Differentiated Object Placement and Location for Self-Organizing Storage Clusters

Hong Tang and Tao Yang*
Department of Computer Science
University of California, Santa Barbara, CA 93106
{htang, tyang}@cs.ucsb.edu

Abstract

Component additions or failures are common for large-scale storage clusters in production environments. To improve manageability, it is desirable to automatically integrate new resources into an existing system as one virtual volume and repair data redundancy damaged by component failures. A key enabling technique for these capabilities is a flexible and scalable object placement and location scheme that quickly adapts to the additions or departures of storage nodes. This paper presents an object placement and location protocol that differentiates between small and large data objects for better space utilization and reduced management overhead comparing with uniform strategies. In our protocol, small objects, which are typically in large quantities, are placed through a high-dimension consistent hashing scheme. Large objects, which are much fewer in practice, are placed through a usage-based policy, and their locations are tracked with Bloom filters. Our protocol is fully distributed and only relies on soft states. It is particularly suitable for large-scale storage clusters with several thousand nodes and millions of objects. We validate the effectiveness of proposed techniques through simulations and experiments.

1 Introduction

Large-scale storage systems are a critical component in data-intensive applications such as multimedia databases, web information services, and data archival repositories. For these applications, their demand for storage capacity keeps growing as faster computers with larger memory at lower cost become available for more advanced computing [2, 8, 20, 22, 36]. To provide easy manageability, scattered storage resources need be unified and interfaced as a virtual disk, and component failures should be invisible to applications as much as possible.

A popular solution is to interconnect multiple storage devices through a dedicated storage area network (SAN) [36], such as Fiber Channels. Storage resources in a SAN system are typically organized in *volumes*, each of which normally consists of one or more disk groups and each group forms a RAID system. Volume formation information, called *volume maps*, are maintained by *volume controllers*. Data are addressed through virtual offsets within a volume, which are directly translated to physical disk addresses through the volume map. Such a volume-based resource virtualization scheme has two limitations:

1) Expanding the capacity of an existing volume typically requires reconfiguring existing disk groups and reorganizing the data. Some systems can expand a volume by adding a complete disk group at the end of the volume to avoid data reorganization. However, the process of extending the existing volume must be done manually and go through corresponding volume controllers.

2) A hard drive failure in a disk group reduces the redundancy level of that disk group, which makes it more vulnerable to further failures. To restore the redundancy level, either the failed hard drive must be replaced or the disk group be reorganized manually. Techniques such as hardware-based dedicated hot-spare or software-based distributed spare lessens the burden of administration, but the process is still not transparent to administrators.

Cluster-based storage systems have gained attention from both academia [2, 4, 7, 12, 14, 15, 16] and industry [21, 34] as an alternative to provide storage for data-intensive applications due to their scalability and cost advantage. In such a system, storage resources are directly attached to dedicated or general-purpose cluster nodes. Cluster nodes coordinate with each other through a shared LAN and aggregate distributed storage resources. Most of previous researches took the same approach as SAN to organize data into RAID-based volumes and thus share the same set of limitations.

This paper addresses the aforementioned limitations in the context of cluster-based storage systems, because the manageability could be exacerbated for a large-scale cluster with up to several thousand nodes. For example, at AskJeeves, disk additions or failures can occur as often as once every 2-3 days for a storage subsystem with several hundred nodes. The proposed solution, which is part of our Sorrento project for building self-organizing storage clusters, attacks both problems through the following two features:

*Also affiliated with AskJeeves/Teoma Technologies.

1) *automatic resource integration* – new resources are automatically integrated as part of the system to expand its capacity without manual intervention;

2) *automatic redundancy restoration* – after detecting node failures, the system automatically restores the redundancy level of affected data.

Designing such a self-organizing storage cluster is not trivial. Server nodes may often be added or temporarily out of reach in a cluster due to node failures or scheduled maintenance. When more nodes are added, data may be relocated from one node to another to maintain the overall balance of storage usage. Data stored on inaccessible disks need be reconstructed through redundancy and moved to other nodes. To accommodate the fact that storage resources and data locations are dynamically changing, we need a new data placement and location scheme that strikes the balance between flexibility and scalability. The scheme must be flexible in the sense that the address mapping from data to storage nodes must be adaptive to node additions or departures. The scheme must also be scalable to perhaps a few thousand storage nodes and millions of data objects, which discourages solutions relying on centralized components or distributed data structures with strong consistency assumptions [13] (such as the page mapping table scheme suggested in [22]).

Our solution adopts the object-based storage device model [1], and views the whole system as a collection of objects: a small number of large data objects and a large number of small objects. The central idea of our solution is to use a differentiated strategy for these two types of objects. The different characteristics of small objects and large objects have been studied in [5] and [35], and used to differentiate I/O operations in various researches [2, 7, 16]. However, to our knowledge, none have taken advantage of it to optimize object placement and location operations.

In our solution, the locations of small objects are chosen by a consistent hashing [18] based scheme. The scheme places balanced share of objects on each storage node. In the event of node additions or departures, it only migrates a small fraction of objects to maintain hash consistency. For large objects, migration is discouraged to avoid high copying overhead and we adopt a simple usage-based object distribution mechanism to balance space usage among nodes. We keep track of their locations by using Bloom filters [6]. The Bloom filters are incrementally updated through multicast, and we use a chunked and redundant refreshing mechanism to tolerate packet losses.

The object placement and location protocol described in this paper has been completely implemented and is the focus of this paper, while the Sorrento prototype is currently under development at UCSB. The rest of the paper is organized as follows: Section 2 presents a brief overview of the system design of Sorrento. Section 3 and 4 describe our object placement and location protocol using consistent hashing and Bloom filters. We show the effectiveness of our protocol design in Section 5 through simulations and experiments. Section 6 discusses related work and Section 7 concludes the paper.

2 System Overview

The design of Sorrento has leveraged two techniques from Slice [2] and OceanStore [20]: *object-based storage device* and *local client request routing*.

Object-Based Storage Device. The concept of *object-based storage device* (OBSD) was proposed by the National Storage Industry Consortium (NSIC) [1] in 1999 and has been adopted in the design of various storage systems such as Slice [2] and OceanStore [20]. Data in an OBSD are addressed through logical offsets within objects. Each object is identified by a globally unique ID called *GUID*. Comparing with a sector-based storage device, where logical addresses are directly mapped to physical addresses and data redundancy is performed on sector basis, an OBSD offers more freedom in choosing data placement and redundancy schemes as a benefit from the location-transparency nature of GUIDs.

In Sorrento, addressable objects from external applications are called *logical objects*. Meta-data corresponding to a logical object, such as redundancy schemes and additional parameters, are stored in a *meta object*, which functions similarly as i-nodes in traditional UNIX file systems. In fact, a logical object shares the same GUID with its corresponding meta object. For a small logical object, its data are attached with the meta object. For a large logical object, it is fragmented into a set of *content objects* and the GUIDs for content objects are stored in its meta object. Both *meta objects* and *content objects* are refereed to as *physical objects* inside Sorrento. Physical objects are stored in their entirety on native file systems. Thus the data location problem in Sorrento is essentially the problem of how to locate physical objects.

We want to emphasize that data layout schemes in traditional RAID systems, such as striping, could also be used to organize data in fragments. In Sorrento, applications can specify a combination of striping and partitioning to organize data in fragments for each logical object.

Local Client Request Routing. In Slice [2], client requests are intercepted and routed to storage providers by a lightweight entity called *μproxy*, which resides in clients and route requests through packet rewriting. By pushing the responsibility of request routing to client nodes, the system can retain transparency with little overhead. A similar idea is adopted by Sorrento, where we use an entity called storage *aggregators* to route client requests. As will be discussed below, however, the approach taken by aggregators is quite different from μproxies.

