An Analytical Framework and Its Applications for Studying Brick Storage Reliability

Ming Chen[+], Wei Chen[+], Likun Liu[*], and Zheng Zhang[+]
Microsoft Research Asia[+]
Tsinghua University[*]
{mingche[+], weic[+], zzhang[+]}@microsoft.com, llk04[*]@mails.tsinghua.edu.cn

Abstract

The reliability of a large-scale storage system is influenced by a complex set of inter-dependent factors. This paper presents a comprehensive and extensible analytical framework that offers quantitative answers to many design tradeoffs. We apply the framework to a number of important design strategies that a designer and/or administrator must face in reality, including topology-aware replica placement, proactive replication that uses small background network bandwidth and unused disk space to create additional copies. We also quantify the impact of slow (but potentially more accurate) failure detection and lazy replacement of failed disks. We use detailed simulation to verify and refine our analytical model. These results demonstrate the versatility of the framework and serve as a solid step towards more quantitative studies of fundamental system tradeoffs between reliability, performance, and cost in large-scale distributed storage systems.

1. Introduction

Storage solution using clustered "smart bricks" connected with local area network is becoming an increasingly attractive alternative to the more expensive storage-area network (SAN) solution. Some of the exemplary systems include NASD [10], GFS [9], FAB [19], Repstore [24], and Boxwood [13]. A smart brick is essentially a stripped down PC with a CPU, memory, network card, and a large disk. The smart-brick solution is cost-effective and can be scaled up to thousands of bricks. Another important trend in storage systems is the increasing demand for storing reference data, data that are rarely changed but need to be stored for a long period of time [1]. As more and more information being digitized, the storage demand for documents, images, audios, videos and other reference data will soon become the dominant storage requirement for enterprises and large internet services [1]. Large scale brick storage fits the requirement for reference data quite well, because it is low-cost, simple, and has good scalability and availability. However, for brick storage systems, providing strong data reliability (i.e., no data loss) as required by reference data storage is confronted with new challenges, because inexpensive commodity disks are prone to permanent failures and failures are far more frequent in large systems. In this paper, we study the reliability of brick storage that stores reference data, or immutable data.

To guard against permanent loss of data, replication is often employed. If some but not all replicas of an object are lost due to brick failures, new replicas can be regenerated before further failures to maintain the same level of reliability. New bricks may also be added to replace failed bricks and data may be migrated from old bricks to new bricks to keep global balance among bricks. We call the process of regenerating lost replicas after brick failures *data repair*, and the process of migrating data to the new replacement bricks *data rebalance*. These two processes are the primary maintenance operations involved in a production system.

The reliability of the system is influenced by many parameters and policies embedded in the above two processes. What complicates the analysis is the fact that those factors can have mutual dependencies. For instance, cheaper disks (e.g. SATA vs. SCSI) are less reliable but they give more headroom of using more replicas. Larger replication degree in turn demands more switch bandwidth. Yet, a more carefully designed replication strategy could avoid the burst traffic by proactively creating replicas in the background. Efficient bandwidth utilization depends on both the given (i.e. switch hierarchy) and the design (i.e. placement strategy). Object size also turns out to be a non-trivial parameter as well. Moreover, faster failure detection and faster replacement of failed bricks can

provide better data reliability, but they incur increased system cost and operation cost. While it is easy to see how all these factors qualitatively impact the data reliability, it is important for system designers and administrators to understand the *quantitative* impact, so that they are able to adjust the system parameters and design strategies to balance the tradeoffs between cost, performance, and reliability.

This work makes two contributions. First, as a significant extension to an earlier work [12], it provides an analytical framework based on Markov model to study data reliability of brick storage systems where random replica placement (as in [9][12][18]) is used. The basic model, described in Section 2, covers data repair and rebalance and considers some fundamental factors such as object size and bounded system bandwidth, while further extensions of the model cover more advanced factors such as proactive replication, failure detection delay and brick replacement delay. While in most cases we cannot derive a closed form solution, the framework provides a computation procedure and it is "scalable" in the sense that it can predict reliability at a system scale not easily affordable by detailed simulation.

Next, given this analytical framework, we show several applications of the framework to study some of the most important issues when building a working system.

First, in Section 3, we study the impact of switch topology. Existing models typically assume a flat, one-level hierarchy. In reality, a scalable storage system is invariably connected in a tree-like topology and thus replica placement strategies need to be topology-aware. We demonstrate that network topology has a large impact on system reliability. We propose a policy where objects are randomly placed among all bricks while data repair are only carried out within the same local switch. The more efficient use of bandwidth results in significant improvement of the system reliability, in some cases more than two orders of magnitude.

Second, in Section 4, we study the impact of proactive replication, which uses a limited bandwidth budget to generate additional replicas in the background even without brick failures [21]. With the increasing capacity of disks, trading unused storage space for better reliability becomes attractive. Nevertheless, there is a lack of quantitative understanding of the effectiveness. Our result shows that in a 3-way replication system, even with 1% of total replication maintenance bandwidth to proactively generate one more replica for each object, we can dramatically improve reliability. Generating two extra replicas with 0.5% bandwidth budget, the reliability exceeds the reliability of a 4-way replication system.

Third, in Section 5, we study the impact of delaying failure detection and delaying the replacement of failed bricks. In reality, failure detection is never accomplished instantly. A longer detection time requires less detection messages and thus saves system bandwidth. More importantly, it tolerates more transient failures and avoids incurring the heavy data repair and rebalance cost prematurely. We show how our model can be extended to study the sensitivity to failure detection. In our system settings, detecting brick failures can be delayed to a couple of minutes without significantly reducing data reliability. For brick replacement, it is impractical to assume an infinite supply of backup bricks. The common practice is to replace failed bricks periodically. This policy is driven by the need to reduce human maintenance cost. We show that delaying replacement does not have a strong impact to data reliability. In the setting that we studied, brick replacement can be delayed for days or weeks.

We offer detailed simulation results in Section 6 and related work in Section 7. We conclude with future work in Section 8.

