objects and allows us to scale up the simulation with a large number of servers. Other traces offer similar behavior, but with slightly lower π values due to variations in sizes. In each trial, we look at the data transfer needed for recovery if the first server fails after every addition of 100,000 objects.

Figure 9 shows the minimum degree of parallelism π for `Kinesis` SR, `Kinesis` LR, and Chain as we increase the number of servers; for Chain we show the result only for the load-balancing recovery strategy because the other strategy yields slightly worse results. The figure shows that Kinesis has substantially higher degree of parallelism compared to Chain. Not surprisingly, `Kinesis` LR performs substantially better than `Kinesis` SR.

6.3 Experiments

We measured the performance of Kinesis and Chain on a cluster of 15 nodes (14 storage servers and one front end) connected by a Gigabit Ethernet switch. We set the block size for Append and Read to 1 MB and set the number of connections for slow RPC operations to one. For the above configuration we experimentally found that the disks in the servers provide a comparable throughput to the network interface at the front end. This means that the front end gets saturated before any

Table III. Storage utilization in GB at each server before and after failure recovery: `Kinesis`$(7, 3)$ achieves significantly better storage balance compared to `Chain`(3).

Server	`Kinesis`$(7, 3)$	`Chain`(3)	`Kinesis`$(7, 3)$	`Chain`(3)
1	11.9	10.7	**F**	**F**
2	11.4	10.7	13.0	14.2
3	11.7	10.2	13.1	14.1
4	11.9	10.6	13.1	13.9
5	11.5	10.3	13.1	10.3
6	11.9	10.8	13.1	10.8
7	11.6	14.0	13.1	14.0
8	12.4	17.8	12.4	17.8
9	12.5	17.4	12.0	17.4
10	12.9	14.3	13.1	14.3
11	13.4	10.9	13.4	10.9
12	12.7	10.7	13.1	10.7
13	12.3	10.4	13.1	10.4
14	11.7	10.6	13.1	10.6

of the storage servers. Therefore, we suppressed data transfer from storage servers to the front end during reads in order to saturate the disks at the storage servers. Note that this suppression does not affect the experimental comparison between Kinesis and Chain.

We performed the experiments using the MSN Video trace described earlier. We used a six-hour portion of this trace with $201,693$ requests for 5038 distinct objects. The total size of the content stored was about 170 GB, including the replicas, and the total amount of content fetched during the six-hour portion of the trace was 2.165 TB. The request rate varied greatly over time; we compressed the trace to three hours in order to create sufficient saturation. We created all the objects at the start of the experiment and collected performance statistics in two-minute intervals.

6.3.1 *Load Balance.* We first describe the performance of `Kinesis`$(7, 3)$ and `Chain`(3) when all servers are healthy. Figure 10 shows the extent of load balance in `Kinesis`$(7, 3)$ and `Chain`(3). Network load shown here is based on the number of requests queued up at a server. The figure also plots the overall request load handled by the system. Note that the instantaneous network load at a server can be much higher than the request rate because previously-issued requests might still be queued up. The graphs also have sudden spikes due to burstiness in the trace and the transient nature of network load.

Figure 10 shows that while both schemes achieve good balance of network load when the request rate in the system is low (in the first hour for instance), `Kinesis`$(7, 3)$ clearly provides better load balance at higher request rates. Moreover, this improvement in load balance translates to better latencies as shown in Figure 11. Similar to Figure 10, the latency incurred by `Kinesis`$(7, 3)$ and `Chain`(3) are comparable during periods of low client activity. However, `Kinesis`$(7, 3)$ incurs significantly lower latency during periods of increased client activity; `Kinesis`$(7, 3)$ measured an average latency of 1.73 seconds compared to 3.00 seconds for `Chain`(3). Overall, these figures indicate a strong correlation between network-load imbalance and client-perceived latencies; a balanced system, such as Kinesis, can provide better performance at a higher load.

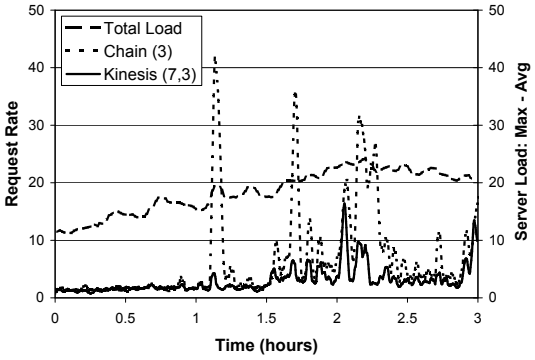


Fig. 10. Network-Load Imbalance vs. Time: `Kinesis`$(7, 3)$ balances network load significantly better than `Chain`(3) during high request loads.