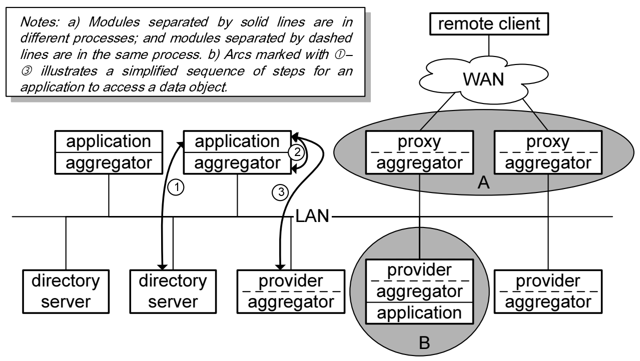

Notes: a) Modules separated by solid lines are in different processes; and modules separated by dashed lines are in the same process. b) Arcs marked with ①–③ illustrates a simplified sequence of steps for an application to access a data object.
remote client
WAN
application
aggregator
application
aggregator
proxy
aggregator
proxy
aggregator
A
①
②
③
LAN
directory server
directory server
provider
aggregator
provider
aggregator
application
B
provider
aggregator

Figure 1: Sorrento system architecture.

Figure 1 shows the system architecture of Sorrento. Applications access Sorrento through a set of library APIs, which interact with three types of entities: *directory server*, *provider* and *aggregator* (left half of the figure).

Directory servers maintain the mapping between human understandable hierarchical names to object GUIDs. The hierarchical tree can be easily stored in a relational database and various partitioning [2] or primary-secondary replication schemes [3, 9, 33] can be applied to provide scalability and fault-tolerance.

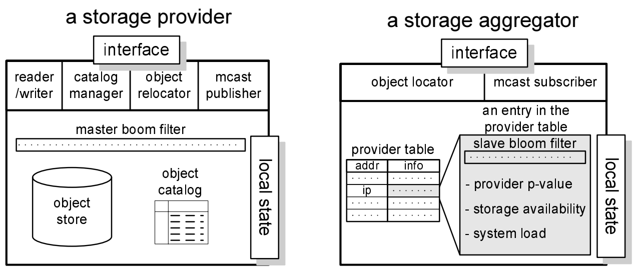


Figure 2: Components inside aggregators and providers.

As shown in Figure 2, each *provider* manages the objects stored in its locally attached physical disks and maintains an object catalog. Providers are responsible for data retrieval/modification (*reader/writer*) and answering queries about the existence of a certain object (*catalog manager*). Additionally, providers multicast control information periodically (*mcast publisher*). The control information disseminated by a provider contains its current work load, storage space availability, and a summary of objects it keeps (*master Bloom filter*). Providers are also responsible for detecting node additions and failures and relocating objects to maintain the healthiness of the system through the process of *proactive introspection* performed by *object relocators* (details in Section 4.3).

Also shown in Figure 2, each *aggregator* subscribes to the same multicast channel and maintains a global picture of the set of live providers (*provider table*). Using the provider table, aggregators can answer queries such as: which nodes may have a certain object, what are the loads of those nodes, or which nodes may have storage space to host a new object. Aggregators will be discussed in detail in Section 4.1 and 4.2. Aggregators can either run as a standalone process and communicate with local processes through IPC, or it can be configured as a set of threads with shared states inside another process. Running an aggregator inside another process eliminates the IPC overhead; however, collecting and maintaining information about the set of live providers is a gradual process and thus for short-lived processes, they must connect to standalone aggregators. Note that each provider has an integrated aggregator module inside it, which is used by relocators to expedite the process of proactive introspection (Section 4.3).

The arcs labeled with ①–③ in Figure 1 illustrate a simplified sequence of steps for an application to access a data object: ① the object name is resolves to its GUID through a directory server; ② the location of this object is returned by a local aggregator (Section 4.1 and 4.2); ③ the client directs read/write operations to the selected provider. Note that all these operations are encapsulated in a user-level library linked to the application, and are transparent to end users.

In Figure 1 (shadowed area marked with A), each *proxy* is a special application that accesses the storage on behalf of clients outside the trusted network boundary. We also depict a possible scenario of co-locating applications and providers on a single cluster node in Figure 1 (shadowed area marked with B), which may be preferable for clusters with functionally symmetric nodes [31].

Data redundancy, consistency, concurrency control and conflict resolution have been extensively studied in storage systems [26, 27], distributed databases [9, 13, 30, 33] and distributed file systems [4, 19, 24], and are not the focus of this work. We briefly describe the schemes adopted in Sorrento on these three aspects as follows: **1) Data redundancy.** Sorrento uses a combination of replication and erasure coding [23, 27] to survive data through software/hardware failures. **2) Consistency model.** Sorrento adopts a version-based consistency model on logical object basis, which bears some similarity with Amoeba [24] and the Coda file system [19]. A user's modifications are first applied to a shadow copy of the object, which is invisible to other users. Once a set of modifications transfer the logical object from one consistent state to another, they can be explicitly made visible to other users by the *commit* operation. **3) Concurrency control and conflict resolution.** Conflicting updates can be avoided by cooperative write-lock leasing through directory servers; otherwise, they will be detected during the *commit* operation and subsequently resolved through some external mechanisms or end-user intervention. Albeit its simplicity, automatic conflict resolution schemes such as OceanStore's predicate-based update primitives [20] or Bayou's merge procedures [33] could be easily implemented upon it.

3 Design Alternatives for Object Placement and Location

The fundamental issue of object placement and location is the management of a mapping from objects (GUIDs) to the dynamically changing set of storage providers. Using a centralized manager to maintain such a mapping is not scalable with millions of objects and thousands of storage providers. In terms of distributed approaches, [22] suggested to use a mapping table with one entry per object. However, replicating and synchronizing the mapping table on all aggregators can be expensive in terms of memory consumption and communication overhead. In this section, we first present two alternative object placement and location schemes together with their limitations. Then we present a solution which differentiates data objects based on their sizes and uses both schemes accordingly.

3.1 Consistent Hashing

The first idea is to use fixed hashing to distribute objects to storage providers to eliminate the need of a mapping table. More specifically, all we need is a hash function that maps object GUIDs to storage providers. A typical modulus-based hash function will not work well because when the number of providers changes, virtually all objects must be migrated due to changed hash results. In fact, a usable hashing scheme must only change the locations of a small fraction of total objects when a provider joins or leaves the cluster. The concept of *consistent hashing* [18] proposed by Karger et al. meets this requirement. The most important feature of consistent hashing is that it maintains stable hashing results for the majority of objects when the membership of the provider set is slightly changed.

The basic idea of their construction is to map objects and buckets (storage providers in our case) to points in a one-dimension interval $[0, 1]$ and thus establish a distance relationship between objects and providers[1]. The scheme then hashes objects to their *closest* providers. One may conceive that every point on $[0, 1]$ is controlled by its closest provider and an object that is mapped to point x will be hashed to the provider controlling x. When a provider is added or removed, only objects in the *neighborhood* of that provider will change their locations. To make sure each provider controls a similar range of neighborhood in $[0, 1]$, the algorithm maps each provider to $\kappa \log P$ points (called *virtual nodes* in their paper), where P is the number of providers. [18] proved that the deviation of the object distribution can be controlled by adjusting the scaling factor κ.

However, such a hashing scheme has two weaknesses: 1) The locations of objects are not controllable to balance storage usages on different nodes under hashing. When some providers run out of storage space, others might only use a fraction of local storage. This practically limits the amount of total usable storage space. This problem could be much worse for heterogeneous environments where storage resources are not evenly distributed to providers. Our experiments show that the percentage of wasted space could be as much as 26% for a homogeneous 100-node cluster. 2) Data migration is inevitable in the event of adding or losing a provider. For a large-scale storage cluster, the amount of migrated data could still be quite considerable.

3.2 Tracking Object Locations with Bloom Filters

In the second scheme, we use a heuristic algorithm to place objects to balance storage usage among different providers. Then we use a compact data structure, Bloom filters[2] [6], to track the locations of individual objects. Briefly speaking, each provider uses a master Bloom filter to capture the set of objects in its local store, and each aggregator maintains an array of slave Bloom filters, each corresponding to a provider's master Bloom filter. To locate an object, an aggregator simply iterates through each slave Bloom filter and performs the membership test.