2. Analytical framework

We analyze the brick storage reliability in terms of the *mean time to data loss* of the system, denoted as $MTTDL_{sys}$. That is, after the system is loaded with desired number of replicas of the objects, the expected time when the first data object is lost by the system.

The analysis has two major steps. In the first step, we fix one arbitrary object, and analyze the mean time to data loss of this particular object, denoted as $MTTDL_{obj}$. In the second step, we estimate the number of independent objects (in terms of data loss behavior), and denote it as π. Then the mean time to data loss of the system is given as: $MTTDL_{sys}=MTTDL_{obj}/\pi$.

2.1. Markov model for $MTTDL_{obj}$

To analyze $MTTDL_{obj}$, we use a discrete-state continuous-time Markov process as depicted in Figure 1(b) to model the dynamics of the system. The Markov process is illustrative and simple enough, so we do not use more advanced stochastic modeling tools such as SPNP [2] to avoid distracting readers with the details of the tools.

A state in the Markov process is defined by (n, k), where n is the number of online bricks, and k is the current number of replicas of the observed object among the online bricks. A brick is online if it is functional and it achieves the balanced load (stores an average amount of data). Initially the system is in state

(*N, K*), where *N* is the total number of bricks and *K* is the replication degree, the desired number of replicas for the observed object. The model has one absorbing state, *stop*, which is the state when all replicas of the object are lost before any repair is successful. Data loss occurs when the system transitions into the *stop* state. $MTTDL_{obj}$ is computed as the mean time from the initial state (*N,K*) to the *stop* state.

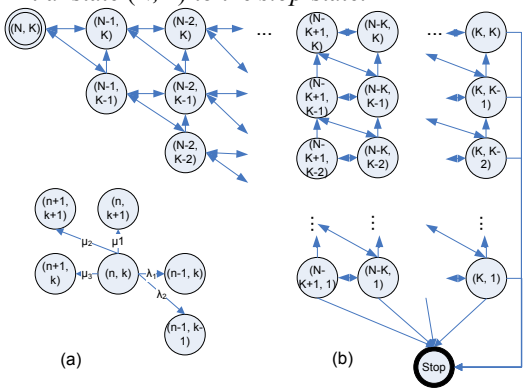

Figure 1. Model 0: Markov process of system replicas maintenance process (states with less than *K* online bricks are omitted for a large system with online bricks less than *K* can be approximately considered broken)

To compute this value, we need to provide transition rates between all states. Our computation always refers to the replicas of the observed object. We assume that each individual brick has an independent failure rate of λ. There are five transitions leaving the state (*n,k*) (Figure 1(a)). λ_1 is the rate of the transition moving to (*n*-1,*k*), the case where a new brick fails but it does not contain a replica. Since there are (*n*-*k*) such bricks, $\lambda_1 = (n\text{-}k)\lambda$. λ_2 is the rate of the transition moving to (*n*-1,*k*-1), in which case the failed brick contains a replica and thus $\lambda_2 = k\lambda$.

Transition rates μ_1, μ_2, and μ_3 are the rates for repair and rebalance transitions[1]. When the system is in state (*n,k*), data repair is the process to regenerate all lost replicas in the failed *N*-*n* bricks among the remaining *n* bricks, and it should do so as fast as possible. If all *n* bricks participate in the repair process, data repair can be done in parallel and can be very fast. In the mean time, data rebalance is carried out to regenerate all lost replicas on the *N*-*n* new bricks that are installed to replace the failed bricks. Data rebalance is the process to replace failed bricks with new bricks, and fill the new bricks with the replicas lost by the failed bricks until each of the new bricks stores an average amount of data, so that they are then brought online for service. When both data repair and data rebalance complete, the replicas lost on the failed bricks have all been recovered on the new replacement bricks, so the replicas generated on the surviving bricks in the repair process will be deleted to keep the desired replication degree *K*. We assume that the repair and rebalance processes are reliable in that no replicas are lost during data repair or rebalance.

In the basic model we assume that brick failures are detected instantaneously and new bricks are installed immediately to replace failed bricks. In Section 5 we extend our model to consider failure detection delays and brick replacement delays.

Transition rate μ_1 is the rate of data repair from state (*n,k*) to (*n,k*+1). In state (*n,k*), the data repair process regenerates (*K*-*k*) replicas among the remaining *n* bricks in parallel. Let *i*=1,2,…*K*-*k* be the (*K*-*k*) bricks receiving these replicas. For each brick *i*, let $d_{r,i}$ be the amount of data it receives for data repair, and $b_{r,i}$ be the bandwidth it is allocated for repair. Then $b_{r,i}/d_{r,i}$ is the rate brick *i* completes the repair of a replica. Since all (*K*-*k*) replica repairs are in parallel, the overall repair rate $\mu_1 = \sum_{i=1}^{K-k} b_{r,i} / d_{r,i}$.

Transition rates μ_2 and μ_3 are for rebalance transitions filling the *N*-*n* new disks. In particular, μ_2 is the rate of completing the rebalance of the first new brick that contains a new replica (to state (*n*+1,*k*+1)), while μ_3 is the rate of completing the rebalance of the first new brick not containing a replica (to state (*n*+1,*k*)). For each of the *N*-*n* new bricks, let d_l be the amount of data to be loaded, and b_l the available bandwidth for copying data. Thus the rate for each new brick to complete rebalance is b_l/d_l. Therefore $\mu_2=(K\text{-}k)*b_l/d_l$ and $\mu_3 = ((N\text{-}n)\text{-}(K\text{-}k))*b_l/d_l$, since (*K*-*k*) new bricks contain replicas of the object and (*N*-*n*)-(*K*-*k*) bricks do not.

The values of $d_{r,i}$, $b_{r,i}$, d_l, b_l depend on placement and repair strategies as well as system configuration such as backbone bandwidth, brick bandwidth, etc. and they are determined in Section 2.2 for random placement and repair strategy.

