Technical Report MSR-TR-2005-179, December, 2005

BitVault: a Highly Reliable Distributed Data Retention Platform

Zheng Zhang, Qiao Lian, Shiding Lin, Wei Chen, Yu Chen, Chao Jin
Microsoft Research Asia
{zzhang, i-qiaol, t-slin, weic, ychen, t-chjin}@microsoft.com

Abstract

In this paper, we report the design and implementation of the storage layer of BitVault: a content-addressable retention platform for large volume of reference data – seldom-changing information that needs to be retained for a long period of time. BitVault uses "smart brick" as the building block to lower the hardware cost. However, the challenges are to maintain low management cost in a system that needs to scale all the way from one brick to tens of thousands of bricks, to ensure reliability and to deliver with a simple enough design. Our design incorporates P2P technologies for its self-managing and self-healing capabilities and uses massively parallel repair to reduce vulnerability window of data loss. The simplicity of the architecture relies on an eventually reliable membership service provided by a perfect one-hop DHT (distributed hash table), and its object-driven repair model yields last-copy recall guarantee: independent of how many other failures that may occur and their sequences, as long as the last copy of a data object still remains in the system, the data can be retrieved and its replication degree fully restored. A prototype has been implemented. Theoretical analysis, simulations and experiments are conducted to validate the design of BitVault.

1. Introduction

Companies today face the problem of managing an increasing amount of *reference data* — seldom-changing information that needs to be retained for its business value or for compliance reasons. The Enterprise Storage Group estimates that[1], by 2005, more than half of the data stored by North American businesses will be reference data (examples include check images, electronic invoices, email messages, etc.). Furthermore, the amount of reference data is growing one and a half times as fast as the non-reference one. The force behind these trends is the digitization of all kinds of data. For example, X-ray images alone produce over 20PB of data every year. Digitizing all phone conferences of a year would have generated 17,000PB worth of data. Also, in the enterprise world, email archiving is of paramount importance due to legal regulations.

Reference data must be kept for an infinite period of time. On the other hand, data must be easily accessible: the SEC-17(a) regulation states that companies must retrieve the required documents in 48hours when a court order is delivered, or face strict penalty. In addition to ease of access, the raw performance of accessing the data should be reasonable, such that search functionality can be built on top of the platform. This is especially important for email archiving applications.

BitVault is a backend storage platform for reference data. Our top-three design goals are 1) low total cost of ownership (TCO), 2) extremely high reliability and availability and 3) simplicity. To achieve these goals, our design combines the latest peer-to-peer technologies and a number of novel techniques. In particular, our main contributions include the followings:

- We employ a weak and eventual membership protocol to organize commodity "smart bricks" into a very large logical space offering a DHT (distributed hash table) abstraction. In contrast to other systems that are either client/server architected or using strong membership protocol, BitVault scales out in a self-organizing manner with low overhead.
- We employ the object-driven repair model that gives objects the central role of the repair process. This model affords the use of soft-state indices and yet delivers the last-copy recall guarantee: as long as an object still has the very last replica, its replication degree is fully restored and the object is accessible as soon as possible. Furthermore this guarantee can withstand arbitrary number of failures and their sequences. Our basic idea is to leverage the service of the membership protocol to rebuild any missing indices if necessary, and the indices in turn repairs missing replicas. This object-centric strategy is very different from the conventional way of dealing with failures, one in which the index is first made to be reliable and consistent, and then functions as the base to react to failures.
- BitVault uses massively parallel repair to significantly bring down the repair window (e.g. minutes instead of hours). Previously, this is achieved only in a centralized-indexed solution like GFS [14].

The simplicity of the BitVault architecture is due to a number of things. The repair model depends on the eventual convergence of live brick membership list, and hence there is no need for a strong consensus protocol at run time. BitVault deals with immutable objects only, and allows extra copies to temporarily exist. Consequently, BitVault does not employ distributed transactions. A prototype of BitVault has been implemented and evaluated in our lab, and the experimental results validate our design choices.

The remaining sections are organized as follows. In Section 2 we elaborate our design goals. Section 3 describes the architecture and the protocols. In Section 4 we describe our implementation. To give an intuition on

how applications may use BitVault, several applications were described in Section 5. In Section 6 we provide experimental results. We discuss the related work in Section 7 and conclude the paper with Section 8.

2. Requirements and design goals

We begin by summarizing the key requirements of a data retention platform of reference data mentioned earlier: 1) very large and rapidly growing volume (hundreds of billions) of small and medium-sized objects (from a few KB of email messages to a few GB of video streams); 2) very high reliability and availability, with good access performance in general.

Accordingly, the design and implementation of BitVault have three top-level objectives, as we elaborate below.

***Goal#1*: *low TCO* (*total cost of ownership*)**. We breakdown TCO into three major components: hardware cost, operational cost and power consumption.

In order to keep the total cost low, a system of such a large scale must ride the economies of large scale. Data retention has been, and still is, dominated by tape libraries, which are expensive to operate, slow to access and are ill suited when we start to look into the possibility of digitizing all kinds of data. In contrast, the price per GB of disk is declining to twice of that of tape and it is therefore justified to backup to disks, instead of tapes. For this reason, we use "*smart bricks*" as our building bricks. Smart bricks are essentially trimmed down PC with large disk(s). We believe that the trend is such that smart bricks will be commodity components, just as PCs are today. This design decision also allows us to have a good access performance, in case search functionality needs to be built on top of the platform. This is especially important for email archiving applications.

Given the longevity of data kept in the system as well as the volume of growth, however, smart bricks of different capability will be procured at different points of time and co-exist in the system. Thus, BitVault must *leverage the inherent heterogeneity* and *support online migration to new hardware*.

In terms of total cost of ownership (TCO), however, the hardware cost is only a small fraction. The management overhead of dealing with the complexity of the system rises quickly with the system scale. To give a total 5PB of raw capacity and with the disk capacity of 500GB, BitVault needs to scale out gracefully upwards to 10K bricks. Of course, we do not believe that initially there will be many instances with such a scale, and thus *incremental scale-out capability* is important. Therefore, BitVault must be as *self-managing* and *self-organizing* as possible: its administration overhead must be low and almost constant, independent of the system scale. Ideally, all that the administrator needs to do is to unpack a brick, install the BitVault software, plug into the system and then forget about it. When a brick fails, it is simply unplugged from the system. All these must be done online, with minimum perturbation to ongoing operations, with no compromise to reliability and availability.

A very large BitVault installment can consume substantial power, and this is where the traditional "cold" media such as tape and CD hold advantages: they cost zero energy. We do not address this issue yet, but believe there are amply opportunities to strike the balance between power consumption and accessibility.

Goal#2*: *extremely high reliability and availability. One of the most important design goals of BitVault is *rapid repair*. In a system of 10K bricks, failure will be frequent, as is observed by works in the context of large scale (e.g. GFS [14]). Some of these failures are transient and only affect availability temporarily, and some others are fatal and impact reliability. In the design of BitVault, we use the term *last-copy fast recall* to characterize a storage system's capability to deal with failures: as long as an object still has the very last replica, its replication degree should be fully restored and the object is accessible as quickly as possible. Therefore, rapid repair is essential. BitVault's strategy is to leverage the scale of the system and spread the repair load so as to utilize all the aggregated network bandwidth, with a target repair window of minutes instead of hours.

A great subset of the data stored in BitVault may need to tolerate site disasters. Since the WAN bandwidth is limited, it is desirable that we can accommodate "replication by mail"[16]. BitVault introduces the concept of *self-identifying bricks*: a brick primed with objects obtained elsewhere can be FedEx-ed to a different site, plugged in and let its content properly replicated and readily and accessible.

Goal#3*: *simplicity. A complex system of such a scale is difficult to get right, and the first two objectives are already ambitious. Thus, the unspoken rule of BitVault is to strike at simplicity as much as possible, so that the system behavior is provable and correct. Many of our decisions, such as soft-state index (3.3.1), object-driven repair model (3.3.2) and the avoidance of global and transaction protocols are founded upon this principle.