Comparing with the mapping table approach [22], Bloom filters reduce the memory requirement by an order of magnitude, with the introduction of a small percentage of false positives. Additionally, since changes to the Bloom filters are propagated from providers to aggregators periodically, it is possible to have false negatives – even though a provider holds an object, the stale slave Bloom filter for that provider on an aggregator might return false for the membership test of that object. The presence of false positives and false negatives will not harm the correctness of the object location process. False positives will be detected when we actually attempt to access an object from a provider and only find that the provider does not hold the object. False negatives can be tolerated by a retry in a later time, or by multicasting a query to all providers asking who has the object.

The weakness with Bloom filters is that the memory and bandwidth required to maintain these filters could still turn out to be too much for a large-scale storage cluster. For example, our simulation study shows that Bloom filters will consume 15MB memory and 142KB/s bandwidth for a modest-sized cluster with 100 nodes and 10 million objects. For a large-scale cluster with several thousand nodes, the

[1]A slight modification introduced in [32] calculates the distance in a one-dimension circular space – for example, in the $[0, 1]$ circular space, the distance from 0.1 to 0.2 is 0.1 and the distance from 0.2 to 0.1 is 0.9.

[2]A Bloom filter encodes a set of object GUIDs in a bit array and can be used to check whether a given GUID is in that set with a small percentage of false positive. Please refer to [6] for details.

	Service-offline	Service-online	Group archive
Total # of Objs	489601	34235	92715
Total Size (MB)	7167382	1978640	46977
%obj using 90% space	1.3	6.9	7.1
%obj using 95% space	4.3	8.1	16.1
%obj using 98% space	9.1	8.8	23.3
knee-point (%obj/%spc)	3.1/94.0	9.5/99.7	5.5/88.8

Figure 3: The statistics and characteristics of three file systems for non-interactive usage.

memory consumption can be more than 100MB and 1MB/s respectively.

3.3 Differentiated Strategy for Small and Large Objects

Our solution is to combine both schemes to manage objects with different sizes. This is motivated by previous studies by Baker et al. [5] and Vogels [35], in which they found that large files and small files exhibit different characteristics in terms of user activities, access patterns, and their life times. Particularly, in terms of file size distribution, it turns out that most operations are for small files while most bytes transferred are for large files. This could be translated to the statement that the majority of files in a file system are small files, but the majority of disk space is consumed by large files. Additionally, most file creations and deletions operations are for small files. Both studies are based on typical *interactive* usage of file systems. However, we believe the conclusions still hold for *non-interactive* workload. To confirm our belief, we studied the file size distributions for three systems under non-interactive usage: 1) storage for offline processing of a commercial search engine at AskJeeves/Teoma (such as page crawling and parsing) (**Service-offline**); 2) storage for online database generation of the same company (**Service-online**). 3) a backup archive server for a research lab at UCSB (**Group archive**);

The sizes and numbers of objects in these three systems are shown in Figure 3. And the file size distributions for these systems are shown in Figure 4. The x-axis in Figure 4 is the percentage of total objects. The y-axis is the accumulated size of the largest $x\%$ objects expressed as a percentage of the total size. To make things easier to understand, Figure 3 further shows the *knee points*[3] on the file size distribution curves in Figure 4 and the percentages of objects accounting for 90%, 95% and 98% of the total size. As we can see, the following $10-90$ rule is valid for all three systems: less than 10% of the objects consume more than 90% of the disk space.

A natural question is whether we can take advantage of this observation in the design of the object placement and location protocol. The answer is *yes*. As we can see, by applying consistent hashing for small objects and Bloom filters for large objects, the weaknesses stated previously can be effectively alleviated: For consistent hashing, since small objects only take a small fraction of disk space, relocating them will also consume much less network bandwidth in comparison with a uniform strategy using only consistency hashing. For the same reason, the wasted space due to uncontrollable storage usage under hashing can also be significantly reduced. For Bloom filters, since large objects are much fewer and created/deleted much less frequently, the memory and bandwidth consumption required by Bloom filters can be significant reduced too. Figure 5 compares the two schemes using only consistent hashing (**CH**) and Bloom filters (**BLOOM**) with our differentiated approach (**DIFF**) in terms of storage space usage, potential data migration overhead, and memory/bandwidth consumption. As we can see, DIFF enjoys the strengths from both BLOOM and CH while avoiding either scheme's weaknesses.

[3]A knee-point is measured as the point on a curve where the slope is 1.0.

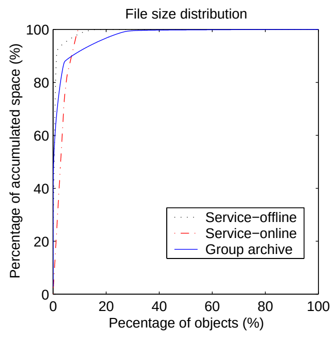


Figure 4: Accumulative file size distributions for three file systems of non-interactive usage. We sort files based on their sizes in descending order. x-axis is the percentage of total objects; y-axis is the accumulative file size normalized as a percentage of the total size. The meaning of a point (x, y) on a curve is that the largest $x\%$ objects account for $y\%$ of the total space taken.

	CH	BLOOM	DIFF
Storage Usage	$-$	$+$	$+$
Data Migration Overhead	$-$	$+$	$+$
Memory Overhead	$+$	$-$	$+$
Bandwidth Overhead	$+$	$-$	$+$

Figure 5: A comparison of three object placement and location schemes. **CH** - consistent hashing; **BLOOM** - Bloom filters; **DIFF** - using CH for small objects and BLOOM for large objects. A "$+/-$" means good/bad or low/high overhead.

The idea of differentiating small and large objects is also reflected in our choice of data redundancy schemes. For large objects, we use a space-efficient *erasure coding* [23, 27] scheme. For small objects, we use full content replication. Replication is not effective for large objects because it will lead to low space utilization. Conversely, using erasure coding for small objects will only have minor improvement in terms of space utilization, but will significantly degrade their write performance due to the heavy overhead of re-computing erasure blocks[4].

Finding the right threshold to distinguish between small objects and large objects is not easy. Ideally, it is the most efficient to set the breakpoint at the knee point of the accumulated file size distribution curve, which is only known after objects have been stored into the system. In Sorrento, we choose the threshold to be 512KB[5], and plan to study how to adaptively set the threshold in the future.

4 Protocol Design and Implementation

4.1 Small Object Placement and Location

In this section, we first present an alternative consistent hashing scheme called *High-Dimension Consistent Hashing* or *HDCH*, and then describe our small object placement and location protocol using consistent hashing.

High-Dimension Consistent Hashing

The basic idea of HDCH is similar to the original consistent hashing scheme [18] (which we call *Single-Dimension Consistent Hashing* or *SDCH*). The difference is that HDCH establishes the distance relationship in a high-dimension space. Our HDCH scheme works as follows: Objects and providers are mapped to n-bit integers, which represent discrete position values (called *p-values*) in an n-dimension space with only two values (0 and 1) in each dimension. Then we store objects to their *closest* providers in terms of their Hamming distances[6]. For providers of the same Hamming distance to an object, we choose the one with the smallest XOR result to break the tie. Figure 6 illustrates the HDCH scheme with $n = 4$ and 4 storage providers. In Sorrento, we use 32-bit p-values so that the distance calculation can be efficiently done through integer arithmetic.

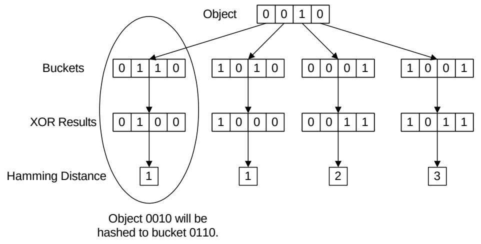


Figure 6: An example of HDCH scheme: an object is hashed into a set of 4 buckets with $n = 4$.

The main advantage of HDCH over SDCH is that HDCH exhibits more consistent behavior for clusters with variable scales, while the SDCH scheme is less effective for small to medium-scale clusters because it relies on statistical randomness to achieve the evenness of object distribution. To give some intuition on this conclusion, consider an extreme case where there are only two providers and calculate the size of the controlling neighborhood for each provider under both schemes[7], because the chance for an object to be hashed to a specific provider is proportional to the size of that provider's controlling neighborhood. Under HDCH, each provider controls exactly 50% of the total mapping space regardless of the p-values of both providers; while under SDCH, such a case rarely happens when the scaling factor κ is far less than the total number of positions in the space[8]. The above statement has been verified with our experimental study, which also shows that both HDCH and SDCH are competitive when the number of providers is large. It is our on-going work to analyze the HDCH algorithm from a theoretical point of view.