When all transition rates are known, $MTTDL_{obj}$ is computed with the following procedure [4]. We number all the states except the *stop* state to be state 1, 2, 3, …, with state 1 being the initial state (*N*, *K*). Let $Q^* = (q_{i,j})$ be the transition matrix, where $q_{i,j}$ is the transition rate from state *i* to state *j*. We then calculate matrix $M = (I\text{-}Q^*)^{-1}$. Finally, we have $MTTDL_{obj} = \Sigma_i m_{1,i}$, where $m_{1,i}$ is the element of *M* at the first row and the *i*-th column. It is also possible to calculate $MTTDL_{obj}$ using a system of linear equations as proposed by Muppala et.al [14]. When the number of states is not

[1] Transition times are approximately considered to follow exponential distribution for network traffic fluctuates.

very large, the calculation of matrix inversion is simple and feasible.

We now briefly justify why we choose to use the model in Figure 1. At each state, data repair for the particular object is affected by the available system bandwidth and the amount of data to be repaired. These quantities are determined by the total number of bricks remaining in the system. So we need parameter n in the state. We also need parameter k in the state to denote how many copies of the particular object have left and when the object is lost. Explicit use of replica number k in the state is also useful when we extend the model to consider other replication strategies, such as proactive replication in Section 4.

2.2. Parameters for random placement and repair strategy

In this paper, we consider random placement and repair strategy, which appeared in a number of studies ([9][12][18]). With this strategy, all replicas of any given object are randomly placed among all bricks in the system. When a replica is lost, a new replica is randomly generated among all remaining bricks in the data repair process.

Table 1. Parameters

Parameter	Explanation	Default
N	Number of total bricks	1024
$\lambda = 1/MTTF$	Death rate of a brick	1/3 (1/year)
K	Replication degree, i.e., Desired number of replicas per object	3
D	Total amount of unique user data	1PB
s	Object size	4MB
B	Switch bandwidth for replica maintenance	3GB/s
b	Brick IO bandwidth	20MB/s
p	Fraction of B and b allocated for repair; (1-p) for rebalance	90%
F	Total number of objects in the system	D/s
x	(approximate) number of failed bricks whose data still need to be repaired	1
A	Total number of remaining bricks that can participate in data repair and data rebalance and serve as the data source	$\min(n, FKx/(n+x))$

Table 1 shows the system parameters and their default values used in our calculation of all the figures. The default values are based on an exemplary petabyte data storage that could be built in a few years, for example B=3GB/s is 10% of bi-sectional bandwidth of a 10Gbps 48 port switch, b=20MB/s is the mixed sequential and random disk access bandwidth, and λ= 1/3yr corresponds to a cheap brick (disk) with 3 year mean time to failure. Several reports ([20][11]) provide lower failure rates about disk failures, but they are based on aggregate failure rates of a number of disks during their initial use period (the first year), and it is well known that failure rate increases as a disk ages, so we choose to use a higher failure rate to be conservative, and it also matches typical disk warranty length. Between data repair and rebalance, we allocate most bandwidth (p=90%) for data repair, since we want the lost replicas to be regenerated as fast as possible to support high data reliability. We now further explain parameters x and A in the table.

When the system is in state $S = (n, k)$, part of the data on the N-n failed bricks have been repaired before the system transitions into the state. However, we do not have direct information from the state to derive the exact amount of data still need to be repaired. If we add extra parameters to the state to record this information, the state space will be too large and make our computation infeasible. Therefore, we use an approximation parameter x in the calculation of $d_{r,i}$. Parameter x denotes the (approximate) number of failed bricks whose data still need to be repaired, and it takes values from 1 to N-n. In other words, when the system is in state S with n online bricks, we assume that in a previous state S' with $n+x$ online bricks, the system has (almost) done its data repair, and only the data in the last x failed bricks need to be repaired in state S.

When $x = N$-n, the approximation is the most conservative, and it ignores data repaired in all previous states. Without any further information, one can use x=N-n to make a conservative estimate of $MTTDL_{sys}$. In general, the value of x is determined by the failure rate of the bricks and the repair speed: the lower the failure rate and the higher the repair speed, the smaller the value of x. For our setting, we use our simulation results (shown in Section 6) to tune the parameter, and we find that x=1 is sufficient for our setting, which means the data needed to be repaired are mostly the data in the last failed brick and other data are mostly repaired already. This is reasonable given our low failure rate (about 1 failure per day) and relatively high repair speed (a few tens of minutes to repair one failed brick in parallel). Henceforth, we use x=1 in all the calculations of our analytical results.

Table 2. Formulas for the key quantities in random placement

$b_{r,i}$	$\min(Bp/A, bp)$, same for all i	Bp/A is the root switch bandwidth allocated for repair for one online brick

		participated in repair; *bp* is the brick IO bandwidth allocated for repair.
$d_{r,i}$	$\frac{D\times K\times x}{(n+x)\times A}$, same for all i	Parameter x is the number of failed bricks whose data still need to be repaired. $(D*K)/(n+x)$ is the amount of data on one brick. With x failed bricks to repair, their data are evenly distributed among the A remaining bricks as repair source.
b_l	$\min(b(1-p)\times A/(N-n), B(1-p)/(N-n), b)$	$b(1\text{-}p)*A$ is the total bandwidth with which A online bricks can contribute for rebalance, and it is evenly distributed for $(N\text{-}n)$ new bricks; $B(1\text{-}p)$ is the root switch bandwidth allocated for rebalance, and it is also allocated evenly for $(N\text{-}n)$ new bricks; b is the brick IO bandwidth one new brick can use for rebalance.
d_l	$\frac{D\times K}{N}$	Average amount of data one brick should maintain.