System model: BitVault operates in a controlled environment and all hardware is assumed to be trusted. Bricks are connected with high-throughput and low-latency LAN, and we assume Gigabit Ethernet as the switching fabric, though our prototype uses 100Mb switches. Transient failures include instances such as brick reboot; cable disconnection, switch failures etc.; they can affect multiple bricks and become sources of

correlated but transient failures. For the time being, we do not consider network partitions. All other failures such as disk crashes are permanent and fail-stop.

3. Design

Interface: BitVault stores *immutable* objects, with *checkin* and *checkout* as the two primary APIs. A 160bit *key* is the handle to retrieve an object. Each object has a unique key and the key distribution is uniform. If the key is derived from hash of the object, as is done in our prototype, then BitVault becomes *content-addressable*.

Given that BitVault's main objectives are self-managing, strong scale-out capability and very high reliability, our design combines the latest peer-to-peer technologies and a number of novel techniques. Specifically, we adopt the DHT logical space as the primary abstract for self-organizing and scale-out capability. While index partition is DHT-based, replica placement is policy controlled, allowing massively parallel repair for a failed brick. To avoid complexity, we use *object-driven* repair model, which leads to soft-state indices that are far simpler to implement, and yet guarantee last-copy recall.

We will start with the high-level system architecture and discuss other design alternatives that we have considered. We will then describe per-brick architecture, followed by individual protocols. Some of the components can have alternative implementations and thus their details are left in the next section.

3.1 High-level system architecture

When we talk about a scalable architecture, the classic client-server architecture often falls out of favor. Yet, works such as GFS [14] have demonstrated that this structure often works and works well. In this architecture (Figure 1(a)), a set of masters manages the bricks storing individual replicas. The masters keep the integrity of the indices, and typically run a replicated state machine (or just hot-standby) to ensure that they act as one entity and endure failures.

However, the architecture is ill-suited for a data retention platform. First of all, unlike GFS in which object size is 64MB, reference data is small. As such, the size of the index will be huge and difficult to fit in the memory of the master (as is done in GFS). This raises both performance and scalability concerns. Secondly, assuming a total of *N* bricks in the system, the O(*N*) periodical beaconing from the master(s) to the bricks can consume substantial resources. If the beacon interval is 10s, then for a 10K brick installment, a master needs to process 1K beaconing every second. None of the above is a real concern if the master does not impose heavily on the access path. GFS is a file system, and can leverage client-side leasing so that client requests can go directly to the bricks most of the time, bypassing the master when lease is valid. Unfortunately, for a data retention platform, it is unlikely that access locality exists, if at all. Hence, the master is likely to become a performance bottleneck, especially when we talk about a 10K scale.

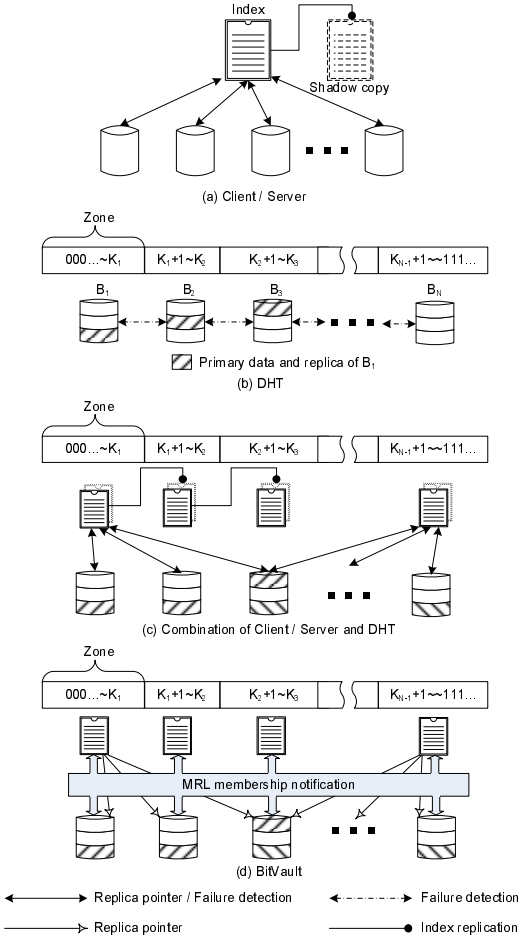


Figure 1. Different architectures for a scalable brick-based system: (a) client-server; (b) DHT, (c) DHT + client-server and (d) DHT-based index partition, policy controlled object placement with a membership service.

The next choice is to base the architecture over DHT (distributed hash table [27][29][31][37]). After all, many wide-area archival systems – even file system, have been proposed (e.g. Ivy[24], CFS[10], PAST[30] and Pastiche [9]). If these systems can handle high churn rate and run in untrusted environment with low resource consumption (often O(log*N*)), they would appear to operate well in a controlled and less dynamic context that BitVault targets. In fact, an earlier version of BitVault, called RepStore [35] was designed with this architecture.

In DHT, nodes join a large logical space (e.g. 160 bits) with random IDs, and thus partition the space in a self-organizing way. The portion of the space a node is responsible for is called its *zone*. For instance, *x*'s zone is (*y.id*, *x.id*] (e.g. Chord [31]), where *y* is *x*'s immediate predecessor in the logical space. A node whose zone covers an object's key (typically the hash of the object) is called the *root* of the object. Typically, all current schemes enforce an invariance such that a number of replicas of the object is placed on a set of logically consecutive nodes, starting from the root brick (Figure 1 (b)). As it is, DHT-based system has no need for global metadata at all, and presents to the upper-layer application with a reliable object store interface. On average, starting from any arbitrary node, lookup an object using its key to invoke checkin/checkout operations at the root node takes O(log*N*) network hops.

In the context of BitVault, we found that the most useful concept of these proposals is the self-organizing logical space. However, there are a few serious problems:

- First of all, uniform node ID yields an exponential zone distribution. Since object keys are uniform (especially when they are hashes of objects), storage utilization across nodes is uneven. This becomes a major problem when raw brick capacities differ, as will be the case in any real BitVault deployment. Giving up the control of object placement is even more problematic when applications demand that certain objects to co-locate.

- Second, during data repair, each of the *k* consecutive bricks after the failed brick needs to share one-*k*th of the load of the failed brick, where *k* is the replication degree. That means each node has to keep about one-*k*th of free space for data repair, so disk utilization is less than optimal in DHT-based scheme.

- Third, adding empty bricks into the system will result in data movement, as dictated by the invariance of the object replication. Ideally, data copying should only occur when there is a need to repair. This overhead exists to purely satisfy the invariance, has nothing to do with reliability, and further contributes to protocol complexity.

- Fourth, DHT-based scheme does not support self-identifying disks. When a brick already loaded with objects is plugged into the system, all objects have to be redistributed according to its key, and this will take hours to complete.

- Finally, probably the most serious issue is that this approach constrains the repair speed of any single brick. The semantic of DHT-based replication implies that the sources and the sinks to repair the content of a failed disk are restricted to a few in its logical neighborhood, and thus it cannot achieve a high degree of parallel repair. In contrast, if replicas can be randomly placed, repair can proceed in parallel because the contents of the failing brick have many sibling replicas stored on many bricks, and repaired to equally many other bricks.

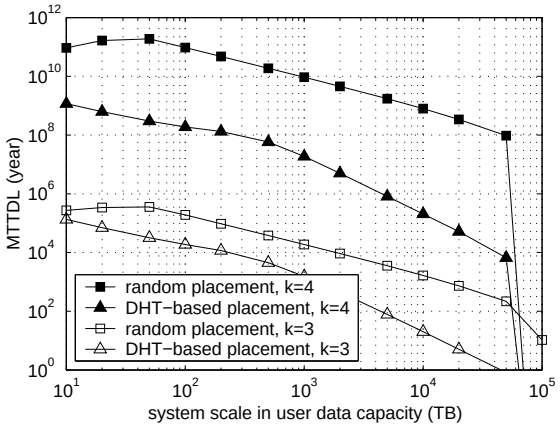


Figure 2. Analytical result on data reliability as measured by *MTTDL* (mean time to data loss) for various system scales, replication degrees, and placement schemes. The average object size in random placement is 10MB. Each brick has capacity of 100GB, with disk bandwidth 5MB/s. The network aggregated bandwidth for data repair is 3GB/s. Mean time to failure of each brick is 1000 days. Each brick fails independently with exponential distribution.