Placing and Locating Small Objects Using Consistent Hashing

Our small object placement and location protocol is summarized on three aspects as follows:

(1) **Provider P-value Generation.** A provider's p-value is generated by hashing the provider's network address. The hash function is based on MD5 and we modified it to guarantee the p-values for two different providers are always different.

(2) **Object Placement.** To determine the set of the replica locations of a newly created small object, the aggregator hashes the object GUID to a sequence of p-values, which will in turn be used to select storage providers though HDCH. The sequence of p-values are generated through iterative execution of MD5 hashing. Figure 7 illustrates how a sequence of 32-bit p-values are generated: First, the object's GUID is taken as the initial seed, and an MD5 digest is calculated from the seed. The 16-byte digest is then split into 4 p-values. This same MD5 digest will be used as the hashing seed in the next round if more p-values are needed.

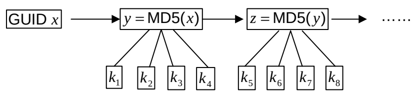


Figure 7: An illustration on p-value sequence generation.

Suppose we want to replicate an object k times, the ag-

[4]Using erasure coding for large objects will show less degradation because large objects have a different access pattern consisting mostly reads.

[5]This choice is fairly typical for storage systems under non-interactive use.

[6]The Hamming distance between two n-bit integers is the number of bits they differ.

[7]The size of a controlling neighborhood for a certain provider is naturally defined as the number of positions that are closer to that provider than other providers in the mapping space. If m providers are of the same closest distance to a position, that position is counted as $1/m$ toward each of the m providers' controlling neighborhood.

[8]Since each position is represented by an n-bit number internally, the number of positions in $[0, 1]$ that may be mapped to by an object is in fact finite and at most 2^n.

Notation	Type	Description
PvalSeqGen	Class	The p-value sequence generator.
$\text{CH}(v, A)$	Function	Consistent hashing function – mapping key v to a node $a \in A$ (A is a set of providers).
$\text{PAR-CHK}(o, A)$	Function	Check providers in A in parallel and return its subset that host o.
$\text{GUID}(o)$	Function	The GUID of object o.
k	Constant	The replication degree.
c	Constant	The scaling factor to tolerate hash collisions.

Figure 8: Notations for small object algorithms.

gregator will first generate $c \times k$ p-values from the GUID, where c is a small integer whose purpose will be apparent by the end of this paragraph. The aggregator then tries to hash each of the $c \times k$ p-values to a provider in the same order as they are generated till k *distinctive* providers are found or all $c \times k$ p-values are attempted. The reason we need to introduce the scaling factor c is that multiple p-values might be hashed to the same provider. The chance that all $c \times k$ p-values are hashed to the same provider decreases exponentially with regard to the value of $c \times k$. In Sorrento, we choose $c = 3$.

(3) **Object Lookup.** Locating small objects is similar to placing them. Again, $c \times k$ p-values are generated from the object GUID. The aggregator divides these p-values into c batches. Each time it hashes k p-values to a number of (at most k) providers, and then confirms the existence of the object on these providers in parallel. We try p-values in batches rather than sequentially so as to avoid overloading a particular provider. In a rare situation when one or more newly joined providers changes the consistent hashing results and renders existing object replicas inaccessible through the above process, the aggregator will perform an exhaustive search by querying the object through multicast.

Figure 9 shows the algorithms to calculate the set of replica locations (REP-SET), and to lookup an object (LOOKUP). The notations are listed in Figure 8.

4.2 Large Objects Placement and Location

Large objects are placed through a usage-based policy and their locations are tracked individually through Bloom filters. We summarize the protocol as follows:

(1) **Object Placement.** Our usage-based large object placement policy aims at balanced share of storage usage on different providers. To select a provider as the owner of an object, an aggregator first filters out providers that have insufficient space to hold the object. Then it filters out providers that have less-than-average amount of free space. Finally, each of the remaining providers has a probability of being chosen as the object owner proportional to its amount of available space.

Algorithm 4.1: REP-SET(o, P)

```
input: $o$: object; $P$: the set of live providers.
returns: A subset of $P$.
comment: Calculate the subset of $P$ where $o$ should be hosted.

PvalSeqGen.SetSeed(GUID($o$))
$Q \leftarrow \phi$
for $i \leftarrow 1$ to $c \times k$
    do
        $v \leftarrow$ PvalSeqGen.GetNextPvalue()
        $q \leftarrow \text{CH}(v, P)$
        $Q$.INSERT($q$)
        if $|Q| = k$
            then return ($Q$)
return ($Q$)
```

Algorithm 4.2: LOOKUP(o, P)

```
input: $o$: object; $P$: the set of live providers.
returns: A subset of $P$.
comment: Find a subset of $P$ that host $o$.

PvalSeqGen.SetSeed(GUID($o$))
for $i \leftarrow 1$ to $c$
    do
        $Q \leftarrow \phi$
        for $j \leftarrow 1$ to $k$
            do
                $v \leftarrow$ PvalSeqGen.GetNextPvalue()
                $q \leftarrow \text{CH}(v, P)$
                $Q$.INSERT($q$)
        $R \leftarrow \text{PAR-CHK}(o, Q)$
        if $R \neq \phi$
            then return ($R$)
comment: Rare case: no replica found through HDCH.
Return results of exhaustive search using multicast.
```

Figure 9: Algorithms used by aggregators to place/locate small objects.

(2) **Setting Parameters for Adaptive Bloom Filters.** The false positive rate of a Bloom filter is affected by its size and the number of hash functions used to set each key. Bloom showed that the space/false-positive trade-off reaches optimal when the percentage of set bits is close to 50% [6]. Additionally, assuming perfectly random hash functions, the false positive rate of a Bloom filter can be calculated as ρ^j, where ρ is the percentage of set bits and j the number of hash functions per key. Based on these two conclusions, we statically choose j such that $\frac{1}{2^j}$ is close to a pre-determined upper limit of false positive rates. In Sorrento, we use 6 hash functions. The initial size (number of bits) of a Bloom filter is chosen as $2jN$, where N is the number of objects. During the runtime, the size of a Bloom filter is dynamically adjusted so that the percentage of set bits is controlled in the range of $20 - 50\%$. If the percentage of set bits is lower than 20%, we shrink the size of the filter by half; if the percentage of set bits is higher than 50%, we double the size of the filter.

(3) **Handling Packet Losses.** Providers send out incremental updates of Bloom filters to aggregators periodically through multicast. Each update packet contains a list of bit-set and bit-clear records corresponding to the object additions and deletions occurred in the previous interval. Occasionally, when an aggregator detects that the size of a Bloom filter has been changed, it will request a complete copy of the Bloom filter explicitly from the provider. Propagating incremental updates through multicast is efficient; however, packets could be silently dropped in multicast and

over time, the slave copy of a Bloom filter might be quite different from the corresponding master copy. Our solution toward this problem is two-fold. First, each bit-set or bit-clear record will be multicast twice in consecutive updates. This way, occasional packet drop could be tolerated. In the second part of the solution, we break down the whole bit array of a Bloom filter into small chunks, and include a number of chunks in every update packet. The idea is to refresh the whole bit array gradually at a controlled pace. Additionally, we prioritize modified chunks over unmodified-modified chunks and send out different chunks at variable frequencies. Such a chunked refreshing scheme is aimed at tolerating a sudden burst of multiple packet drops. In Sorrento, the size of each chunk is 128 bytes.

Combining Two Protocols Together.
Our object placement and location protocol is a combination of the two protocols described in Section 4.1 and 4.2. The access of a logical object always starts with a lookup of the meta object (small object) through consistent hashing. If the actual data are stored in content objects, the Bloom filter-based large object location protocol will be invoked.