Quantity A denotes the total number of remaining bricks that can participate in data repair and data rebalance and serve as the data source, and it is calculated as follows. Let F be the total number of objects stored in the system, then $F = D/s$, where s is the average size of object. In state S with n online bricks, the total number of lost replicas is given by $FKx/(n+x)$, since by our assumption in a previous state S' with $n+x$ online bricks all data are repaired so each brick has $FK/(n+x)$ replicas and from state S' to S all data on the last x failed bricks are lost and need repair. Then we have $A = \min(n, FKx/(n+x))$. This is because, when $FKx/(n+x) > n$, all lost replicas can be equally distributed among n remaining bricks as data source for repair and rebalance; when $FKx/(n+x) < n$, at most $FKx/(n+x)$ bricks can serve as data source for lost replicas.

Table 2 shows the formulas for $d_{r,i}$, $b_{r,i}$, d_l, and b_l and their explanations. We provide extra explanation for $d_{r,i}$ below. Since in state (n, k) the data on the last x failed bricks need repair, and each brick contains $DK/(n+x)$ amount of data, so the total amount of data to repair is $DKx/(n+x)$. These data are eventually distributed among A participating repair sources, so each brick has $DKx/[(n+x)A]$ amount of data to repair.

2.3. Estimate π, the number of independent objects for random placement and repair

It is difficult to estimate the number of independent objects, since replicas of different objects may be partly co-located in same nodes and thus these overlapping objects are not completely independent.

To solve this problem, we consider an ideal model in which we can calculate the exact quantity π, and use it as our estimate for π in our model. In the ideal model, time is divided into discrete slots, each of which with length Δ. Within each slot, each machine has an independent probability P to fail. At the end of each slot, data repair and data rebalance are completed instantaneously. In this model, we can derive the exact formulas for $MTTDL_{obj}$ and $MTTDL_{sys}$, and thus we can obtain $MTTDL_{obj}$ / $MTTDL_{sys}$. We then let Δ tend to zero (so P tends to zero), and we use the quantity

$$\lim_{P\to 0}\frac{MTTDL_{obj}}{MTTDL_{sys}} = C_N^K(1-(1-1/C_N^K)^F)$$

$$\approx C_N^K(1-e^{-F/C_N^K})$$

as our estimate of π. In real systems data repair and rebalance can usually be done in a much smaller time scale (hours) comparing with the life time of a brick (years). Thus assuming instant data repair and rebalance in the ideal model would give a close estimate of π.

Some typical values of the above approximation are as follows. When $F << C_N^K$, it is F; when $F >> C_N^K$, it is C_N^K. In other words, if there are too few objects, then their failures can be regarded as independent; and if there are many objects, then any combination of K bricks can be considered as one independent pattern. Also, when $F = C_N^K$, it becomes $C_N^K(1-e^{-1})$.

2.4. *Sample results*

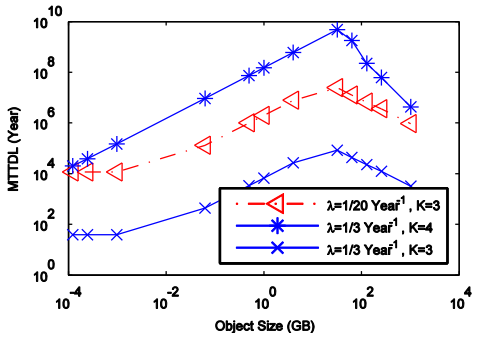

Figure 2. *MTTDL* in random placement

Figure 2 shows the reliability of the system with respect to the size of the objects in the system. The result shows that data reliability is low when the object size is small, because the huge number of randomly placed objects uses up all replica placement combinations C_N^K, and any K concurrent brick failures will wipe out some objects. On the other hand, when the object size is too large, the reliability also decreases because there are not enough parallel repair degree to speed up data repair. Therefore, increasing parallel repair bandwidth and decreasing the number of independent objects are two important ways to improve data reliability. Moreover, there is an optimal object size for system reliability, where the number of independent objects π is reduced to a point when the system bandwidth is just about fully utilized for parallel repair process. This result is consistent with the results in [12]. The figure also indicates that a 4-way replication system with low reliability bricks (average brick life time is 3 years) can achieve much better reliability than a 3-way replication system with high reliability bricks (average brick life time is 20 years). This shows that highly reliable bricks can be traded with lowly reliable (and thus cheaper) bricks with extra disk capacity, and increasing individual brick reliability is less effective than increasing replication degrees of data objects.

In the following sections, we will apply our analytical framework to analyze a number of issues that are related to data reliability in distributed brick storage systems.

3. Topology-aware placement and repair

As indicated in the previous section, increasing available parallel repair bandwidth and reducing the number of independent objects are important for improving data reliability. In this section we apply our analytical framework to analyze and compare different placement and repair strategies that utilize network switch topology.

We assume a typical switch topology with multiple levels of switches forming a tree topology. We refer to the set of bricks attached to the same leaf level switch as a cluster. Traffic within a cluster only traverses through the leaf switch, while traffic between the clusters has to traverse through parent switches. Given the tree topology, we have three different replica placement and repair strategies, based on the choices of *initial placement* (where to put object replicas initially) and *repair placement* (where to put new object replicas during data repair):

1. Both initial and repair placement are fully random across the whole system, in which case potential repair bandwidth is bounded by the root switch bandwidth. We refer to this as global placement with global repair (GPGR).
2. Both initial and repair placement are random within each cluster. Essentially each cluster acts as a complete system and data are partitioned among clusters. In this case, potential parallel repair bandwidth is bounded by the aggregate bandwidth of those leaf switches under which there are failed bricks. We refer to this as local placement with local repair (LPLR).
3. Initial placement is random across the whole system, but repair replacement is within the same clusters as the repair source. This approach significantly improves data repair bandwidth, since it could aggregate the bandwidth of *all* leaf switches for repair. Data rebalance still consumes root switch bandwidth. We refer to this as global placement with local repair (GPLR).