We diverge from our course of architectural discussion a little by presenting some analytical results that support our argument. Figure 2 shows the result based on an analytical framework we developed to study brick storage reliability [20]. The important point is that random placement provides orders of magnitude better reliability than DHT-based placement. However, our study [20] also reveals that when the object size is small and thus the number of objects is large, pure random placement at the object level may also suffer degrading reliability. This is because when the number of objects is large, pure random placement exhausts all permutations of placement and thus is very sensitive to multiple concurrent disk failures. Our work in [20] proposes modifications to overcome this problem in order to deliver high reliability consistently across a wide range of object sizes. For the discussion here, it suffices to point out that more sophisticated placement is necessary.

The key point is that we need to decouple object placement with the logical space. BitVault's strategy thus employs DHT-based index partition but with policy-controlled replica placement. On the surface, this can be achieved by combining the client-server architecture

and DHT, as shown in Figure 1(c). In this approach, the index of an object -- instead of the object itself, is kept at the object's root brick, and the index further points to the *k* bricks storing the replicas (*k* being the replication factor and can vary from object to object). This gives an architecture that meets the majority of the requirements: accesses are fully distributed, repair can be delivered in parallel, heterogeneity can be leveraged, and there is no need to move data any more when adding new brick.

This naïve approach, however, has a number of problems. If replica placement is purely random, then the total failure detection traffic will amount to $O(N^2)$, with each brick handling exactly the same amount as in the client-server option. Complexities increase as well. In addition to the DHT protocol that manages the space, we still need to run the same replicated state machines in every pair of logically neighboring bricks to keep the index reliable and consistent.

The core of the above dilemma lies at the mindset inherited from the client-server architecture, which places the index at the center of the design: it must be kept reliable and consistent, which is a precondition to trigger repair upon failure detection. This need not be the case. The last-copy recall property provides both challenges and hints to the solution: in fact, the replicas should be given the central role. As long as the replica is notified when its index is gone, it can initiate repair to rebuild its index. The index, in turn, can initiate replica repair if it observes the loss of replica(s). We call this the *object-driven* repair model. In order for this to work, a weakly consistent membership service with the following guarantee suffices: any change to the logical space due to brick addition and departure is eventually and reliably known to every live bricks. In BitVault, we call this service the *Membership and Routing Layer* or MRL in short; this is so because MRL is also responsible to act like a one-hop DHT to route message to its root brick. Since indices are distributed to all bricks and they can be reliably rebuilt, with the help of MRL we can afford to use *soft-state* indices that are kept in memory, achieving all the remaining goals, as we will discuss in more detail later.

In BitVault, all bricks participate in maintaining the MRL. Many weakly consistent membership protocol exist, with low maintenance overhead typically of O(log*N*). BitVault's MRL is implemented using XRing, a *perfect* one-hop DHT (Section 4.1), with the insight that the semantic is equivalent to a weakly consistent membership protocol. The layering approach gives a clear and clean division of responsibilities, and is critical to the simplicity of the BitVault design.

3.2 Per-brick architecture

The components of a BitVault brick are shown in Figure 3. The MRL module is responsible for two things. First, it implements the eventual membership protocol and keeps a full list of all live bricks. It notifies the DM and IM modules upon any change of the list. Second, MRLs of all bricks collectively form a DHT. Brick ID is a random number of 160bits, and the ordered list defines the zone of any live brick: the zone of a brick *x* is (*y*.id, *x*.id], where *y* is *x*'s immediate logical predecessor in the space. This is basically consistent hashing as in Chord [31]. MRL provides the routeTo(key, msg) service to route a message, typically in one network hop, to the brick whose zone owns key.

Figure 3. Components of one BitVault brick

Index module (IM) keeps the indices for any objects that are rooted at this brick. The index is soft-state, and records *k* pointers to the actual locations of the replicas, where *k* is the replication degree of the object. IM listens to MRL for membership changes and issues repair for missing replicas if necessary.

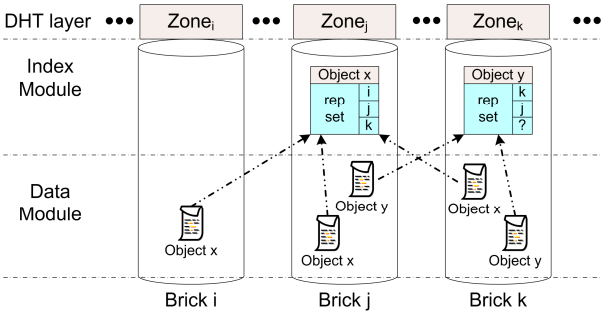


Figure 4. Example of the object layout in BitVault. Object *x* has replicas in brick *i, j* and *k*, whereas object *y* has replicas in brick *j* and *k*.

Data module (DM) stores replicas to local disk. Along with the object we store a few metadata, including its key and the specified replication degree. A reverse table is built in the memory, and the table entries record the physical addresses of the root bricks of the objects. This

table is used to perform indices repair, also triggered by the MRL notifications.

Finally, the Access module (AM) is a very light module that serves as the brick's gateway to external client requests. Figure 4 shows the relationship of these components with some replica placement examples.

3.3 Individual protocols

3.3.1 Checkin/checkout

The checkin request carries the object, the replication degree *k*, supplied from upper-layer application and the object's key. The checkout request needs the key as the only parameter. Both requests can be submitted to an arbitrary brick in BitVault.

When performing checkin, the brick will invoke the *placement policy* (detailed in Section 3.3.4) to pick *k* bricks, and send replicas to these bricks. When the first acknowledgement is received, the checkin procedure is completed from client's perspective. If desired, the checkin can wait for more replicas' responses.

Figure 5. Object index state machine.

A brick persistently stores a received object to its DM and at the same time *publishes* the object with an ipublish message. Out of all messages in BitVaut, this is the only one that needs to be reliable, and we ensure this with ack/resend mechanism. This message uses the object's key and is routed through MRL to reach the root brick; the message also includes the specified replication degree. The ipublish message is one of the events that trigger the index state machine (Figure 5) at the root brick. An index has two possible states: *partial* and *complete*. A pointer in the index is valid if and only if the brick pointed to is alive and contains a copy of the object. A complete index has the number of valid pointers equal to the specified replication degree; whereas a partial index is the one with fewer valid pointers. Therefore, when receiving the first such message, a root brick will start a partial index with one valid pointer. At the same time, it will start a timer. If *k* such pointers are collected, the index becomes complete and the timer is stopped. However, if the timer expires before enough pointers are collected, repair will be triggered.

Checkout is simple; the request is routed to the root brick, and follows any one of the pointers to retrieve the object.

3.3.2 Repair of permanent failure

The last-copy recall property dictates both the accessibility and the total restoration of replication degree as long as the very last copy survives, independent of any other component failures and their sequences. For example, right after we restore another copy, both the source replica and the index disappear simultaneously. In this case, the last-copy transit from one replica to the other and this sequence can occur infinitely number of times. However, once the system stabilizes, both the index and *k* replicas should be intact.

Translating the last-copy property to the BitVault data structure, we have the following properties: a) eventually an object's index is always found at the root brick of the object, and b) eventually all indices should be in the "complete" state. BitVault relies on the membership service provided by MRL and the reliable ipublish message to deliver both, and with a very simple set of mechanisms.