4.3 Automatic Redundancy Restoration and Object Relocation

The differentiated protocol described in Section 4.1 and 4.2 maps objects to the current set of live providers, which means newly added nodes and resources are automatically integrated to the system. The next question is how to support automatic redundancy restoration using the differentiated protocol. In this section, we first present a description of the tasks involved in automatic redundancy restoration, then we show our basic approaches and several optimizations.

The main task of automatic redundancy restoration is to examine the redundancy levels of accessible logical objects, discover those with damaged redundancy and restore them. The meaning of *damaged redundancy* actually differs between small and large objects (recall that we use full content replication for small objects and erasure coding for large objects). For a small logical object, this means that one or more providers in the replica location set (calculated by the REP-SET algorithm in Figure 9) do not actually own that object; and for a large logical object, this means that one or more fragments of the object are no longer accessible. Typically, an object's redundancy is damaged by provider failures. Additionally, we treat the relocation of a small object caused by provider additions as a special case of redundancy restoration. This is because immediately after its arrival, the new provider effectively takes over another provider as the new owner of that small object based to the changed hashing result, which practically reduces the redundancy degree of the small object and requires object relocation.

We accomplish the task of automatic redundancy restoration through two means. The first one is called *on-demand restoration* and is invoked by aggregators during the normal process of object location. When an aggregator discovers that the redundancy of an object requested by an application is damaged, it restores the redundancy either by replication (small objects) or by reconstructing the damaged fragment through erasure coding (large objects).

The second part is called *proactive introspection* and is carried out periodically by *object relocators* on providers (Figure 2). During each proactive introspection session, an object relocator walks through the whole object catalog. For each locally stored logical object, it attempts to access the object (including all replicas or external fragments) through the integrated aggregator module[9]. If not all object replicas or external fragments are available, it will attempt to restore them in the same way as on-demand restoration. It is rare but possible that different relocators attempt to restore the same object at the same time. This could result in some communication overhead; however, data integrity will not be compromised because of our version-based consistency model.

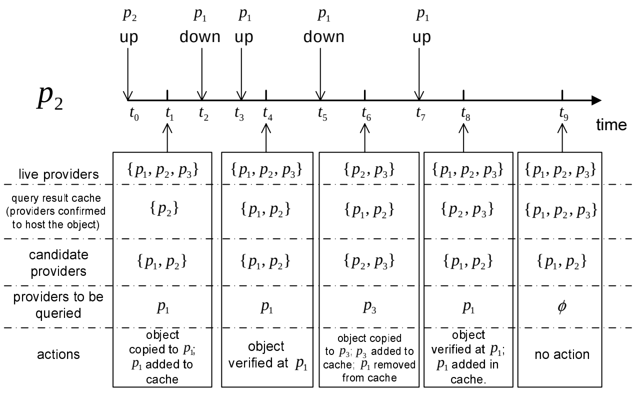
A problem with such a simple algorithm for proactive introspection is that to discover whether a logical object's redundancy is damaged or not, a relocator may need to query some providers to confirm that they do have a specific physical object (called *object-existence queries*). The cost of one or more network transactions per object is prohibitively expensive in practice. In the rest of the section, we devise two optimizations upon this basic proactive introspection algorithm, which would bound the number of network transactions in one proactive introspection session by the number of providers, and that no network transaction is required if no provider joins or leaves the system since the last session.

1) Batching Queries. Our first optimization is to batch the object-existence queries for the same provider in one network transaction. Typically, all queries for a same provider during an introspection session can be aggregated in one UDP request. So the total number of network transactions would be bounded by the total number of providers. However, batching queries does not reduce the number of object-existence queries. Additionally, proactive introspection will generate constant load of network transactions during each session even though no provider leaves or joins the system.

2) Query Result Caching. In our second optimization, we try to cache the results from previous queries and reuse them to avoid contacting providers for authoritative answers as long as the cached results are valid. A cached result remains valid if the authoritative provider stays alive since the result has been cached. Node failures are detected through *epoch numbers*. Every time a provider restarts, it increments its epoch number which is maintained on persistent

[9]An object relocator can directly access the information kept in the aggregator module through function calls.

storage. Epoch numbers are included in control packets disseminated by providers, and a provider failure can be detected by the change of the epoch number[10].



p_2 up
p_1 down
p_1 up
p_1 down
p_1 up
p_2
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
time
live providers
query result cache (providers confirmed to host the object)
candidate providers
providers to be queried
actions
$\{p_1, p_2, p_3\}$ $\{p_2\}$ $\{p_1, p_2\}$ p_1 object copied to p_1; p_1 added to cache
$\{p_1, p_2, p_3\}$ $\{p_1, p_2\}$ $\{p_1, p_2\}$ p_1 object verified at p_1
$\{p_2, p_3\}$ $\{p_1, p_2\}$ $\{p_2, p_3\}$ p_3 object copied to p_3; p_3 added to cache; p_1 removed from cache
$\{p_1, p_2, p_3\}$ $\{p_2, p_3\}$ $\{p_1, p_2\}$ p_1 object verified at p_1; p_1 added in cache.
$\{p_1, p_2, p_3\}$ $\{p_1, p_2, p_3\}$ $\{p_1, p_2\}$ ϕ no action

Figure 10: An illustration of using query result cache to relocate a small object in the presence of node failures and recoveries.

Due to space constraint, we only present an illustration in Figure 10 of how query result caching works. There are totally 3 providers p_1, p_2, p_3 and the activities shown in Figure 10 are initiated by the relocator on p_2. The relocator performs the proactive introspection periodically (at time t_1, t_4, t_6, t_8 and t_9). All activities are related to a small object which is only hosted by p_2 initially and has a specified replication degree of 2. The figure is relatively self-explanatory and we provide some rationales behind the actions taken in each session as follows:

At t_1, the object is replicated to p_1 to meet the specified redundancy requirement. At t_4, although p_1 is in the query result cache for that object, the relocator must confirm the object's existence again since p_1 suffers a failure between t_1 and t_4, which might cause the object no longer accessible from p_1. At t_6, p_1 is down, so the relocator removes p_1 from the cache and replicates the object to p_3 to maintain the object's redundancy. At t_8, p_1 is up again but the relocator has no knowledge whether p_1 might have the object any more, so it queries p_1 and only finds that p_1 does have the object. At t_9, all providers in the candidate set are in the cache, and there is no node joins or leaves since t_8, so no action is needed.

By batching multiple queries for the same provider in one network transaction and caching query results, the overhead of proactive introspection is significantly reduced. Figure 11 (a) summarizes the time complexity of the overhead of proactive introspection (excluding actual data copying cost) before and after the optimization. Figure 11 (b) provides an estimation of the actual cost after the optimizations for a specific setting of a 1000-node cluster, where an introspection session only takes about 8.9 seconds with the majority cost on the local computation of candidate sets.

[10]In Sorrento, disk failures are generalized as a type of node failures. When a provider detects a disk failure, it also increments the epoch number. So a disk failure is manifested as if the provider suffers a failure and recovers immediately.

(a) The time complexity of proactive introspection.
(P - # of providers; N - # of objects in the local store; k - replication degree; l - # of providers that join/leave the system since the last scan.)

	Candidate set calculations	UDP requests	Obj existence verifications
Before optimization	N	kN	kN
After optimization	N	$\min(P, kN)$	$\frac{klN}{P}$

(b) The actual cost breakdown for a specific setting.
The values of the parameters are: $P = 1000$, $N = 50000$, $k = 4, l = 10$. The per-operation cost is measured from a Pentium III 1.2GHz PC, and the time of a UDP request is measured as the round-trip delay of a 1.4KB UDP packet across a 100 BaseT link.

Cost category	Time/op (μs)	# Ops	Sub-total (ms)
Candidate set calculations	40	200000	8000
UDP requests	825	1000	825
Obj existence verifications	20	2000	40
		total time (ms)	8865

Figure 11: The overhead of proactive introspection.

In Sorrento, providers schedule proactive introspection sessions once a day, so such a small overhead is insignificant for the overall system performance.

