On the Impact of Replica Placement to the Reliability of Distributed Brick Storage Systems

Qiao Lian

Wei Chen

Zheng Zhang

June, 2005

MSR-TR-2005-71

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

On the Impact of Replica Placement to the Reliability of Distributed Brick Storage Systems

Qiao Lian, Wei Chen, Zheng Zhang
Microsoft Research Asia
{t-qiaol, weic, zzhang}@microsoft.com

Abstract

Data reliability of distributed brick storage systems critically depends on the replica placement policy, and the two governing forces are repair speed and sensitivity to multiple concurrent failures. In this paper, we provide an analytical framework to reason and quantify the impact of replica placement policy to system reliability. The novelty of the framework is its consideration of the bounded network bandwidth for data maintenance. We apply the framework to two popular schemes, namely sequential placement and random placement, and show that both have drawbacks that significantly degrade data reliability. We then propose the stripe placement scheme and find the near-optimal configuration parameter such that it provides much better reliability. We further discuss the possibility of addressing the problem of correlated brick failures in our analytical framework.

1. Introduction

Storage solution using clustered "smart bricks" connected with LAN is becoming an increasingly attractive alternative to the more expensive SAN (storage-area network) solution. Some of the exemplary systems include Petal [11], NASD [9], GFS [8], FAB [7], Repstore [22], and Boxwood [14]. A smart brick is essentially a stripped down PC with a CPU, memory, network card, and a large disk. For these systems, providing strong data reliability is confronted with new challenges, because inexpensive commodity disks are more prone to permanent failures and failures are far more frequent in large systems. To guard against permanent loss of data, replication is often employed. If some replicas are lost due to disk failures, other replicas are still available and can be used to regenerate new replicas to maintain the same level of reliability.

Replica placement refers to the strategy of placing replicas among the participating bricks. The two widely used replica placement schemes are staggered sequential placement like in chained declustering [10] used by Petal [12] as well as in many proposals based on DHT (distributed hash table) [4][17], and the totally random placement like in GFS [8]. Mirroring can be viewed as a degenerated special case of sequential placement.

Replica placement can significantly affect the reliability of the system due to two factors. The first is the repair speed: the more bricks participate in the data repair process, subject to the available network bandwidth, the sooner that the reliability level returns. The second is the sensitivity to multiple and concurrent failures: the more permutation choices that the placement generates, the more likely a random failure of several bricks will wipe out some data permanently. These two factors are conflicting in nature. For instance, the random placement has very fast repair speed, but is prone to concurrent failures, whereas the sequential placement is precisely the opposite.

The contributions of the paper are mainly twofold. First, we provide a systematic framework not only to identify, but also to reason and quantify the impact of the replica placement policy to system reliability. In particular, our result points out that under different parameters, random and sequential placement can have vastly different results. Our framework captures the bounded available network bandwidth for data maintenance, something no other models have done.

Second, the insight that we gain from the first result leads us to propose the *stripe placement* scheme which attempts to achieve the best balance between the two competing forces. While it is difficult to derive a closed form, we do provide the near-optimal configuration parameter verified with simulations. This is the second contribution of this paper.

Moreover, we also sketch a proposal to extend our framework to deal with correlated failures that frequently occur in practice.

The roadmap of the paper is as follows. In Section 2, we discuss the replica placement schemes and the data reliability metric. In Section 3, we present the analytical framework for the reliability study. In Section 4, we apply the framework to the two placement schemes and compare the results. In Section 5, we describe the stripe placement scheme, and find the near-optimal parameter for the scheme. We discuss how to deal with correlated failures in Section 6. Related work is summarized in Section 7, and we conclude the paper in Section 8. There are a lot of symbols are using in this paper, we put also gives symbol table in Appendix A.

2. Replica Placement and Reliability Metric

Without loss of generality, we consider an ordered array of N bricks on which replicas are placed. The number of replicas of an object is called the *replication degree* of the object, and it is denoted by k. Replication degree may differ from object to object, but for simplicity we assume all objects have the same k. The responsibility of replica placement is to designate the bricks on which the replicas are hosted. All other issues, such as the interface of the system, are orthogonal to the reliability study.