- Indices repair: the DM filters membership change events sent from MRL. For any object whose root shall now change to a different brick (either due to brick failures or additions), DM issues the same ipublish as it receives the object the first time towards the new root via MRL. The first ipublish message establishes a partial index at the root; the rest *k*-1 messages turn the index state to complete and hence repair the index. Since indices are partitioned across all members, index repair occurs when there is any membership change. An optimization to improve index availability is to lazily backup *x*'s indices to *x*+1, so that *x*+1 can serve with the cached indices when *x* crashes and *x*+1 takes over.
- Data repair: the IM filters membership change events sent from MRL. For any index that has replicas in a failed brick, the IM changes its state from complete to partial and instructs one of the replica keepers who, after consulting the object placement policy, inserts another copy to the selected brick. Data repair occurs only when brick crashes.

These are the only necessary steps. Notice that when a new replica is made, the receiving brick will generate an ipublish message towards the root, and the message changes the corresponding index's state to complete again and thus closes the repair cycle. Should anything interrupt this distributed procedure, the fact that the index will stay in the partial state means that repair will

continue to happen. This is true even when multiple failures occur (e.g. the index and several replicas are gone simultaneously). While an elaborate proof is out of the scope of this paper, we offer an informal argument that these set of protocols deliver the last-copy recall guarantee.

We assume that failures have wiped out all but the very last copy. The membership service of MRL has the following properties: eventually and with high probability in O(log*N*) time bound (where *N* is the number of bricks in the system), every live bricks is known to every brick, whereas every failed brick is excluded. Since the membership list defines a DHT space, this means that the last copy can watch the change of its root, and hence publishes towards the root the ipublish which includes the specified replication degree. Notice that this can go on even if the root changes *infinitely often* (due to brick crash or addition), and if the last-copy transit from one replica to the other. This message will start a partial index and the repair timer which, when expired, will instruct the last copy to insert new replicas into the system. The cycle is forced to its closure if and only if enough replicas are generated and the index state is changed to complete.

We note two key properties here. First, the repair strategy is *object-driven*. Indeed one can say that repairing missing replicas is triggered by the index being at the state of partial, but the index itself is generated from any surviving object replicas. This is different from many existing approaches that rely on the robustness of index coupled with direct monitoring to data so as to ensure availability. Second, the contents of a DM are pointed to from IMs of many different bricks, and their sibling replicas are spread across the whole system. Thus, both repair triggering and repair source are distributed, and this is the basis of rapid and parallel repair. Figure 4 illustrates both points: if brick *j* fails, index repair for object *x* can be triggered by either brick *i* and *k*, and data repair for object *x* and *y* can be processed by *i* and *k* in parallel.

A simple calculation can show the gain of parallel repair. When using parallel repair, we need to consider the network bandwidth, especially the bandwidth of the root switch since it may become the bottleneck. Let B_{BRICK} be the disk I/O bandwidth, B_{NET} be the available bandwidth of the root switch for data repair. Then the *parallel repair degree* N_r, the number of repair source-destination pairs that can participate in repair in parallel, is given by $N_r = B_{NET}/B_{BRICK}$. N_r is the repair speedup, if the object replicas are spread evenly among roughly N_r bricks. For example, if the disk bandwidth B_{BRICK} = 5MB/s, and the available root switch bandwidth B_{NET} = 1GB/s (67% of a 1Gigabit 24 port switch bisection bandwidth), then N_r = 200, which means instead of taking more than one day to repair one failed disk with 500GB data, the parallel repair can be done in 8 minutes. This immensely reduces the repair time and thus the vulnerability window. Therefore, spreading replicas among a large number of bricks can achieve much faster data repair speed.

3.3.3 Brick additions

A new brick is ready to join the service after it installs the BitVault code. It takes a random ID and contacts any of the existing bricks. As part of the MRL protocol, all live bricks will include this new brick into their membership list; likewise, the new brick acquires the same list as well. This typically converges in O(log*N*) time. Since BitVault uses consistent hashing to partition the space, for any objects whose root changes to the new brick, their hosting DMs will issue index repair to build indices onto the IM of the new brick. Similar to the optimization that improves index availability when dealing with brick failure, when *x* joins, brick *x*+1 can split its indices that belongs to *x* and sends it to *x*, so *x* can have a cached copy of indices to begin serving.

If the new brick is empty, typically the background load-balance process will kick in to move some replicas to the brick. If, however, the brick comes with some objects already, it will initiate index repair for these objects via the ipublish messages towards their roots, and then data repair will be triggered to replicate the objects to other bricks. This is how the object-driven model implements self-identifying brick.

3.3.4 Load-balance

When the system evolves with brick failures, brick additions and data repair movements, the storage load on bricks is likely to be unbalanced. Unbalanced load reduces object access performance since overloaded bricks become bottleneck while underutilized bricks are mostly idle.

To address the load balance issue, BitVault performs background load balancing operations. Periodically, each brick queries an in-system monitoring utility SOMO [36] (further discussed in the implementation section) to gather the information about the average load and low-load bricks in the system. If the load of the current brick is over a certain threshold than the average load (in our prototype it is set as 5%), then the brick will randomly pick some replicas on it and move them to the low-load bricks. As before, the bricks receiving the replicas will send ordinary ipublish messages to build indices. When the source brick receives confirmation from the sink brick that a replica has been created there, it issues a delete message to the root of the object.

The delete protocol is coordinated at IM. It first removes the pointer to the replica to be deleted, and then insert this pointer to a delete pool. The IM then picks an entry from the delete pool and issues a delete request to the target brick. It will keep on retrying until an ack is received. The entry is removed from the delete pool if 1) the ack is received or 2) the target brick crashes (as notified by MRL). The protocol works correctly and ensures that there is never a situation where we have a dangling pointer. The worst can happen is that there is a replica that the index is not aware of, and this occurs if the delete pool, as a soft-state in memory, somehow is corrupted. In this case and all others where the local state (including the index) may be bad, we simply reset the brick itself and let transient failure handling to fix the problems.

The delete protocol is not exposed as an API, but it can be invoked not only by the load-balancing process, but also for other garbage collection purposes as we will discuss shortly.

3.3.5 Dealing with transient failures

In the context of BitVault, many transient failures can occur: reboot as a result of software upgrade, cable drop, switch failures, power failures etc. In these cases, some data may become inaccessible for a short period of time, and as long as some bricks are alive, the system can still operate, albeit with reduced performance.

The primary difficulty in dealing with transient failures is that it is hard to tell whether a failure is transient. It is true that in some instances such as software reboot, there may be a way of informing the nature of the disruption. However, in general, that only adds administration overhead, which is what we want to avoid at the first place. One can delay the triggering of repair, hoping that the affected components can return online soon. However, this only enlarges vulnerability window if the failure is in fact permanent.

Our strategy is to initiate repair regardless. The worst that can happen is that, when the bricks come back online, extra replicas exist. We set a high watermark (e.g. k+1) and when the total number of copies exceeds that threshold, we will start deleting until total copies equal to k. Notice that if future failures reduce replicas to k or above, no repair is triggered. Also, if the watermark is equal to k, then eventually the replication degree is strictly enforced. This strategy is the same as what is proposed in TotalRecall [5].

3.3.6 QoS control

BitVault needs some QoS provisions in order to guarantee stableness. First of all, MRL messages are delivered and processed with the highest priorities. If they are jammed, false failures may be declared, resulting in cascading false repairs which will eventually cause the system to collapse. Ideally, other protocol messages should be prioritized accordingly as well, but we have not implemented them yet. Secondly, for a very large installment, even though BitVault can quickly process repair, there is no guarantee that there shall be no concurrent repairs of multiple bricks in the system. Just as in GFS[14], repair of objects that have lost more replicas should take higher priority. This is governed by a set of rules that run at the repair source (i.e. the DM module of a brick who is instructed to make another copy in the system):

- A repair quota (in terms of bytes/second) is enforced. This is the upper bound that a brick can copy out replicas and hence occupy network resource for the purpose of repair.
- The repair request, generated from the IM that keeps the index, carries number of remaining replicas. With this, the repair source can calculate locally the repair ranks of all pending repair requests. Higher ranked repairs are those that have lost more replicas, and take higher priorities. Requests of the same rank are ordered according to the failure time of the departed brick.