5 Protocol Evaluation

The protocol described in this paper has been fully implemented, which is part of the Sorrento prototype under development at UCSB. The overall objective of the evaluation is to demonstrate the *effectiveness* and *scalability* of our differentiated object placement and location protocol:

1) The *effectiveness* of our differentiated protocol will be demonstrated by comparing it with two uniform strategies: a) consistent hashing-based object placement and location; b) usage-based object placement with Bloom filter-based tracking. We will analyze their space utilization, data migration cost, and memory/bandwidth consumption to validate our conclusion. (Section 5.2)

2) We will also demonstrate the *scalability* of this protocol in terms of the overhead for protocol invocation and protocol state maintenance. We will show that our protocol has very low overhead when varying the number of providers, the number of objects stored in the cluster and client request rates. (Section 5.3)

Due to space constraint, we could not present detailed evaluations on other techniques used in our protocol, such as the comparison between the original consistent hashing (SDCH) and HDCH; the usage-based large object placement policy; the effectiveness of chucked and redundant refreshing of Bloom filters to tolerate packet losses; and the effectiveness of proactive introspection.

Since we do not have a hardware infrastructure to deploy the kind of large-scale storage clusters described in this paper at UCSB, we have developed a fairly sophisticated simulator to aid our evaluation. A description of the simulator is presented in Section 5.1. For several timing-dependent metrics such as the service time for an object location request, we evaluate them through experiments in which different processes on different machines are used to imitate aggregators and providers. The detailed settings of these experiments will be discussed within specific contexts. All protocol-specific functions in the following simulations and experiments are directly linked to the same set of library modules which export the protocol interface. All experiments and simulations are conducted on a cluster of 40 dual Pentium III 1.2GHz Linux PCs connected with a Fast Ethernet switch.

5.1 Simulator Design

The simulator models the environment where a storage cluster serves object creation, deletion or location requests generated by clients. We first present the detailed simulation model. Then we discuss how client requests and objects with different sizes are generated to mimic realistic application workload.

Simulation Model. The main objective of this simulator is to model the system behavior related to object placement and location activities, thus disk operations or data transfers over the network are not modeled. The performance data we collect from the simulator include disk space usages and memory/bandwidth consumptions. Our main assumption is that client request load will not saturate the system, which is generally true in practice – a storage system is typically saturated due to insufficient disk/network bandwidth instead of high object placement/location request rates.

The work flow of the simulator is as follows. All client requests are combined as if they come from a single client that is able to generate requests as fast as specified. Additionally, all requests go through a single aggregator which serves each client request upon its arrival. Subsequently, each request will trigger other events representing the interactions between aggregators and providers during object placement and location operations. Each provider handles these events right after they are received. All providers multicast control information periodically. We simplify the modeling of network delays or request service times as fixed constants whose values are taken from real measurement, since we do not rely on the simulator to collect timing-dependent metrics.

The simulator maintains the following system states. Each provider has a pre-configured storage capacity, its master Bloom filter and object catalog. Each aggregator maintains a provider table. The simulator can dump a snapshot of the system state information at any time during a simulation.

The protocol invocation and state maintenance components in the simulator are directly linked to the protocol implementation modules. Thus they function in the same way as real situations. The simulator runs as a single process,

Name	Description
P	Number of providers.
V	Each provider's storage capacity.
r_c, r_d, r_l	Object creation, deletion, and lookup request rate.
I	Control information multicast interval.
T	Simulation duration.
N	Initial number of objects.

Figure 12: The parameters of the simulator.

and is able to simulate up to 80 million objects and 2500 providers. Figure 12 summaries the parameters of the simulator.

Request Generation. Simulations are driven by a client request generator module which generates object creation, deletion or lookup requests based on either Poisson processes or from the Sprite traces [5, 25]. Due to space constraint, we only present the simulation results where request arrivals are modeled as Poisson processes and our extended study showed that the results obtained by using Sprite traces are similar. To reflect the fact that small objects are created, deleted or located more frequently than large objects, we set the chance for a small object being chosen as the target of a request 10 times higher than that for a large object. Since the number of small objects is an order of magnitude higher than that of the large objects, this means that the number small object requests is two orders of magnitude higher than large object requests.

Object Generation. We use the file size distribution measured from a storage system used in the AskJeeves/Teoma search engine (the *Service-offline* curve from Figure 4) to generate objects with random sizes. However, the size distributions captured in Figure 4 are for logical objects, and some conversion must be done. For a logical object smaller than 512KB, we generate 4 physical replicas. For a logical object larger than 512KB, we divide it into F_d data fragments and then create F_r redundancy fragments through erasure coding. The values of F_d and F_r are calculated by the following formulas, where S is the object size:

$$F_d = \max\left(\left\lfloor \log_2\left(\frac{S}{256KB}\right)\right\rfloor, \left\lceil \frac{S}{200MB} \right\rceil\right); F_r = \max\left(2, [0.2F_d]\right)$$

The idea is to generate larger fragments for larger logical objects and limit each fragment within 200MB. Additionally, the space overhead of redundancy fragments is set at 20% and each large logical object must have at least 2 redundancy fragments. After the conversion, physical objects above 512KB consist 5.5% of total objects and consume 95.8% of total size.

5.2 Effectiveness of the Differentiated Protocol Design

We first evaluate the effectiveness of the differentiated protocol design (**DIFF**) through simulation. We com-

(a) Evaluation metrics and methodologies.

Metric	Meaning	Methodology
EffSpace	The percentage of effectively usable space	Keep creating objects until a provider runs out of space, and measure the percent of space used by created objects.
MigData	The percentage of data to be migrated when we add a provider	Create a set of objects, save each provider's catalog. Calculate offline the objects that must be migrated to the newly added node.
Memory	Memory consumed by the provider table	Calculated during simulation when the system is in a stable state ($r_c = r_d$).
Bandwidth	Bandwidth used by all control info updates	Calculated during simulation when the system is in a stable state ($r_c = r_d$).

(b) The settings of the simulator parameters.

Name	Value	Name	Value
P	100	V	100GB
r_c	100 req/sec	r_d	100 req/sec
r_l	1000 req/sec	I	1 second
T	100 seconds	N	10 Million

(c) The results.

Metric	HDCH	BLOOM	DIFF
EffSpace	73.7%	99.9%	98.8%
MigData	1.00%	0	0.042%
Memory	1.2KB	15.0MB	630KB
Bandwidth	3.20KB/s	142KB/s	7.42KB/s

Figure 13: Effectiveness of the differentiated protocol design.

pare our protocol with the schemes using only HDCH (**HDCH**) or Bloom filters with usage-based object placement (**BLOOM**). Essentially, we want to validate the "$+/-$" signs in Figure 5 with quantitative results. We compare these three schemes in terms of storage utilization, data migration overhead, and memory/bandwidth consumption. We summarize the definitions and evaluation methodologies of the metrics in Figure 13 (a).

The settings of the simulator parameters are shown in Figure 13 (b). In the measurement of *EffSpace*, the number of objects N is not bounded and we keep creating objects until some provider runs out of space. In the measurement of *MigData*, the request rates are not used because the simulator stops immediately after it creates N objects and dumps all providers' object catalogs.

The evaluation results are shown in Figure 13 (c). We can see that DIFF improves the *EffSpace* from 73.7% of HDCH to 98.8%, and is comparable with BLOOM which places objects purely based on usage. In terms of *MigData*, DIFF reduces the amount of migrated data by 24 folds comparing with HDCH, from 1% (15GB) to 0.042% (638MB). The reason for both improvement lies in the fact that the imbalanced storage usage and the data migration under HDCH is caused by the uncontrollable object placement of consistent hashing. By applying consistent hashing for small objects which consist only 4.2% of the total space, DIFF effectively confines the extent of the imbalanced storage usage and data migration within a much smaller scale. Overall, the percentage of usable space and data migration overhead are significantly improved system-wide.

Secondly, DIFF reduces the memory/bandwidth consumption by 24 and 19 folds respectively comparing with BLOOM. The reason lies in the fact that the memory and bandwidth consumption under BLOOM is roughly proportional to the number of objects and the object creation/deletion rates. By only using Bloom filters to track the locations of large objects, which are much fewer and are created/deleted much less frequently, DIFF is able to significantly reduce the management overhead. Note that the results are only for a modest-sized cluster with 100 nodes. Even though the overhead for BLOOM (15MB memory and 142KB/s as shown in Figure 13 (c)) seems fine with today's commodity hardware; however, they will reach more than 100MB and 1MB/s respectively for large-scale clusters with several thousand nodes.