Each individual brick may fail permanently and lose all replicas stored on the brick. In this paper, we equate brick failures with disk failures, since disk failures ultimately cause data loss. When a brick fails, to keep data reliability of the system at the same level, the system needs to automatically regenerate the lost replicas at the remaining bricks. This replica regeneration process is called *data repair*.

2.1 Data reliability metric *MTTDL*

To measure the data reliability of a system, we use the metric *MTTDL* – mean time to data loss in the entire system. *MTTDL* indicates, after the system is loaded with data objects, how long on average the system can sustain before it permanently loses the *first* data object in the system. This is a metric widely used in storage literature, e.g. on RAID storage [2].[1]

There are two important factors that affect data reliability. The first one is the speed of data repair. Fast data repair means that the lost replicas are likely to be repaired before further brick failures, so it reduces the time window in which concurrent brick failures occur and wipe out all replicas of some object.

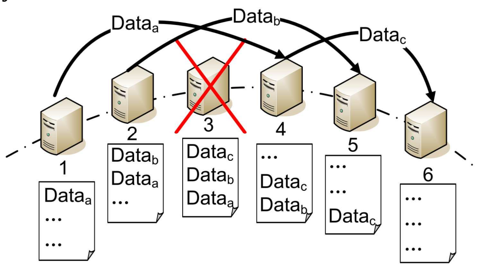
The second factor is the likelihood of data loss when concurrent brick failures do occur. When k random bricks fail concurrently in the system, the likelihood that some object whose k replicas are located on the k failed bricks depends on the placement scheme used, as we will discuss shortly. In general, the more likely the concurrent brick failures happen to wipe out all replicas of an object, the less reliable the system is.

2.2 Sequential placement

Sequential placement is simple in nature: one of the brick acts as the lead brick, and the k replicas are placed on the lead brick and its k-1 followers. This is what the *chained declustering* [10] does and is employed in Petal [11]. This scheme is also a common strategy in peer-to-peer wide-area storage systems like CFS [4] and PAST [17], in which the lead brick is typically identified by the hash of the object.

With sequential placement, when a brick fails, the k neighboring bricks on each side of the failed brick can participate in brick repair (Figure 1). The limited parallel repair degree leads to slow repair speed, which then negatively affects the data reliability of the system.

However, the restrictive nature of placement reduces the sensitivity to concurrent failures. If and only if k simultaneous failures occur on k consecutive bricks in the ordered array can any object be lost. This is unlikely when N is much larger than k. Thus, sequential placement has a low likelihood of data loss when concurrent failures occur, which improves data reliability.



Note: Replication degree is 3. When brick 3 fails, data repair can be carried out as brick 1 copying Data$_a$ to brick 4, brick 2 copying Data$_b$ to brick 5, and brick 4 copying Data$_c$ to brick 6.

Figure 1. Data repair in sequential placement.

2.3 Random placement

In *random placement*, replicas are placed randomly among the N bricks, and this is used in the work of [8][19].[2] The main objective here is to improve the speed of data repair. With random placement, when a brick fails, the replicas on the failed brick can be found on many other bricks, and thus many bricks can participate in data repair in parallel, resulting in faster

[1] One may also consider the amount of data loss when it happens. However, when the goal of a storage system is to provide nearly no-data-loss reliability, *MTTDL* is a more important metric than the amount of data loss. Thus the paper focuses on the analysis of *MTTDL*.

[2] An indexing scheme is needed for random placement to access all objects in the system. However, indexing is orthogonal to our study of reliability, so we will not discuss it in detail.

data repair speed. This is its main advantage over sequential placement in improving data reliability.

However, crashing k random bricks will likely remove all the replicas of some objects with random placement. In the extreme case when there is a large volume of objects in the system and therefore the actual placement choices have exhausted all possible combinations, any k crashes cause data loss. High sensitivity to multiple and concurrent failures, therefore, is the drawback of the random placement scheme.

It is intuitive to see that neither placement scheme is perfect. As we will reveal in Section 4, under different circumstances their difference can be dramatic.

3. Analytical Framework

In this section, we present the analytical framework for the data reliability analysis. The framework shows how to derive *MTTDL* from known system parameters. This framework can be applied to different object placement schemes, as we will show in the next section.