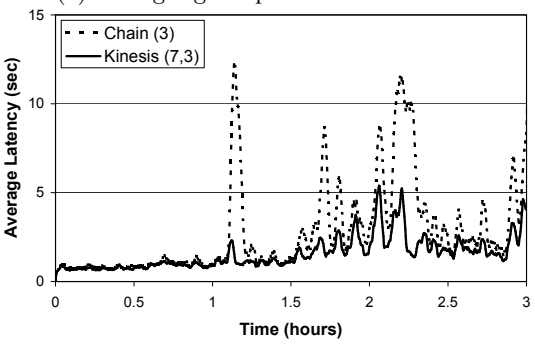


Fig. 11. Latency vs. Time: `Kinesis`$(7, 3)$'s better load balance leads to lower latencies compared to `Chain`(3).

6.3.2 *Failure Recovery.* We compare failure recovery in Kinesis and Chain by using the first hour of the trace, where both schemes incur comparable latency in the absence of failures (see Figure 11). We made server 1 fail 15 minutes into the trace and started failure recovery 15 minutes after the failure. We use the storage-balancing recovery strategy (i.e., `Kinesis` SR.)

Table III summarizes storage utilization before and after recovery. First of all, note that in `Kinesis`$(7, 3)$, storage consumption is almost uniformly balanced at all servers to begin with while the storage consumption is inherently unbalanced in `Chain`(3). Again with freedom-of-choice, Kinesis maintains storage balance during failure recovery. As shown in Table III, all servers in the system except Server 8 (which belongs to the same segment as the failed server) and Server 11 (which does not receive new replicas due to its high storage utilization) receive new replicas during failure recovery leading to a balanced storage system. In contrast, the rigid replica placement rules in Chain means that only the adjacent Servers 2, 3, and 4 perform recovery resulting in poor storage balance.

Table IV shows the overall performance of the failure recovery process in `Kinesis`$(7, 3)$ and `Chain`(3). Kinesis incurs a substantially lower recovery time of 17 minutes compared to 44 minutes for Chain due to the high degree of parallelism, 6 for Kinesis

Table IV. Failure Recovery: `Kinesis`$(7, 3)$ recovers from failures faster and with more uniformly distributed recovery load compared to `Chain`(3).

	Recovery Time	Recovery Nodes	Degree of Parallelism
`Kinesis`$(7, 3)$	17 min	12	6.0
`Chain`(3)	44 min	5	2.0

compared to 2 for Chain. The high degree of parallelism, measured as the ratio of the total recovery traffic to the maximum recovery load on a single storage server, indicates that the recovery load was more evenly spread between the servers.

The recovery process did not increase the client-perceived latency by a significant amount for both Kinesis and Chain.

6.4 Summary

The simulations and experimental results show that the Kinesis replica placement algorithm is a promising building block for high performance, scalable, and reliable distributed storage systems. The excellent storage and load balance due to Kinesis lead to lower provisioning requirements for meeting application demands, as well as better client-perceived latency, compared to a less balanced system. Kinesis preserves good resource balance even in the presence of system expansions and server failures. Finally, a high degree of parallelism during failure recovery improves the availability of the system.

7. DISCUSSION

This section discusses the key issues that arise in a practical application of Kinesis.

7.1 Choice of k and r

The choice of k and r has a big influence on the behavior of Kinesis. In practice, r is chosen based on the reliability requirement for the data. A larger r offers better fault tolerance and increases the number of choices for balancing read load although with higher overhead. We think that $r = 3$ is a reasonable tradeoff.

The gap between k and r decides the *level of freedom* in Kinesis—larger the gap, more the freedom. This translates into better storage balance and better write load balance. Our analysis shows that a sufficiently large gap ($\geq r+1$) facilitates servers to catch up after incremental expansion at the expense of slightly higher write load imbalance. A larger gap also offers more choices of locations on which new replicas can be created when servers fail, which helps to balance the load on servers for reconstructing the lost replicas.

However, a larger gap (i.e. a larger k with a fixed r) incurs higher cost for finding the object as k hash locations may need to be probed. We find the setting of $k = 2r + 1$ (e.g. `Kinesis`$(7, 3)$) offers a good tradeoff based on our theoretical analysis and simulations.

7.2 Choice of Objects

What constitutes an *object* is application-dependent. Applications can choose to store logical units (e.g., video files in the MSN trace) as objects. Alternatively, logical data items can be split or merged together to create an appropriate object for

Kinesis. Logical units tend to be of variable lengths, which is a significant cause of storage imbalance. While Talwar and Wieder [Talwar and Wieder 2007] guarantee that the system would be well-balanced on average, exact bounds are beyond our theoretical analysis. In the case where a logical unit is split into multiple fixed-length blocks, access to a logical unit might benefit from parallelism similar to that of disk striping, but requires coordination and assembly.

Bigger objects have less overhead per I/O operation since an object is often the unit of an I/O operation. It is desirable that the size is not too large: each server should be able to accommodate enough objects to facilitate parallel recovery: the number of objects on a failed server is an upper bound on the degree of parallelism during reconstruction of replicas. An object also needs a unique identifier. It could be the full path for a file or some artificially generated identifiers.