We consider all switches having the same bandwidth B as given in Table 1. GPGR calculation is already given in Table 2. For LPLR, each cluster can be considered as an independent system to compute its $MTTDL_c$, and then the $MTTDL_{sys}$ is $MTTDL_c$ divided by the number of clusters. GPLR has the same $d_{r,i}$, d_l, b_l, and π as the GPGR method, but it has a different (and larger) repair bandwidth $b_{r,i} = \min(Bp/(A/(N/c)), bp)$, where c is the cluster size. The formula means that for the A bricks that may participate in data repair, they are evenly distributed among N/c clusters, and each cluster can provide Bp switch bandwidth for repair. Therefore, each repair destination can obtain $Bp/(A/(N/c))$ bandwidth from its leaf switch for data repair.

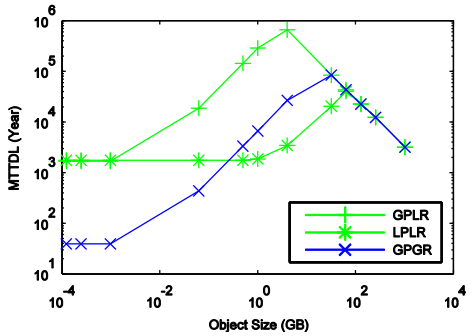

Figure 3. Reliability with different placement and repair strategies that utilize switch topology. Cluster size c=48.

Figure 3 shows the reliability of the three different placement and repair strategies. First, GPLR is several orders of magnitude better than GPGR in most cases, because they have the same number of independent objects while GPLR can aggregate a much larger

bandwidth for data repair. Only when the object size is very large, in which case there is not enough parallelism in repair and repair is bounded by brick bandwidth, does the two strategies have the same reliability.

Second, comparing GPGR with LPLR, GPGR has much worse reliability when the object size is small, because its placement is not restricted and it has a much larger number of independent objects. When the object size is large, GPGR has better reliability, because in this range there is still enough repair parallelism such that GPGR can fully utilize the root switch bandwidth, but in LPLR repair is limited within a cluster of size 48, and thus cannot fully utilize the leaf switch bandwidth for parallel repair.

Third, comparing GPLR with LPLR, GPLR is usually better than LPLR unless the object size gets very small or very large. This means that the aggregated bandwidth in GPLR plays a significant role in speeding up parallel repair, until the number of independent objects gets too large or the parallel repair degree gets too low such that the gain of aggregated bandwidth in repair is cancelled out.

With this analysis, we reach an important conclusion concerning the utilization of switch topology: If the system can choose the object size appropriately (perhaps by grouping small objects together), randomly placing replicas uniformly among all bricks while carrying out parallel repair locally within the same switch provides by far the best data reliability.

4. Proactive replication

Proactive replication [21] exploits free storage space and volatile network bandwidth to improve reliability by continuously generating additional replicas besides the desired number K in the constraint of fixed allocated bandwidth. When using proactive replication together with reactive data repair strategy (i.e., a *mixed repair strategy*), the actual repair bandwidth consumed when failures occur is smoothed by proactive replication and thus big bursts of repair traffic can be avoided. When configured properly, the mixed strategy may achieve better reliability with a smaller bandwidth budget and extra disk space. In this section, we study the impact of proactive replication to data reliability in the setting of GPGR.

As in Section 2, we still focus on an object and refer to this object by default. When the number of replicas of this object drops below the desired degree K, the system tries to repair the number of replicas to K using reactive repair. And the system also uses reactive rebalance to fill new empty bricks. Once the number of replicas reaches K, the system switches to proactive replication to generate additional replicas for this object.

We extend the model in Section 2 to cover proactive replication by adding states (N, K+1), (N-1, K+1), …, (N, K+2), (N-1, K+2), …, until (N, $K+K_p$), (N-1, $K+K_p$), to the model in Figure 1, where K_p is the maximal number of replicas generated by proactive replication. The calculation of transition rates is given below.

First, for every state (n,k), the two failure transitions λ_1 and λ_2 leaving state (n,k) have the same formulas $\lambda_1 = (n-k)\lambda$ and $\lambda_2 = k\lambda$ as before, because state (n,k) by definition has n online bricks and k of them have replicas of the object. Second, we consider the repair and rebalance transitions μ_1, μ_2 and μ_3 leaving state (n,k). Because reactive repair, rebalance, and proactive replication all evenly reproduce data among bricks, we could logically divide data on a brick into two categories for the purpose of analysis: data maintained by reactive repair and rebalance, called *reactive replicas*, and those generated by proactive replication, called *proactive replicas*. Such a classification does not distort the working of the modeled system and simplifies the analysis.

For the state (n,k) with $k < K$, we have one transition to (n, k+1) for reactive repair, and two transitions to (n+1,k+1) and (n+1,k) for rebalance. Since by our classification reactive repair and rebalance do not need to regenerate proactive replicas, the computation of the transition rates is exactly like in Sections 2.1 and 2.2, except that now we need a new bandwidth allocation. The switch bandwidth and brick bandwidth are divided into three components: p_r for reactive repair, p_l for rebalance, and p_p for proactive replication, and $p_r+p_l+p_p=1$. That is, we restrict proactive replication bandwidth to be p_p percent of total bandwidth, and it is usually small (e.g., 1%). With this allocation, we only need to change the calculations in Table 2 such that p is replaced with p_r and (1-p) is replaced with p_l. The rest calculation of μ_1 μ_2 and μ_3 remains the same for state (n,k) with $k < K$.

We now consider proactive replication and rebalance transitions for state (n,k) with $k \geq K$. We still use μ_1, μ_2 and μ_3 as the transition rates denoted in Figure 1 (b), but they have different meanings now. First, $\mu_2 = 0$, because rebalance does not generate proactive replicas for this object. Thus transition to (n+1,k) is the only transition for rebalance, and $\mu_3 = (N-n)*b_l/d_l$, where b_l and d_l are the same as in Table 2 with p_l replacing (1-p).