In conclusion, our differentiated protocol takes advantage of both HDCH and BLOOM schemes, makes effective use of available storage, and maintains low management overhead.

5.3 Evaluation of Protocol Scalability

In this section, we present a detailed evaluation of the overhead of our differentiated object placement and location protocol in relation to various scaling parameters. By doing so, we seek to verify whether our protocol is able to meet the scalability target to support a storage cluster with several thousand nodes and millions of objects. Note that because of our local request routing design, the overhead associated with protocol invocation and state maintenance is mainly on the aggregator side.

Small Object Placement and Location.

First we evaluate the overhead of our consistent hashing scheme. The metric we use is the average service time for an aggregator to place or locate a small logical object. The service time is measured through a standalone program. To place an object, the program calculates a set of candidate providers to host the object replicas (through REP-SET() in Figure 9) and then contacts the providers sequentially. To locate an object, the program invokes the object location protocol (through LOOKUP() in Figure 9). The parallel object existence checking is implemented through asynchronous network operations. The providers are imitated by P dummy UDP echo servers spread on 30 machines. The responses returned by these dummy providers are ignored by the program. The results are shown in Figure 14.

As we can see, the service time for both object placement and location grows with the number of providers because for each provider, HDCH needs to perform a Hamming distance calculation. However, the total service time is dominated by the network operations, and the service time increase is only modest. Additionally, object placement is more costly than object location, because each placement request actually creates 4 object replicas. Overall, for a 2500 node cluster, it only takes $1.4ms$ or $3.8ms$ to locate

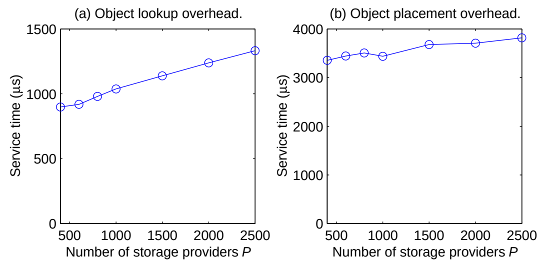


Figure 14: Service time for small object placement and location.

(a) Evaluation metrics.

	Metric	Parameter
(a)	Object lookup time.	P
(b)	Bandwidth.	P
(c)	Bandwidth.	r_c (Note: $r_c = r_d$)
(d)	Memory.	N

(b) Default simulator parameters.

Name	Value	Name	Value
P	1000	V	100GB
r_c	25 req/sec	r_d	25 req/sec
r_l	0 req/sec	I	1 second
T	100 seconds	N	5 Million

Figure 15: Evaluation settings for the overhead of Bloom filters.

or place an object.

Large Object Tracking.

The evaluation of the overhead introduced by Bloom filters is quite complicated. First of all, there are three metrics we would like to measure: object location service time, memory and bandwidth consumptions. Secondly, each metric could depend on three parameters: number of objects, number of providers, and object creation/deletion rate. Due to space constraint, we only present four of the nine possible combinations which are the most interesting to observe. These four combinations are summarized in Figure 15 (a).

To measure the object location service time, we augment the simulator such that in the middle of the simulation, it calls a special subroutine which serves 10000 object location requests and reports the average time for each lookup. For each object location request, the subroutine uses the slave Bloom filters stored in the aggregator to select a candidate set of providers and then contacts them over the network. As in the previous experiment, the dummy UDP echo servers are used to imitate providers. The other metrics are directly reported by the simulator.

We made a few changes to reduce the complexity of the simulation. We modify the request generation module such that it only generates large objects requests. Additionally, no object location requests are generated. Note that these changes will not affect the evaluation results because only large objects are managed by Bloom filters and that the absence of lookup requests will not change the metrics we are interested in. All simulation settings are taken from Figure 15 (b) except for the scaling parameters. Note that the values of N, r_c and r_d in Figure 15 (b) are only for large objects. Based on statistical estimation, this setting corresponds to a large cluster with 90 million mixed-sized objects and a high object creation or deletion rate of 4500 req/sec.

The evaluation results are shown in Figure 16. We can see that the object lookup service time increases with the number of providers, because each object location involves the linear scan of the array of Bloom filters. Secondly, the bandwidth consumption grows with either the number of providers or the object creation request rate. It is understandable because when more objects are created, more changes to the Bloom filters need be propagated to aggregators. When there are more providers, more periodical update packets will be generated, and each update packet will incur a constant overhead (such as the packet headers). Finally, the amount of memory consumed by Bloom filters grows proportionally with the number of objects.

Regardless of the linear increases of these metrics in response to the increases of various parameters, it is important to note that the overhead of object location using Bloom filters is fairly low. For example, an object location request takes only $1.8ms$ to complete with 2500 providers. Additionally, the bandwidth consumption never exceeds 50KB/s, which is insignificant for a Fast Ethernet connection. Finally, Bloom filters only consume 15MB memory to track 10 million large objects, which can be easily accommodated on a desktop PC.

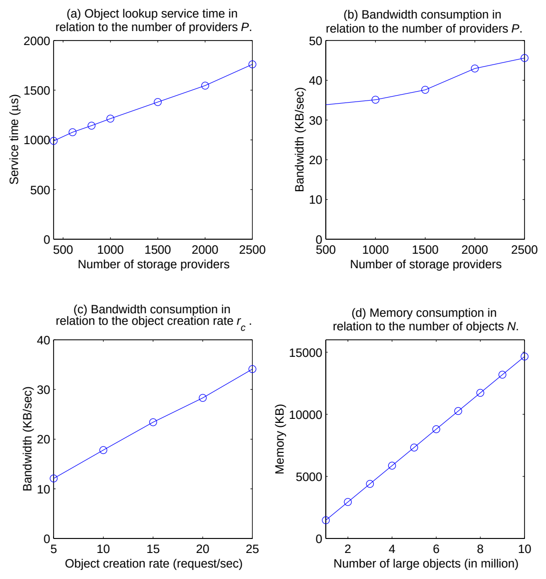


Figure 16: The overhead of Bloom filters.

In conclusion, our differentiated protocol incurs low overhead in terms of protocol invocation (service time) and protocol state maintenance (memory/bandwidth) to scale up to several thousand providers and millions of objects. These results clearly demonstrate the feasibility of our protocol to support large-scale storage clusters.

6 Related Work

Our work is in large part motivated by previous work on distributed file systems [4, 16, 19, 24] and cluster-based storage systems [2, 15, 21]. Previous studies mainly focus on infrastructural support of unifying distributed storage resources and have not addressed adequately how objects should be placed and located efficiently in response to node additions and departures. Xfs [4], Swarm [15], Zebra [16], and Petal [21] organize available storage resources as a group of RAIDs or volumes, which share the same set of limitations as SAN systems.

Slice [2] proposes a software architecture in which storage request traffic is intercepted by a light-weight μproxy routing filter. Slice distributes objects through a combination of hashing and logical-to-physical node mapping table (called *routing table*). Such a scheme is more flexible than volume maps. However, it introduces its own set of limitations and does not completely solve the problem. For example, the centralized management of the routing table could be a single-point of failure. Additionally, if a node recovers from a transient failure, it needs to be mapped to the same set of logical servers as it was before the failure; otherwise, old objects stored on that node would not be located.

OceanStore [20] is a global-scale storage system and objects are cached anywhere in the system and are located through a combination of attenuated Bloom filters and Tapestry [17]. However, their goal is to provide secure and highly available data access over the WAN, and thus has a different set of design constraints and assumptions. For example, the top priorities in OceanStore are security and how to deal with unstable end-to-end network bandwidth; while we assume a trusted environment and nodes are connected with a high-bandwidth low-latency network. Additionally, OceanStore assumes that data objects are cacheable anywhere in the network; while we have to deal with very large data objects that must be fragmented to fit onto physical devices.