7.3 Coping with Heterogeneity

Heterogeneity among machines in a large scale cluster is often unavoidable over time. Our scheme is expected to accommodate such variations well because the differences in disk capacities or in other resources can be reduced to the bins-and-balls problem with non-uniform sampling probabilities for bins: for example, a server with twice the capacity can be regarded as two servers, each with half of the sampling probability. The multi-choice paradigm copes with such cases reasonably well, as described in Section 4.

7.4 Data Migration

Moving data between their hash locations after initial placement can provide additional benefits. Pagh and Rodler [Pagh and Rodler 2004] explored this idea in the context of hash tables. It was shown by Sanders *et al.* [Sanders et al. 2003] and Czumaj *et al.* [Czumaj et al. 2003] that near-perfect allocation can be achieved. Our experiments (not reported in this paper) confirmed this result. The amount of data movement required is large, but prudent movement could improve storage balancing significantly without introducing prohibitive overhead.

Objects can also be moved for load balancing if popular objects happen to reside on the same set of servers. We did not find the need for such movement in our experiments. Exceedingly-popular objects can be further replicated, not for reliability, but for load balancing. Due to dramatic popularity changes, an appropriate adaptive scheme must be developed for deciding when to create or destroy such replicas.

8. CONCLUSIONS

Kinesis is a simple, practical data and replica placement model that provides substantial cost savings due to reduced over-provisioning requirements, improved user-perceived performance, and fast, parallelized recovery from failures of a single or shared component. The effectiveness of Kinesis derives from a replica placement strategy that is structured (through the use of hash functions), flexible (through freedom of choices), and scattered (through pseudo-randomness). It results in superior balance of storage and load during normal operation, after incremental expansions, and during failures and recovery. The balance achieved by Kinesis, predicted by theoretical results, is confirmed here by simulations and experiments.

Given the simplicity of Kinesis, it is rather surprising that designers of distributed storage systems have not proposed Kinesis-like replica placement strategies in the past. Nevertheless, as far as we are aware, Kinesis is the first proposal of this kind. It appears to be a powerful theme that could be incorporated in the design of many existing and future distributed storage systems.

REFERENCES

Azar, Y., Broder, A. Z., Karlin, A. R., and Upfal, E. 1999. Balanced allocations. *SIAM Journal Computing 29,* 1, 180–200.

Berenbrink, P., Czumaj, A., Steger, A., and Vöcking, B. 2000. Balanced allocations: the heavily loaded case. In *Proc. of STOC*.

Byers, J., Considine, J., and Mitzenmacher, M. 2003. Simple load balancing for distributed hash tables. In *Proc. of IPTPS*.

Czumaj, A., Riley, C., and Scheideler, C. 2003. Perfectly balanced allocation.

Dabek, F., Kaashoek, M., Karger, D., Morris, R., and Stoica, I. 2001. Wide-area cooperative storage with CFS. In *Proc. of SOSP*. Banff, Canada.

Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The Google file system. In *Proc. of SOSP*.

Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., and Stoica, I. 2004. Load balancing in dynamic structured p2p systems. In *Proc. of INFOCOM*.

Hsiao, H. and DeWitt, D. J. 1990. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proc. of International Conference on Data Engineering (ICDE)*.

Ji, M., Felten, E. W., Wang, R., and Singh, J. P. 2000. Archipelago: An island-based file system for highly available and scalable internet services. In *Proc. of Windows Systems Symposium*.

Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., and Panigrahy, R. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of STOC*.

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. 2000. OceanStore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*.

Lee, E. K. and Thekkath, C. A. 1996. Petal: Distributed virtual disks. In *Proc. of ASPLOS*.

Litwin, W. 1980. Linear hashing: A new tool for file and table addressing. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*.

Lumb, C. R., Golding, R., and Ganger, G. R. 2004. DSPTF: Decentralized request distribution in brickbased storage systems. In *Proc. of ASPLOS*.

MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. 2004. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of OSDI*.

Pagh, R. and Rodler, F. F. 2004. Cuckoo hashing. *Journal of Algorithms 51,* 2, 122–144.

Pai, V. S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. 1998. Locality-aware request distribution in cluster-based network servers. In *Proc. of ASPLOS*.

Quinlan, S. and Dorward, S. 2002. Venti: a new approach to archival storage. In *Proc. of FAST*.

Rowstron, A. and Druschel, P. 2001. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of SOSP*. Banff, Canada.

Sanders, P., Egner, S., and Korst, J. H. M. 2003. Fast concurrent access to parallel disks. *Algorithmica 35,* 1, 21–55.

Talwar, K. and Wieder, U. 2007. Ballanced allocations: The weighted case. In *Proc. of STOC*.

van Renesse, R. and Schneider, F. B. 2004. Chain replication for supporting high throughput and availability. In *Proc. of OSDI*.

Vöcking, B. 1999. How asymmetry helps load balancing. In *Proc. of FOCS*. New York, NY.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. of OSDI*. Seattle, WA.

Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. 2006. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proc. of International Conference on Super Computing (SC)*. Tampa Bay, FL.

Wieder, U. 2007. Ballanced allocations with heterogeneous bins. In *Proc. of Sympostiom on Parallel Algorithms and Architecture (SPAA)*.