Finally, we consider the proactive replication transition from (n,k) to (n,k+1) when $k \geq K$ and its rate μ_1. To calculate μ_1, we need to calculate quantities d_p and

b_p, where d_p is the amount of data for proactive replication in state (n,k), and b_p is the bandwidth allocated for proactive replication, all for one online brick. However, state (n,k) does not provide enough information to derive d_p directly. To avoid introducing another parameter into the state and causing state space explosion, we estimate d_p by calculating the mean number of online bricks, L. Parameter L is calculated in the model using only reactive repair (with p_r bandwidth) and rebalance (with p_l bandwidth). Let $A_p = \min(n, FK_p(N-L)/N)$, which denotes the total number of online bricks that can participate in proactive replication. Then we calculate $d_p = DK_p(N-L)/(NA_p)$, because $(DK_p)/N$ is the amount of data on one brick that are generated by proactive replication, there are $(N-L)$ bricks that lose data by proactive replication, and all these data can be regenerated in parallel by A_p online bricks. The calculation of A_p and d_p does not include a parameter x used in A and $d_{r,i}$, because proactive replication uses much smaller bandwidth than data repair and thus we cannot assume that most of the lost proactive replicas have been regenerated. For b_p, we have $b_p = \min(Bp_p/A_p, bp_p)$ similar to the counterpart with $k<K$. Now we can calculate the transition rate $\mu_1=(K_p+K-k)b_p/d_p$, because there are (K_p+K-k) proactive replicas for the object to be regenerated, and each has the rate b_p/d_p.

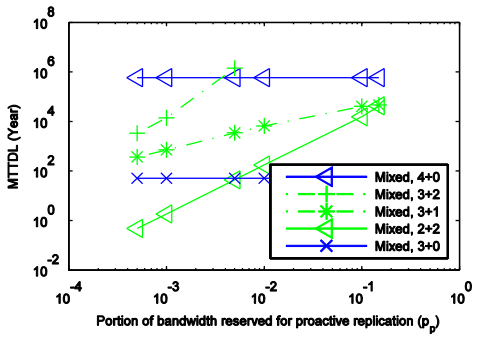

Figure 4. Comparison between pure reactive repair and mixed repair with a limited proactive replication bandwidth. File size=4M. Bandwidth budget for rebalance p_l= 10%. The notion "x+y" means K=x, K_p = y.

Figure 4 compares the reliability achieved by reactive repair and mixed repair with varied bandwidth budget allocated for proactive replication. It also shows different combinations of reactive replica number K and proactive replica number K_p. Figure 4 demonstrates the following results.

First, with increasing bandwidth budget allocated for proactive replication, the reliability of mixed repair is significantly improved, though it is still lower than pure reactive repair with same number of replicas. For example, when proactive replication bandwidth increases from 0.05% to 10%, the reliability of mixed repair with "3+1" combination improves two orders of magnitude, but is still lower than that of reactive repair with 4 replicas (by an order of magnitude). Mixed repair with "2+2" also shows similar trends.

Second, mixed repair provides the potential to dramatically improve reliability using extra disk space without spending more bandwidth budget. Comparing the mixed repair strategies "3+2" with "3+1", we see that "3+2" has much better reliability under the same bandwidth budget for proactive replication. That is, without increasing bandwidth budget, "3+2" provides much better reliability by use some extra disk capacity. Comparing "3+2" with reactive repair "4+0", when the bandwidth budget for proactive replication is above 0.5%, "3+2" provides the same level of reliability as "4+0" (larger bandwidth budget results are not shown because the matrix $I\text{-}Q^*$ is close to singular and its inversion cannot be obtained). Therefore, by using extra disk space, we can dramatically improve data reliability without incurring much burden on system bandwidth.

5. The delay of failure detection and replacement of failed bricks

The model developed in Section 2, called Model 0, assumes that the system detects brick failure and starts the repair and rebalance process instantaneously. In reality, a system usually takes some time, referred to as failure detection delay, to detect brick failures. In this section, we extend Model 0 to Model 1 to capture failure detection delay and study its impact on *MTTDL*. In real systems, failure detection techniques range from simple multi-round heart-beat detection to sophisticated failure detectors. Distributions of detection delay vary in these systems. For simplicity, we assume that the detection delay obeys exponential distribution. In simulation, we will evaluate the impaction of this assumption by using constant detect latency.

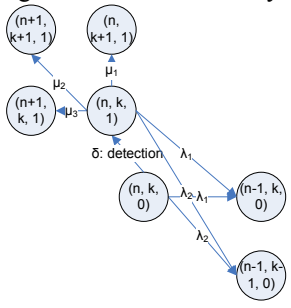

Figure 5. Model 1: consider failure detection delay

One way to extend Model 0 is to split state (n, k) into states (n, k, d), where d denotes the number of failed bricks that have been detected and therefore ranges from 0 to $(N\text{-}n)$. The problem is that the state space is

exploded to $O(KN^2)$. To control the state space, we make a simple approximation by allowing only 0 and 1 for value d: d=0 means that the system has not detected *any* failures and will do nothing, and d=1 means that the system has detected *all* failures and will start the repair and rebalance process (Figure 5). As long as the detection delay is far less than the interval of two consecutive brick failures---an assumption holds for most of real systems, the approximation is reasonable. We call the model in Figure 5 Model 1.

The transition rates of Model 1 are calculated as follows. After a failure occurs, state (n, k, d) transits to state (n-1, k, 0) at rate λ_1 if no replica is lost or state (n-1, k-1, 0) at rate λ_2 if one replica is lost. To be conservative a state is always transited to an undetected state (d=0) after a failure. The calculation of rates λ_1 and λ_2 are the same as in Model 0. The transition from state (n, k, 0) to state (n, k, 1) represents failure detection, the rate of which is denoted δ ($1/\delta$ is the mean detection delay). In state (n, k, 0) there is no transition for data repair and rebalance because failures have not been detected yet. State (n, k, 1) could transit to (n, k+1, 1), (n+1, k+1, 1), or (n+1, k, 1) with respective rates μ_1, μ_2, and μ_3, representing data repair and rebalance transitions. The calculations of μ_1, μ_2, and μ_3 are the same as in Model 0.