These rules are simple and practical, but they are not complete. For instance, it is possible for a brick to contend network resources with repair requests that have lost only one replica, while there are ongoing repairs of higher ranks initiated from other bricks. Also, there is a natural tension between enforcing QoS quota and maximizing repair speed. This remains as one of our ongoing research work.

3.4 Discussion

The design we have described achieves all design goals iterated at the beginning of this section. Adopting a layering design and leveraging a weak and eventual membership protocol allows us to scale out with a fully distributed architecture, deliver last-copy recall and rapid repair, all without the need of any global consensus protocol or distributed transaction protocols.

There are several fundamental reasons. First of all, there is *already* a global agreement before a brick starts its life in BitVault, namely it is joining a logical space composed by all live bricks. To deliver the last-copy recall, only the eventual consistency of the membership is necessary, hence there is no need for a strong consensus at run time. Dealing with reference data means that we can work with immutable objects. Had we wanted to support in-place updating, then without any doubts we must employ transactions. The decision to allow extra

copies to temporarily exist is also important. We have described earlier that extra copies will be generated when handling transient failures and doing load balancing. Moreover, even when replicas are fully installed but the associated ipublish messages are delayed, the repair timer at the index will trigger new round of repair, also resulting in extra copies. These copies will eventually be garbage collected. Since storage capacity is becoming far less an issue and that in no time we have compromised the correctness of the system, we believe that this is a right tradeoff to make.

The current design is fully distributed and each brick's functionality is completely symmetric. However, if required, with very little change we can accommodate design points between fully distributed/symmetric and centralized. We can divide the logical space into two equal halves, let indices be on the one half and all replicas be on the other (controlled by the object placement policy). Thus, bricks in half of the space are serving indices, and the rest are storing replicas. In the extreme case, there can be only *one* brick on the index half, and hence this becomes essentially a GFS-like system. The density of the bricks in either half of the space can be dynamically adjusted. This is the flexibility brought by working with a DHT-like logical space.

4. Implementation

4.1 Membership and routing layer

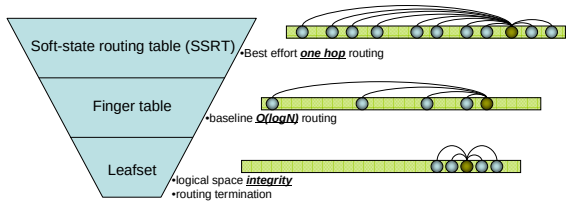


Figure 6: the 3 layers of routing tables in XRing and their corresponding functionality.

We have defined the two responsibilities of the MRL earlier. First, it provides an eventual membership service: once the system is stabilized, every live brick will eventually include in its membership list all of the active bricks *only*. The convergence should be as rapid as possible. Second, it should give an abstraction of DHT. Many of the other design decisions, such as soft-state index, object-driven repair model as well as rapid and parallel repair, depend on MRL.

An eventual membership service does not require the agreement among bricks on the intermittent membership views of the system. Therefore, more expensive view-based group membership protocols (e.g. [6], [8]) are not necessary. Many eventual membership protocols exist, such as SWIM [11]. These protocols gain their scalability by dividing the protocol into two correlated parts: failure detection and failure dissemination. Since MRL needs to function as a DHT also, we choose to extend a best-effort one-hop DHT called XRing[34] to avoid re-implementing a full membership service. The key insight is that an eventually *perfect* one-hop DHT implements just that. Such a DHT is the one that lookup is resolved always in one-hop when the system stabilizes.

XRing divides a 160bit logical space with participating nodes using consistent hashing as in Chord[31]. Each node in XRing has a three-layer data structure maintained by three protocols (Figure 6). The first two layers are rather conventional. The lowest one is the *leafset*, which is a set of 2L+1 nodes including L closest nodes on each side of the DHT logical space plus the home node itself. The heartbeat messages carrying the full leafset of a node are sent between every pair of leafset nodes to maintain the leafset data structure. Leafset members use a voting mechanism for detecting and broadcasting brick leave and join events to reduce erroneous detections. The middle layer consists of a *finger table*, which contains O(log*N*) entries to implement a straightforward O(log*N*) prefix-based routing algorithm. A node's *i*-th finger points to the node that owns the key that is identical to the node's ID except with the *i*-th bit flipped. Regular probing messages are sent to finger table entries to detect failures and repair the finger table. Finally, the third layer SSRT (*soft-state routing table*) enables one-hop lookup performance with high probability. SSRT is maintained by broadcasting node join or leave events detected by the leafset heartbeat protocol using a scalable broadcast through finger and leafset members. The SSRT structure of XRing already contains most brick membership information, but does not satisfy the eventual reliability because the broadcast, though has O(*L*+log*N*) redundancy, is best-effort.

To enhance the SSRT structure of XRing to provide an eventually reliable membership service, we add a background anti-entropy protocol so that bricks can periodically reconcile missing membership information with other random nodes in the system. More specifically, at some regular interval, each brick *x* computes a signature based on its local SSRT, and sends it out to a brick *y* randomly selected from SSRT. When *y* receives the anti-entropy message, it compares with the signature computed from its local SSRT, and if it's different from the received signature, it sends its SSRT back to *x*. Brick *x* merges its local SSRT with the SSRT received from *y*. If *x* detects that its local SSRT is actually more up-to-date, it sends its SSRT back to *y*. To guarantee that the latest leave or join event about a node is the correct one reflecting the node status, timestamps are used on events. Therefore, in order to achieve fast SSRT reconciliation, in an anti-entropy round a brick is

both trying to pull an SSRT from and push its own SSRT to a randomly selected brick. We optimize the protocol such that when the delta is only one missing event, only that event is reconciled instead of sending the whole SSRT.

With the periodic anti-entropy protocol, bricks can quickly resolve the differences in their SSRTs in one or a few anti-entropy rounds, ensuring that eventually every brick will have all the latest membership change events. We have verified this through extensive simulations as well as theoretical analysis.

The anti-entropy protocol is also used when a new brick joins for the first time or rejoins after leaving the system for a while. In this case, the new brick either has no SSRT at all or a possibly outdated SSRT, and the anti-entropy protocol will quickly bring its SSRT up-to-date.

It is interesting to see how XRing implements the two stages of a weak membership protocol: the leafset detection corresponds to membership change detection in a local range, fingers maintain a structured graph for fast event dissemination, and that randomized anti-entropy gives the eventual convergence guarantee. Using a DHT to implement the membership service has its advantages. For instance, the loads of failure detection are evenly distributed, and that node join is handled by default.

4.2 In-system monitoring utility

The task of an in-system monitoring utility is to gather various statistics, filter and aggregate them, and disseminate the results back to each brick. These statistics are necessary to guide replica placement at check-in, repair as well as load balance time.

This functionality is delivered by an improved version of SOMO[36], a self-scaling and self-organizing metadata overlay layered over any DHT. The basic idea of SOMO is to draw a logical tree with a fixed fan-out (e.g. 8) first. The positions of the tree nodes can be calculated by each brick independently. Given its responsible zone in the DHT, each node selects the highest logical tree node that it hosts as its representation in the SOMO hierarchy, and then calculates the position of the parent logical node, routes to that parent tree node to form a child-parent link. A hierarchy is thus built in a self-organized fashion. The SOMO hierarchy is completely self-governing and self-healing, and can gather and disseminate metadata in $O(\log_{fan\_out} N)$ time.

Periodically (e.g. 5s), the top-*n* and bottom-*n* list of disk-usage information are obtained by performing merge-sort when they are gathered towards the root of the SOMO tree. Total storage utilization is aggregated along the upward path as well, allowing each brick to calculate the average load individually. These metadata are then propagated downwards through the SOMO hierarchy to reach every brick. In our implementation, *n* is 500 and the SOMO fan-out is 8. We note that other alternatives such as RanSub[18] can accomplish the same functionality as SOMO does.