The novelty of the framework is its consideration of the bounded network bandwidth available for data repair, which directly affects data repair speed.

3.1 System model for analysis

We consider a system with N bricks and the replication degrees of all objects are k. The average amount of data stored on each brick is c. We assume a reasonable amount of free space on each brick for data repair. Brick failures follow an exponential distribution with *MTTF* (mean time to failure) as its mean. We first assume that each brick fails independently, and later we will consider a model for correlated brick failures. We do not model transient failures of bricks that only affect data availability but not affect data loss. When a brick fails, we assume a new brick is added into the system immediately to keep the system scale at N all the time.

All bricks are connected in a LAN with a root switch. The network provides certain bandwidth for data repair traffic, and the bound of which is given by B, which is called the *backbone bandwidth*. The backbone bandwidth can be viewed as a certain percentage of the bandwidth of the root switch that is allowed for data repair traffic, because in many simple topologies, all (or nearly all) data repair traffic goes through the root switch.

We do not separately consider network failures. A network failure does not cause data loss directly, but it may reduce the data repair speed, and thus affect data reliability. We fold this aspect into the available data repair bandwidth B.

3.2 Analysis

3.2.1. Introducing $MTTDL_{obj}$. To conduct the analysis for *MTTDL*, we first introduce an intermediate metric $MTTDL_{obj}$, which is the mean time to data loss for an *arbitrary* object. $MTTDL_{obj}$ measures the data reliability of an individual object stored in the system. If the system contains m objects, and these m objects have independent data loss distributions, then we have $MTTDL = MTTDL_{obj} / m$. Intuitively, this is because that each object has a data loss rate of $1/MTTDL_{obj}$, and when they are considered together in a system, the total data loss rate is $m/MTTDL_{obj}$ since their individual data loss behaviors are independent.

Of course, when object replicas are placed in the system, their data loss behaviors depend on the failures of the bricks, and thus may not be independent of each other. In particular, if the replicas of two objects are co-located at the same set of bricks, their data loss behaviors are perfectly correlated. In this case, they should be considered as one object instead of two independent objects.

For the above reason, we only consider objects whose replica placements are different. Let m be the total number of different replica placement combinations under a placement scheme. We thus have [3]

$$MTTDL = MTTDL_{obj} / m. \qquad (1)$$

In a system with N bricks, m could be as large as $C(N,k)$.[4]

3.2.2. Markov Chain Model. To analyze $MTTDL_{obj}$, we introduce a Markov model as in Figure 2 to model the evolution of the system with brick failures and data repair. In the model, state i represents the state of the system where exactly i bricks have failed, and the lost replicas on the failed bricks have not been completely repaired.

The arcs in the figure represent the transitions between different states in the system. From state i *to* $i+1$, one more brick failure occurs, and this could occur on any of the N-i remaining bricks. Thus, the rate of transition is $(N\text{-}i)/MTTF$, since the brick failures are independent. Transition from state i to state 0 represents

[3] The formula is still an approximation, because different objects may have some but not all their replicas co-located on the same set of bricks and thus their data loss behaviors are correlated. However, such co-locations are dictated by the replication degree k. Thus, when the system scale N is much larger than the replication degree k, the correlated data loss of objects caused by partial co-locations can be ignored. For sequential placement, we compared the approximation with an accurate analysis and the result shows that the approximation matches with the accurate analysis very well.

[4] $C(x,y)$ denotes the total number of combinations of picking y objects from x objects.

that data repair is completed before a new brick fails. Let *MTTR*(*i*) denote the mean time to repair all the failed replicas in state *i*. Thus the transition rate from state *i* to state 0 is 1/*MTTR*(*i*). *MTTR*(*i*) depends on the size of data to be repaired and the available bandwidth for repair, and it will be determined shortly. In the next section, we will show the advantage of random placement over sequential placement in that random placement has a much smaller *MTTR*(*i*).