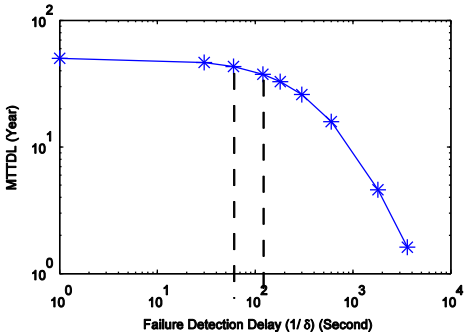

Figure 6. Impact on *MTTDL* by failure detection delay.

Figure 6 shows the $MTTDL_{sys}$ with respect to various mean detection delays. The result demonstrates that a failure detection delay of 60 seconds has only small impact to $MTTDL_{sys}$ (14% reduction), while a delay of 120 seconds has moderate impact (33% reduction). Such quantitative results can provide guideline on the speed of failure detection and helps the design of failure detectors.

We further extend the model to study the impact caused by delay of replacement of failed bricks. We found that replacement delay from 1 day to 4 weeks does not lower the reliability significantly. This is because replacement delay only slows down data rebalance but not data repair, and data repair is much more important to data reliability. With this result, we can conclude that (in environments similar to our settings) data reliability is not an important concern in determining brick replacement frequency. System administrators can choose long replacement delay to reduce maintenance cost, or determine the delay frequency based on other more important factors such as performance.

6. Verification and Tuning with Simulation

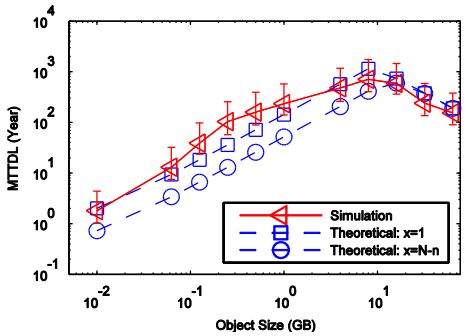

Figure 7. *MTTDL* of simulation and theoretical analysis. 95% confidence intervals for the simulation results are included.

We verify our analytical results with event-driven simulations. Due to space limitation, we present only a few of them. The simulation results are also used to tune parameter x (the number of failed bricks that account for repair data). The event-driven simulation is down to the details of each individual objects. It includes more realistic situations that have been simplified in the analysis, and is able to verify the analysis in a short period of time without setting up an extra system and running it for years.

The simulation runs as follows. Initially, objects are distributed uniformly at random across bricks, which are all connected to a switch. Brick life time follows exponential distributions, but data transfer time, detection delay, and replacement delay are constants, which are closer to the situations in practice. Once a brick fails, a scheduler randomly select source-destination pairs for both data repair and rebalance. The simulation is stopped once some object loses all its replicas.

Since our simulation needs to simulate the behavior of each individual object in the system, we cannot achieve the same large scale as our analytical framework. Thus, we scale down our parameters so that the simulation can complete in a reasonable amount of time (two to three days using one machine with two AMD 2.8GHz Opteron 280 processors and 16GB memory). In particular, we run the simulations with 150 bricks, 4 Terabyte of unique data, 125MB/s backbone bandwidth, 12.5MB/s brick bandwidth, 20%

bandwidth allocated for rebalance, and 3 replicas for each object, and the mean life time of a brick is 0.1 year. Each data point is the average of 10 simulation runs.

Figure 7 compares the results of the simulation against the theoretical calculations in the basic model. We can see that the simulation results match very well with the trend (as object size increases) predicted by our analytical framework. Since the simulation is at the individual object level, it naturally accounts for partial repair and rebalance, which means that the data repair and rebalance effort spent in one state will not be lost when the system transitions to another state. This helps us to tune parameter x in Section 2.2, which denotes the approximate number of failed bricks whose data still need to be repaired when the system has lost N-n bricks. The conservative approximation of x=N-n does lower the reliability prediction (sometimes to an order of magnitude), but when x=1, the theoretical prediction aligns with the simulation results quite well (most data points fall into the 95% confidence interval). Since the failure rate in our simulation setting (0.1 year brick life time with 150 bricks) is higher than our analytical setting (3 year brick life time with 1024 bricks) while the repair bandwidth in our simulation is much lower (125MB/s vs. 3GB/s), we therefore use x=1 in our analytical results under our sample setting.

Figure 8 compares simulation results with the theoretical prediction when we consider failure detection delay (Model 1). The results again show that when we use x=N-n we obtain a conservative prediction while when we use x=1 we obtain a theoretical prediction that is both close in trend and in values to the simulation results.

We also can see that results predicted by our analytical framework assuming that data transfer time and detection delay follow exponential distributions match reasonably well to the simulation results that are produced with constants latency instead.

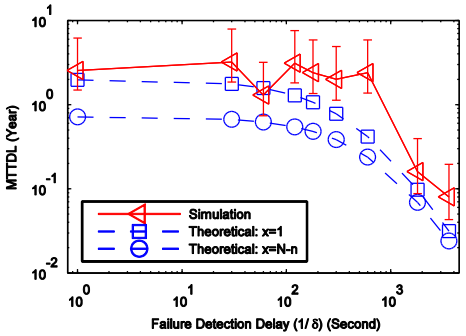

Figure 8. Simulation and analytical results with detection delays. 95% confidence intervals for the simulation results are included.

Overall, our simulation results both verify the correctness of our analytical framework and help to tune a parameter so that we can obtain fairly accurate prediction from our analytical framework.

7. Related Work

Reliability is critical to storage systems, and it has been intensively studied. In [5] [22], researchers studied the reliabilities of RAID systems in term of *MTTDL*, using Markov models with independent and exponentially distributed disk failures. Our model could be viewed as an extension of these models to distributed bricks storage systems, which have bounded network bandwidth and different repair strategies.