4.3 Prototype strategy

BitVault is prototyped entirely using a tool we have developed called WiDS (WiDS implements Distributed System). WiDS combines three aspects of a typical development process: prototyping and debugging, large scale simulation and deployment. WiDS defines a message-passing API and also includes fundamental utilities such as one-time and periodical timers. Protocol logics are written using these APIs and timers. Messages and events can be queued into an event-wheel, enabling many instances of the protocol logics to be debugged within one process while causality among events is enforced. We can emulate wide-area conditions by specifying simulated latency and packet loss over arbitrary pair of communication ends. This allows us to understand how the system behaves in different network settings and also stress different code path. To speed up simulation, WiDS also has a parallel and distribute simulation version. We have successfully simulated complex protocols for 1 million nodes scale, using 250+ machines. Finally, when the protocol code is relatively mature, we re-link it to a different WiDS package which uses sockets to send messages, thus produces an executable that can be deployed and run with real network. In this mode, preliminary logging supports are provided. In the future, we plan to log enough events so that we can replay them in the debug mode. One important point of WiDS is that there is *no* code divergence: the core logic remains the same in every aforementioned stage. All components of BitVault and WiDS are implemented using C++. Currently, BitVault, XRing, SOMO and WiDS have about 6K, 4K, 2.5K and 7K lines of code, respectively.

The prototype includes most of BitVault's key features, except some advanced QoS control of repair and the optimizations to improve index availability.

5. Building applications over BitVault

Any complete applications that use BitVault as the backend storage must incorporate some mechanism to manage the object IDs. One solution is to set aside a SQL database for this functionality. However, the database server is single point of failure and, under heavy loads, a scalability bottleneck and single point of failure. We explore another alternative by using the Catalog

utility which builds application-level and soft-state index *inside* BitVault.

When an object is stored into BitVault, it can optionally take a "tag" which is persisted to disk along with the replicas. The tag must contain a keyword and a descriptor, both supplied by the applications. Later, the application can use the hash of the keyword to retrieve a list of objects that share the same keyword. This list is called a *catalog*, each entry of which is an <OID, Descriptor> pair. Catalog is entirely soft-state and is built in the same way that the object index is built: the node that receives a replica, when seeing its tag, publishes towards the node that contains the hash of the keyword. The node receives the tag then appends the entry to the catalog with the specified keyword. If membership protocol indicates that the node covering the keyword of a catalog changes, we rebuild the catalog by republishing the tags. This is a simple and robust mechanism to add metadata management support inside BitVault.

We now discuss two BitVault applications, both of which respond to day-to-day requirements from users in our lab and are ready to be deployed.

BitVault Client Utility (BCU). BCU allows users to backup and retrieve their files (documents and project files) from any desktop as long as they can connect to a BitVault store. Interestingly enough, in many cases users do this via the mail server. In BCU, a client piece is fully integrated with Explorer, upon right click the user can choose to checkin the file or directory, or retrieve its version history and select one to checkout. A tag is always generated and checked into BitVault along with the object. The keyword of the tag is the hash of a user's account name, and the descriptor is the complete path of the file name and optional text annotation. Thus, inside BitVault there is a complete catalog corresponds to a user. BCU can retrieve this catalog keyed by hash of the user account, parse and load into an Access database file so the user can perform simple queries and checkout different versions of files.

Machine Bank. Like many research institutes, MSRA has a large shared-lab for hundreds of intern students. The shared-lab scenario is such that there is a tension between flexible resource utilization and productivity. A student may get a different PC across different working sessions. It is therefore important to preserve their entire working environment across sessions, or otherwise the students will frequently use the server of their associated research group, and consequently reduce a capable PC to a dummy terminal. In Machine Bank, analogous to the safebox of a banking institute, PCs in the shared-lab run Microsoft's Virtual PC. A VM (Virtual Machine) is broken into 64KB blocks and stored into BitVault. Since majority of the VM images across different users and time are the same, thus most of the blocks are the same. This avoids the problem of having each VM takes its entire space, Each PC also implements local persistent cache to improve the performance, and at the end of a session only modified blocks are checked into BitVault. The mapping between the blocks and their hash is captured in a file called Virtual Machine Instance (VMI). When all modified blocks are checked into BitVault, VMI is checked in as well with a tag which uses the hash of the user account name as the keyword. Thus, a catalog of the user's VM images are built and stored inside BitVault. At the beginning of the login, the user can select any VM instances in the past to reinstantiate at the current PC, thus accomplishing the task of seamless work environment migration both across time and space. More details can be found in [].

-- old stuff below --

Figure 7 illustrates a prototype application we have built on top of BitVault. The Web application presents a Microsoft Sharepoint-like interface, displaying objects for which there is a local copy. A SQL server stores, for each object, the hash key, metadata such as replication policy, user supplied descriptions and finally its audit trail (the access history). Simple queries can be issued against the SQL.

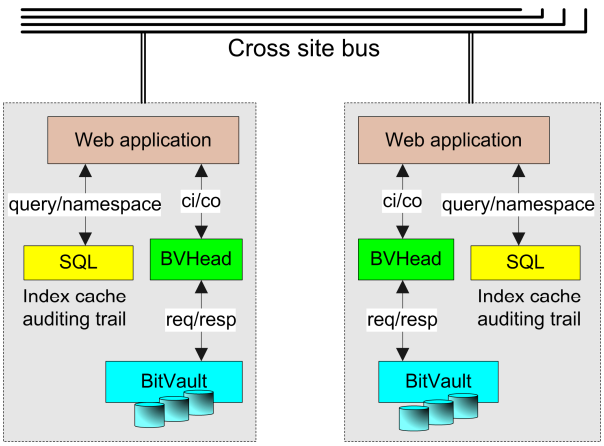


Figure 7. BitVault cross-site architecture.

In this application, multiple geographically distributed BitVault sites can link up for disaster-tolerance. As a user-specified parameter at the time of check-in, an object can be replicated within a site only, or across multiple sites. If an object can not be retrieved from the local site, the checkout request is sent to other sites that have the replica, which is then re-inserted into the local site. The set of sites where an object is replicated is also part of the check-in parameter and kept in SQL.

We have set up BitVault with 5 sites: two in our Beijing lab, two in Redmond each at different building, and finally one in Silicon Valley. The setup survived several

unplanned downtime (moving an entire site to a different location, or unplugging cables).

While the first application uses database to do the bookkeeping, we are also evaluating the option of leveraging per-client's local file system for interactive and online backup using BitVault. One design lets user select what files and/or directories they want to backup. Each file can thus have three states: no backup, local copy + backup, and backup only. This gives the user the flexibility of moving and reclaiming capacity between his local machine and his allotment in BitVault. A small database file that records the files being backed up is also backed up into BitVault, and is accessed with a unique key known only to each user. This allows the user to reborn his backed up files on another machine.

Finally, we note that BitVault's smart brick is underutilized in terms of their CPU power. We are evaluating the option of introducing some preliminary searching and index building functionality into the BitVault layer.

6. Evaluation

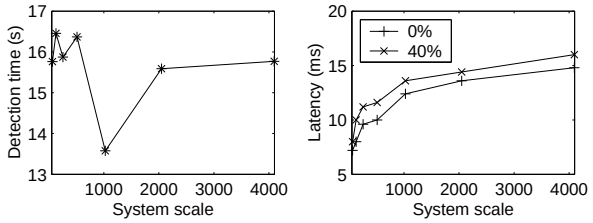
This section provides detailed evaluations of all major aspects of BitVault. We build a prototype of 30 bricks, each of which is a commodity PC. These PCs run Windows XP, and their hardware configurations are 3GHZ Pentium4 CPU, 512MB memory and 120GB STAT Seagate disk. These PCs are connected with two AT-8324SX 100Mb switches stacked together. Unless otherwise specified, k=3 in all experiments.