Our differentiated object placement and location protocol relies on the different characteristics of small and large objects, which have been studied by Baker et al. [5] and Vogels [35]. Previous researches such as Slice [2], Swift [7] and Zebra [16] have used it to differentiate I/O operations for small and large objects. However, to our knowledge, none have taken advantage of it to optimize object placement and location operations.

Consistent hashing was proposed by Karger et al. [18] and has been applied in Chord [32]. Our HDCH is a variation of the basic approach and exhibits more consistent behavior for variable scales. High dimension topologies can be found in architecture researches such as hypercube network, and has also been adopted by CAN [29] in wide-area index distribution to facilitate request routing for peer-to-peer systems. In comparison, our HDCH design emphasizes on balanced object distribution and maintaining stable hashing results in the event of node additions or departures.

Bloom filters have been used in SummaryCache [11], OceanStore [20] and PlanetP [10]. All these studies exchange filter updates through reliable point-to-point communications over a WAN. In this work, we multicast filter updates over a LAN and devise techniques to tolerate packet losses.

Proactive introspection is inspired by the idea of introspective replica management used in AT&T Radar [28] and OceanStore [20]. Our proactive introspection is tightly coupled with the object location protocol and we have demonstrated how it can be efficiently performed.

7 Conclusions and Future Work

This paper presents the design and implementation of an differentiated object placement and location protocol for large-scale storage clusters. We use high-dimension consistent hashing to place and locate small objects. For large objects, we combine a usage-based placement scheme with Bloom filter-based tracking. The main advantage of this protocol is that it is able to flexibly place objects to live storage nodes and make effective use of storage resources. It is also scalable with very low management overhead in comparison with uniform strategies. The effectiveness and scalability of the protocol is validated with a combination of simulations and experiments.

Our work has leveraged techniques from previous research on distributed storage systems such as Slice [2] and OceanStore [20], and complements the existing work by targeting at the manageability problem of large-scale storage clusters in response to frequent node additions or departures. Based on this protocol, a storage cluster can automatically integrate available resources and repair damaged data redundancy.

The protocol described in this paper is part of the Sorrento project, with the goal of making storage clusters self-organizing and minimizing human administration. Much future work remains. We plan to study how to adaptively distinguish small and large objects. We also plan to investigate how to support automatic resource scheduling, tuning and QoS guarantees. We will analyze the performance impact and possible optimizations for version-based consistency model and erasure coding-based redundancy scheme in the context of self-organizing storage systems. Finally, we plan to conduct a detailed study of workload characteristics of storage clusters used by new data-intensive applications.

References

[1] ANDERSON, D. Object based storage devices: A command set proposal. Tech. rep., National Storage Industry Consortium, October 1999.

[2] ANDERSON, D., CHASE, J., AND VAHDAT, A. Interposed request routing for scalable network storage. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)* (October 2000).

[3] ANDERSON, T., BREITBART, Y., KORTH, H., AND WOOL, A. Replication, consistency, and practicality: Are these mutually exclusive? In *Proceedings of 1998 ACM SIGMOD International Conference on Management of Data* (June 1998), ACM Press, pp. 484–495.

[4] ANDERSON, T., DAHLIN, M., NEEFE, J., PATTERSON, D., ROSELLI, D., AND WANG, R. Serverless network file systems. In *Proceedings of the 15 Symposium on Operating Systems Principles SOSP'1995* (December 1995).

[5] BAKER, M., HARTMAN, J., KUPFER, M., SHIRRIFF, K., AND OUSTERHOUT, J. Measurements of a distributed file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles* (1991), ACM Press, pp. 198–212.

[6] BLOOM, B. Space/time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery 13*, 7 (1970), 422–426.

[7] CABRERA, L.-F., AND LONG, D. Swift: Using distributed disk striping to provide high I/O data rates. Tech. Rep. UCSC-CRL-91-46, 1991.

[8] CHASE, J., ANDERSON, D., THAKUR, P., AND VAHDAT, A. Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating Systems Principles SOSP'2001* (October 2001).

[9] CHUNDI, P., ROSENKRANTZ, D., AND RAVI, S. S. Deferred updates and data placement in distributed databases. In *ICDE* (1996), pp. 469–476.

[10] CUENCA-ACUNA, F. M., PEERY, C., MARTIN, R., AND NGUYEN, T. PlanetP: Using gossiping to build content addressable peer-to-peer information sharing communities. Tech. Rep. DCS-TR-487, Department of Computer Science, Rutgers University, 2002.

[11] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Summary Cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking 8*, 3 (2000), 281–293.

[12] GEOFFRAY, P. Opiom: Off-processor IO with Myrinet. *Parallel Processing Letters 11*, 2/3 (2001).

[13] GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The dangers of replication and a solution. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1996), pp. 173–182.

[14] HANSEN, J., AND LACHAIZE, R. Using idle disks in a cluster as a high-performance storage system. In *IEEE International Conference on Cluster Computing* (September 2002).

[15] HARTMAN, J., MURDOCK, I., AND SPALINK, T. The Swarm scalable storage system. In *Proceedings of International Conference on Distributed Computing Systems* (1999), pp. 74–81.

[16] HARTMAN, J., AND OUSTERHOUT, J. The Zebra striped network file system. *ACM Transactions on Computer Systems (TOCS) 13*, 3 (1995), 274–310.

[17] HILDRUM, K., KUBIATOWICZ, J., RAO, S., AND ZHAO, B. Distributed object location in a dynamic network. In *Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures* (August 2002), pp. 41–52.

[18] KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEVIN, D., AND PANIGRAPHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of ACM Symposium on Theory of Computing* (1997), pp. 654–663.

[19] KISTLER, J., AND SATYANARAYANAN, M. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles* (1991), vol. 25, ACM Press, pp. 213–225.

[20] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (November 2000), ACM.

[21] LEE, E., AND THEKKATH, C. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1996), pp. 84–92.

[22] LEE, E., THEKKATH, C., WHITAKER, C., AND HOGG, J. A Comparison of Two Distributed Disk Systems. Tech. Rep. 155, Compaq (DEC) System Research Center, April 1998.

[23] LUBY, M., MITZENMACHER, M., SHOKROLLAHI, M. A., AND SPIELMAN, D. Analysis of low density codes and improved designs using irregular graphs. In *Proceedings of ACM Symposium on Theory of Computing* (1998), pp. 249–258.

[24] MULLENDER, S., AND TANENBAUM, A. A distributed file service based on optimistic concurrency control. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (1985), ACM Press, pp. 51–62.

[25] OUSTERHOUT, J., CHERENSON, A., DOUGLIS, F., NELSON, M., AND WELCH, B. The Sprite network operating system. *IEEE Computer Magazine 21*, 2 (1988).

[26] PATTERSON, D., GIBSON, G., AND KATZ, R. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (1988), ACM Press, pp. 109–116.

[27] PLANK, J. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software Practice and Experience 27*, 9 (September 1997), 995–1012.

[28] RABINOVICH, M., RABINOVICH, I., RAJARAMAN, R., AND AGGARWAL, A. A dynamic object replication and migration protocol for an internet hosting service. In *Proceedings of IEEE International Conference on Distributed Computing Systems* (1999), pp. 101–113.

[29] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM'01* (August 2001), pp. 161–172.

[30] SCHLAGETER, G. Optimistic methods for concurrency control in distributed database systems. In *Proceedings of 7th International Conference on Very Large Data Bases (VLDB)* (September 1981), IEEE Computer Society, pp. 125–130.

[31] SHEN, K., TANG, H., YANG, T., AND CHU, L. Integrated Resource Management for Cluster-based Internet Services. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002).

[32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference* (2001), pp. 149–160.

[33] TERRY, D., THEIMER, M., PETERSEN, K., DEMERS, A., SPREITZER, M., AND HAUSER, C. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (1995), ACM Press, pp. 172–182.

[34] THEKKATH, C., MANN, T., AND LEE, E. Frangipani: A scalable distributed file system. In *Proceedings of Symposium on Operating Systems Principles* (1997), pp. 224–237.

[35] VOGELS, W. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM symposium on Operating systems principles* (1999), ACM Press, pp. 93–109.

[36] WAXMAN, J., AND MCARTHUR, J. Storage area networking - opportunity for the indirect channel. IDC White Paper, 2000.