Many researches [3][9][24][13] studied similar brick storage systems, focusing on replica placement issues for various reasons other than data reliability. In Farsite [3], Douceur and Wattenhofer studied dynamic replica placement strategies that improved the overall availability of files [7][8]. Renesse and Schneider in [17][18] studied support for high throughput and availability by chain replication and pseudo-random placement. GFS [9] used random placement to improve data repair performance, but did not provide a study on the resulted reliability.

In [12], Lian et.al studied the tradeoff of reliability between sequential and random placement using a Markov model, and proposed the stripe placement strategy that groups small objects together to improve data reliability. Our current work significantly extends the work in [12] in the following aspects. First, we extend the Markov model to consider data rebalance, and include parameter k in the state to make the model more precise and enable us to study proactive replications. Second, we study the impact of switch topology and corresponding placement strategy. Third, we extend the model to consider realistic failure detection delays and brick replacement delays. Finally, we provide a more rigorous treatment to the estimation of π, the number of independent objects.

Ramabhadran et.al [16] studied a single object's *MTTDL* in long-running replicated system, using a Markov chain model that only captures replica dynamics and assuming exogenous available repair bandwidth.

Sit et.al [21] proposed proactive maintenance for distributed hash table in wide-area storage and used simulation to study its durability in term of fractional availability, not *MTTDL*. Chun et.al [6] tried to provide separate reliability and availability of data in the environment where there were transient failures and permanent failures, which is different from our settings.

Yu et.al [23] focused on the operation availability involving multi-object with different assignment of object replicas to machines in wide-area network. Nath et.al [15] studied erasure-coded data availability in face of correlated failure in wide-area network via live deployment and simulation. Both studies focus on availability rather than reliability.

8. Conclusion and Future Work

In this paper, we present a framework for analyzing brick storage reliability in the dynamics of brick failures, data repair, data rebalance, and proactive replication. We applied the framework to a number of settings and provide quantitative results to show how data reliability can be affected by switch topology, proactive replication, failure detection delay, and brick replacement delay. We also use simulation results to verify and refine our analytical framework. The framework is able to provide important guidelines to storage system designers and administrators on how to fully utilize system resources (extra disk capacity, available bandwidth, switch topology, etc) to improve data reliability while reducing system and maintenance cost.

While these results are concrete, they are only partially useful. Ultimately, what we want to solve is a multi-constraint problem for brick storage design and maintenance: given the performance, reliability and capacity targets, define the least expensive architecture and also suggest the best policies and parameters. We believe that our framework is a significant step towards this goal.

Reference

[1] "Reference information: The next wave", The Enterprise Storage Group, June 2002.

[2] http://www.ee.duke.edu/~kst/software_packages.html

[3] A. Adya, et al, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment", in Proc. of the 5th OSDI, 2002.

[4] R. Billinton and R. N. Allan. "Reliability Evaluation of Engineering System: Concepts and Techniques", Perseus Publishing, 1992.

[5] W. A. Burkhard, J. Menon, "Disk Array Storage System Reliability", in Proc. of Symposium on Fault-Tolerant Computing, 1993.

[6] B.G. Chun, et. al, "Efficient Replica Maintenance for Distributed Storage Systems", in Proc. of NSDI'06, May 2006.

[7] J. R. Douceur and R. P. Wattenhofer. "Competitive Hill-climbing Strategies for Replica Placement in a Distributed File System", in Proc. of the 15th Symp. on Distributed Computing. Oct. 2001.

[8] J. R. Douceur and R. P. Wattenhofer, "Optimizing File Availability in a Secure Serverless Distributed File System", in Proc. of SRDS, 2001.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System", in Proc. of the 19th SOSP, Oct. 2003.

[10] G. A. Gibson, et al. "A Cost-Effective, High-Bandwidth Storage Architecture", in Proc. of the 8th ASPLOS, Oct. 1998.

[11] Jim Gray, Catharine Van Ingen, "Empirical Measurements of Disk Failure Rates and Error Rates", Technical Report, MSR-TR-2005-166, Dec. 2005

[12] Q. Lian, W. Chen, Z. Zhang, "On the Impact of Replica Placement to the Reliability of Distributed Brick Storage Systems", in Proc. of 25th ICDCS, Jun 2005.

[13] J. MacCormick, et.al, "Boxwood: Abstractions as the Foundations for Storage Infrastructure", in Proc. of OSDI'04, Dec. 2004.

[14] Heidelberger, P., Muppala, J. K., and Trivedi, K. S. 1996, "Accelerating mean time to failure computations", Perform. Eval. 27-28 (Oct. 1996), 627-645.

[15] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan, "Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems ", in Proc. of NSDI'06 , May 2006.

[16] S. Ramabhadran, J. Pasquale, "Analysis of Long-Running Replicated Systems", in Proc. of INFOCOM, 2006.

[17] R. van Renesse, F. B. Schneider, "Chain replication for Supporting High Throughout and Availability", in Proc. of OSDI'04, Dec. 2004.

[18] R. van Renesse, "Efficient Reliable Internet Storage", in Proc. of Workshop on Dependable Distributed Data Management, 2004.

[19] Y. Saito, et al, "FAB: Building Distributed Enterprise Disk Arrays from Commodity Components", in Proc. of the 11th ASPLOS, Oct. 2004.

[20] Seagate Technology. "Estimating Drive Reliability in Desktop Computers and Consumer Electronic Systems", http://www.digit-life.com/articles/storagereliability/, 2006.

[21] E. Sit, et.al, "Proactive Replication for Data Durability", in Proc. of IPTPS 2006.

[22] Q. Xin, E. L. Miller, D. D. E. Long, S. A. Brandt, T. Schwarz, W. Litwin, "Reliability Mechanisms for Very Large Storage Systems", in Proc. of 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems & Technologies, Apr., 2003

[23] H. Yu, P. B. Gibbons, and S. Nath, "Availability of Multi-Object Operations", in Proc. of NSDI'06, May 2006.

[24] Z. Zhang, et al, "RepStore: A Self-Managing and Self-Tuning Storage Backend with SmartBricks", in Proc. of the first IEEE International Conference on Autonomic Computing, May 2004.