Except the one on MRL performance, all results are obtained through the direct measurement of the 30-brick prototype.

6.1 Performance of MRL

We use simulation to study the performance of MRL. In this experiment, we select a node to crash from a stabilized system. As we mentioned earlier, the membership protocol works in two phases. In the failure detection phase, the leafset nodes vote out a dead neighbor. In the failure dissemination phase, the takeover node starts a broadcast through its fingers and leafset nodes. The broadcast is best-effort and the anti-entropy protocol ensures eventual convergence.

XRing's leafset heartbeat, finger probing and anti-entropy use interval of 5s, 5s and 10s respectively. The leafset size is 8 (i.e. 4 logical neighbors on each side). A brick marks a neighbor as dead after failing to hear from it in 3 heartbeat cycles. The vote among the leafset members will declare a brick's departure in 10~20 seconds. To understand the MRL's robustness, we drop r% of packets. The dropping applies uniformly to all types of MRL messages.



(a) Failure detection time (b) Membership convergence time

Figure 8. Failure detection time (a) and convergence time of MRL (b)

Figure 8 shows the failure detection and convergence speed, for different system scale and we use two drop rate, 0% and 40%. The failure detection time is around 16 seconds, irrespective of system size. This is because that the detection is done through the leafset nodes. This value is consistent with what we observed in prototype. We set the network latency to be 2ms, and thus convergence speed is very fast and rises with O(log$N$) in general. Higher drop rate yields longer converging time, but the difference is negligible in practice.

6.2 Check-in and check-out

Table 1. Latency of Checkin/out request

Size	Client Request Latency(ms)					
	Remote NTFS		1-brick		30-brick	
	CI	CO	CI	CO	CI	CO
10K	5	4	11	4	17	7
100K	17	14	19	12	34	18
1M	147	118	105	99	220	105
10M	1437	1155	1003	995	2126	1009

Our first study compares the raw checkin and checkout performance in the 30-brick prototype with different object sizes. The requests are issued synchronously, and the results are the averages of 10 runs. We also compare against a 1-brick and the native read/write performance of a remote mounted NTFS directory (k=1 for these two configurations). The result is summarized in Table 1. The 1-brick data is comparable with the remote NTFS. The 30-brick case adds more network trips, but the performance is still competitive.

Next we study scalability. In this test, there are 14 clients. Each client executes a loop to fire synchronous requests to a prototype system with varying numbers of bricks (from 2 to 16). Objects requested do not overlap across clients, and object IDs are random. Checkin and checkout are measured separately. Figure 9 shows the total throughputs in MBytes-per-second against number of bricks. The results are average values over 20 runs and the object placement policy is random.

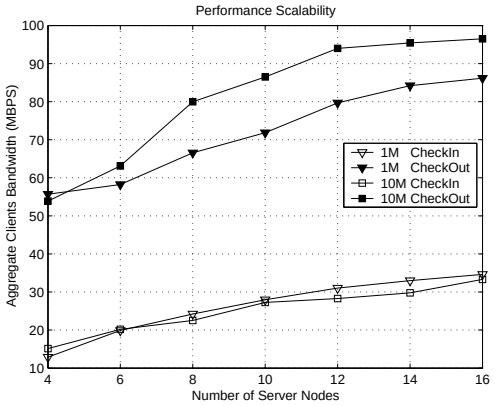


Figure 9: checkin and checkout throughput for 1MB and 10MB objects.

The checkout performance is 3~4 times better than checkin, simply because for each checkin there are 3 times more requests going to the disks. Checkout of 10MB objects is better than 1MB due to sequential access to disks. After brick number increases to 14, the curves become flat because the client requests can no longer overload the bricks. Because the clients fire requests synchronously and that the requests are randomly scheduled to bricks, brick loads are not completely even and thus the scalability curve is sub-linear. We can not fully get rid of the caching effect of the hosting file system, and this is the reason that checkin performance of 1MB is close to that of 10MB. The maximum throughputs of checkin and checkout of 10MB object is 97MB/s and 35MB/s, respectively. These numbers are comparable with the GFS[14] data on a similar testbed configuration.

6.3 Repair performance

Table 2 takes a closer look at what happens inside the system under repair. We let each brick log the number of objects and indices it hosts periodically, and merge them at the end after aligning the clocks. In this experiment, every brick has 30K 1MB objects (30GB/brick), and we fail several bricks in sequence. We vary the total number of initial bricks at a step of 5 bricks.

Table 2. Repair speed experiment. The experiment is done by initially setting 5*i* healthy bricks with each one filled by 30G replicated data. Then manually fail one brick and measure the time to repair 90% of the lost 30G data.

Brick Number (after crashed)	Time to repair 90% (min)	Repair Bandwidth	
		MB/s	GB/m
4	56.4	8.2,	0.5
9	20.3	22.8,	1.4
14	9.9	46.5,	2.8
19	9.3	49.4,	3.0
24	5.3	86.4,	5.2
29	5.3	87.8,	5.3

The total duration to repair 30GB data with 20 and 30 bricks takes 600 and 300 seconds each, giving a repair rate of 50MB/s and 100MB/s, respectively. The super-linear improvement is probably due to better utilization of memory and other per-brick resources. The rate of 20 bricks is about 1/8 of what a 227-node GFS cluster achieves [14]. BitVault's repair performance shall improve nicely with number of bricks (up to a ceiling imposed by the network bandwidth), and we are confident that it is comparable with the GFS performance.

6.4 Performance under failure

BitVault should self-heal and continue to function even in the face of failure. To verify this, we conducted checkin from 16 clients into a 16-brick BitVault, and then failed one brick. Each client continuously checks in 1MB size objects. We gather statistics in units of 5-second granularity. For the client-side throughput, we log the aggregate throughput in terms of total successful checkins. Similarly, we log the total number of objects that the bricks receive, again aggregated over all bricks. At the 10^{th} minute, we failed one brick. The client-side throughput corresponds to what users perceive, while the server-side throughput reflects both the checkin traffics as well as the repair traffics. The expected behavior is that the performance will drop while repair is going on, and then return to the normal level afterwards.

Figure 10 shows the variation of the throughput of both clients and servers, and the server-side throughput is normalized by 3 (the replication degree). Before the crash and after the repair, the client-side throughput matches with the server-side throughput. However, after the crash and during the repair window, the client-side throughput decreases because resources are dedicated to repair the failed disk. The repair traffic, represented by the exceeding dark area, corresponds to about 3GB worth of data on the failed disk. The repair window is about 70 seconds. If there were more data on the failed disk, the repair window would increase. During repair, whether repair traffic takes higher priority than user requests is a policy issue. In this prototype, they compete against each other with the same priority (only MRL messages have higher priority).

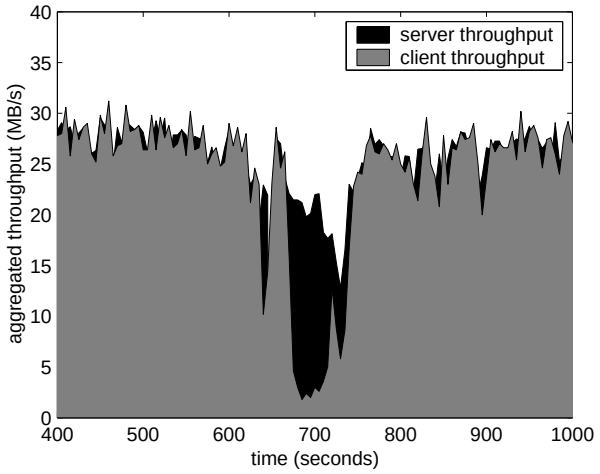


Figure 10: performance under failure experiment. Server throughput is normalized by 3. A failure is introduced around the 10th minute.