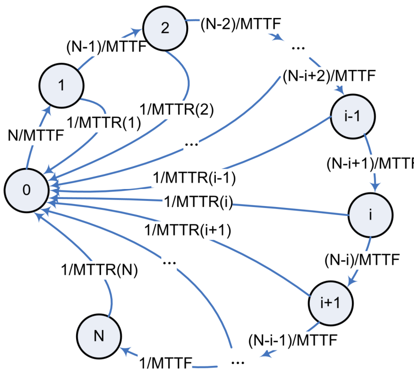


Figure 2. Markov model for reliability analysis

The Markov model of Figure 2 assumes that there is no transition from state *i* back to state *j* with 0<*j*<*i*, which means that data repair for all failed bricks complete at the same time. In reality, data repair for a brick that failed early is likely to be completed early, even without centralized scheduling of data repair. Hence the assumption is a conservative one.

The Markov model of Figure 2 models the evolution of the entire storage system, as opposed to other models (e.g. [20]) that only model the evolution of one object. The reason is that in our environment, data repair traffic is limited by the backbone bandwidth. Thus, more brick failures are likely to slow down data repair because more data repair traffic are sharing the limited bandwidth. Therefore, we have to look at the state of the entire system to determine the speed of data repair.

3.2.3. Deriving *MTTDL_{obj}*. First, we define *MTBF*(*i*) to be the mean time between two consecutive occurrences of state *i* in the Markov model. Each time when the system is in state *i*, there is a chance that a particular object is lost. We denote *L*(*i*) as the probability that the object is lost when *i* bricks fail concurrently. We then have the following formula to compute $MTTDL_{obj}$.

$$MTTDL_{obj} = [\sum_{i=k}^{N} MTBF^{-1}(i) \cdot L(i)]^{-1} \cdot \qquad (2)$$

The reasoning of the above formula is as follows. We consider a stochastic process where the event is an object being lost, and we assume that as soon as the object is lost, a new object is generated with exactly the same probability distribution of the time to data loss. Let *N*(*t*) be the number of object loss events by time *t*. Thus we have

$$MTTDL_{obj} = \lim_{t\to\infty} \frac{t}{N(t)} \cdot \qquad (3)$$

Let *N*(*i*,*t*) be the number of times state *i* appears by time *t*. Each time state *i* appears, with probability *L*(*i*) the object is lost, so when considering state *i*, the event of object loss by time *t* occurs *N*(*i*,*t*)·*L*(*i*) times. When summing up all the different states together, we have

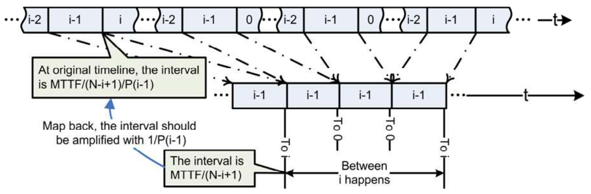
$$N(t) = \sum_{i=k}^{N} N(i,t) \cdot L(i) \cdot \qquad (4)$$

Moreover, by the definition of *MTBF*(*i*), we have

$$MTBF(i) = \lim_{t\to\infty} \frac{t}{N(i,t)} \cdot \qquad (5)$$

Plug in (4) and (5) in to (3), we thus obtain the formula in (2).

Probability *L*(*i*) is easy to compute. An object with replication degree *k* can be located on *C*(*N*,*k*) possible combinations of *k* bricks, where *N* is the total number bricks. When *i* bricks fail concurrently, totally *C*(*i*,*k*) combinations will cause data loss. Thus the probability of losing the object when *i* bricks fail is *C*(*i*,*k*) / *C*(*N*,*k*).



i-2
i-1
i
i-2
i-1
0
i-2
i-1
0
i-2
i-1
i
t
At original timeline, the interval is MTTF/(N-i+1)/P(i-1)
i-1
i-1
i-1
i-1
t
Map back, the interval should be amplified with 1/P(i-1)
To i
To 0
To 0
To i
The interval is MTTF/(N-i+1)
Between i happens

Figure 3. Extracting timing period for state i-1 for the derivation of MTBF(i).