7. Related work

As stated in [25], the primary challenges for systems like BitVault will not be performance but, instead, management and availability. The primary contributions of BitVault are: 1) use eventual membership protocol with a DHT abstraction to offer great scale-out capability with low overhead and in a self-managing fashion, and 2) employ massively parallel repair to achieve very high data reliability and availability, and 3) deliver both with a simple architecture. Below we will contrast its core contributions with previous systems.

Single-box solutions such as Venti[26] cannot meet the challenge of coping with the volume and growth rate of reference data; client/server architecture such as GFS[14], NASD[14] and WiND[4] works to certain extent but will hit bottleneck as well. The fact that objects are often small and there is no or little access locality exacerbates the scaling problem further. Existing fully distributed proposals such as Boxwood[23], FAB[11], Petal[20] and xFS[3] all require strong consensus protocol which, even when is not placed on the critical path, presents a scalability challenge.

The DHT-based systems such as Oceanstore [19], Pond [28], CFS [10], Ivy [24], PAST [30] and Pastiche [9] have gone to the other extreme. They operate over a logical space with a hash table abstraction, maintain a small list of other members (O(log*N*) and traverse the space in O(log*N*) steps, often require replicas to be placed on a fixed set of nodes starting from the one that hosts the hash of the object. They primarily target at wide-area P2P sharing scenario, and are thus self-organizing and can scale out. However, their target context is dynamic and has led to legitimate concerns on what guarantee these systems can provide [7]. In the course of designing BitVault, we have found that, unfortunately, these designs do not fit the more benign environment either. The restriction of placing replicas sequentially impacts the ability of handling heterogeneity for better storage utilization, causing data movement not for the sake of repair but to satisfy the placement invariants. Coupling object placement with the logical space does not support self-identifying disks, and does not leverage abundant network bandwidth to achieve rapid and parallel repair. These are the issues that can only be solved by using indices to control the placement.

The problem of using indirection is that it introduces the indices as yet another vulnerability point. The conventional methodology, adopted by many including GFS[14] and TotalRecall[5], has been to first ensure the integrity of index, which then reacts to failures via failure detection. BitVault demonstrated that, if coupled with an eventual membership service, the object-driven repair strategy, one in which the survival of the last replica can quickly restore both the index and the rest of replicas, is both simple and effective. This architecture also affords very rapid repair by spreading repair loads, which has so far only been done in a centralized-indexed system such as GFS [14].

EMC Centera [12] is a brick-based retention platform that aims at self-healing and manageability. However, no architectural details are available, and its scalability target is not clear.

BitVault as a scalable store for immutable object is only a starting point. For example, it is conceivable to use Farsite[2]'s directory service for the namespace while storing objects inside BitVault. Similarly, if we combine an in-system P2P locking protocol [22], it is possible to build a file system, perhaps in the same style as Frangipani [33].

8. Conclusion and future work

The main objectives of a large-scale distributed storage system are its maintainability and availability. P2P technologies – currently widely explored for wide-area context, are immensely interesting design alternatives in keeping the management cost down. When large amount of components are brought together, they also bring the possibility of doing massively parallel repair for high data availability. BitVault has demonstrated both of the above points.

Our future work will focus on developing BitVault applications in order to understand whether new functionalities are necessary inside BitVault. This includes the client utility that backs up user selected files/directories in their private namespace, and also an initiative to build search and query layer on top of BitVault.

References

[1] "Enterprise Storage Group Reference Information: The Next Wave.", June 2002.

[2] A. Adya, W.J. Bolosky, M. Castro, et al, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment", OSDI'02.

[3] T.E. Anderson, M.D. Dahlin, J.M. Neefe, et al. "Serverless Network File Systems", SOSP'95.

[4] A. Arpaci-Dusseau, R. Arpaci-Dusseau, et al, "Manageable Storage via Adaptation in WiND", CCGrid'01.

[5] R. Bhagwan, K. Tati, Y.C. Cheng et al, "Total Recall: System Support for Automated Availability Management", NSDI'04.

[6] K. Birman and R. van Renesse, "Reliable Distributed Computing with ISIS Toolkit", IEEE Computing Society Press, 1994.

[7] C. Black, R. Rodrigues, "High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two", HOTOS'03.

[8] G. V. Chockler, I. Keidar, and R. Vitenburg, "Group communication specifications: A comprehensive study", ACM Computing Surveys, 88:4, 2001, 427—469.

[9] L.P. Cox, C.D. Murray, B.D. Noble, "Pastiche: Making Backup Cheap and Easy", OSDI'02.

[10] F. Dabek, M.F. Kaashoek, D. Karger, et al, "Wide-area cooperative storage with CFS", SOSP'01.

[11] A. Das, I. Gupta, A. Motivala, "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol", DSN'02

[12] EMC-Centara: http://www.emc.com/products/systems/centera.jsp

[13] S. Frolund, A. Merchant, Y. Saito, et al, "FAB: enterprise storage systems on a shoestring", HOTOS'03.

[14] S. Ghemawat, H. Gobioff, S.T. Leung, "The Google File System", SOSP'03.

[15] G.A. Gibson, D.F. Nagle, K. Amiri, et al. "A Cost-Effective, High-Bandwidth Storage Architecture", ASPLOS'98.

[16] J. Gray, W. Chong, T. Barclay, et al. "TeraScale SneakerNet: Using Inexpensive Disks for Backup, Archiving, and Data Exchange", MSR Technical Report No. MSR-TR-2002-54.

[17] J. Gray, "Storage Bricks Have Arrived," invited talk FAST'02.

[18] D. Kostić, A. Rodriguez, J. Albrecht, et al, "Using Random Subsets to Build Scalable Network Services", USITS'03.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, et al, "OceanStore: An Architecture for Global-Scale Persistent Storage", ASPLOS'00.

[20] Q. Lian, W. Chen, Z. Zhang, "On the Impact of Replica Placement to the Reliability of Distributed Brick Storage Systems", submitted to ICDCS'05.

[21] E.K. Lee, C.A. Thekkath, "Petal: Distributed Virtual Disks", ASPLOS'96.

[22] S.D. Lin, Q. Lian, M. Chen et al, "A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems", IPTPS'04.

[23] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, Boxwood: Abstractions as the Foundation for Storage Infrastructure, ODSI'04

[24] A. Muthitacharoen, R. Morris, T. M. Gil, et al, "Ivy: A Read/Write Peer-to-peer File System", OSDI'02.

[25] D. Patterson, A. Brown, P. Broadwell, et al, "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies", UCB Technical Report No. UCB/CSD-02-1125.

[26] S. Quinlan, S. Dorward, "Venti: a new approach to archival storage", FAST'02.

[27] S. Ratnasamy, P. Francis, M. Handley, et al, "A Scalable Content-Addressable Network", SIGCOMM'01.

[28] S. Rhea, P. Eaton, D. Geels, et al, "Pond: the OceanStore Prototype". FAST '03

[29] A. Rowstron, P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems", IFIP/ACM Middleware'01.

[30] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", SOSP'01.

[31] I. Stoica, R. Morris, D. Karger, et al, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", SIGCOMM'01.

[32] N. Talagala, S. Asami, D. Patterson, et al, "Tertiary Disk: Large Scale Distributed Storage", UCB Technical Report No. UCB//CSD-98-989.

[33] C.A. Thekkath, T. Mann, E.K. Lee, "Frangipani: A Scalable Distributed File System", SOSP'97.

[34] Z. Zhang, Q. Lian, Y. Chen, "XRing a Robust and High-Performance P2P DHT", Microsoft Research Technical Report No.MSR-TR-2004-93.

[35] Z. Zhang, S.D. Lin, Q. Lian, C. Jin, "RepStore: A Self-Managing and Self-Tuning Storage Backend with Smart Bricks", ICAC'04

[36] Z. Zhang, S.M. Shi, J. Zhu, "SOMO: Self-Organized Metadata Overlay for Resource Management", IPTPS'03.

[37] B.Y. Zhao, J. Kubiatowicz, A.D. Josep, "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing", UCB Technical Report No. UCB/CSD-01-1141.