We now need to derive *MTBF*(*i*). To do so, consider a continuous timeline separated by different time segments corresponding to different states of the system (Figure 3). Based on the Markov model, on the time line a period of state *i*-1 is followed either by a period of state *i* or a period of state 0. We extract the time period for state *i*-1 out and put them next to each other to build a new time line (Figure 3). That is, the new timeline only records the period when the system stays in state *i*-1. In the new timeline, the boundary of each period corresponds to either the transition from state *i*-1 to state *i*, or the transition from state *i*-1 to state 0. From one transition to state *i* to the next transition to state *i*, one of the remaining *N*-*i*+1 bricks fail. Since brick failures are memoryless, we can simply concatenate the time segments of state *i*-1 together in the new timeline without changing its stationary probabilistic behavior. Therefore, the mean time between two transitions to state *i* in the new timeline is *MTTF*/(*N*-*i*+1).

Let *P*(*i*) be the probability of the system staying in state *i*. Therefore, the new timeline we constructed only corresponds to the *P*(*i*-1) portion of the original timeline. In other words, when we match the period between two transitions to state *i* in the new timeline back to the original timeline, the period is amplified by a factor of 1/*P*(*i*-1). Hence, in the original timeline, the mean time between two occurrences of state *i* is

$$MTBF(i) = \frac{MTTF}{(N-i+1)\cdot P(i-1)}.$$

The next calculation is for probability *P*(*i*). This is given by the fact that in the equilibrium state of the Markov model, the total incoming transition rate must be equal to the total outgoing transition rate. The formulas are as follows.

$$\begin{cases} \dfrac{P(i)}{P(i-1)} = \dfrac{(N-i+1)/MTTF}{(N-i)/MTTF + 1/MTTR(i)}, \text{ for all } i \geq 1 \\ \sum P(i) = 1 \end{cases}$$

3.2.4. Deriving *MTTR(i)*. *MTTR*(*i*) depends on both the amount of data to repair and repair bandwidth. Let *D*(*i*) and *rb*(*i*) be the amount of data to repair and the repair bandwidth in state *i*, respectively. Let *T* be the time to detect a failure in the system (10sec is used for all analysis and simulation). Then

$$MTTR(i) = T + D(i)/rb(i). \qquad (6)$$

The amount of data to repair *D*(*i*) depends on both *c* (the amount of data in the last failed brick), and the amount of the un-repaired data *ur*(*i*) left from the previous state *i-1*.

$$D(i) = ur(i) + c.$$

The amount of un-repaired data left from the previous state *ur*(*i*) depends on (a) the total amount of the previous state's data to be repaired *D*(*i*-1), (b) the mean time to the next failure in the previous state *mf*(*i*-1), and (c) the previous state's repair bandwidth *rb*(*i*-1).

$$ur(i) = \begin{cases} D(i-1) - rb(i-1)\cdot mf(i-1) & (when \quad D(i-1) > rb(i-1)\cdot mf(i-1)) \\ 0 & (when \quad D(i-1) \leq rb(i-1)\cdot mf(i-1)) \end{cases}$$

In state *i*-1, the mean time to next failure *mf*(*i*-1) is *MTTF*/(*N*-*i*+1). So we have

$$D(i) = \max[D(i-1) - rb(i-1)\cdot MTTF/(N-i+1), 0] + c$$

Once *rb*(*i*) is known, *D*(*i*) can be calculated iteratively by the above formula.

The repair bandwidth *rb*(*i*) at state *i* varies with different placement schemes. In the next section we will determine this value for both sequential placement and random placement.

4. Comparing Sequential Placement with Random Placement

The previous section provides the general framework to analyze the data reliability of a distributed brick storage system with a bounded backbone repair bandwidth. The two terms undecided in the analysis are: (a) *m*, the possible replica placement combinations in the system; and (b) *rb*(*i*), repair bandwidth at state *i*. It is not hard to see that the larger the *m*, the worse the data reliability, while the larger the *rb*(*i*), the faster the data repair can be completed and thus the better the data reliability.

Table 1. **Key differentiating quantities for sequential placement and random placement**

	m	$rb(i)$
Sequential	N	$\min(B, b\cdot k\cdot i/2)$
Random	$\min(C(N,k), (N\cdot c)/(k\cdot s))$	$\min(B, b\cdot(N-i)/2)$

These two terms vary among different placement schemes. In this section, we determine the two terms for both sequential placement and random placement and compute the reliability of the two schemes. Table 1 lists the results and the explanation follows, where *s* denotes the average object size, and *b* denotes the brick bandwidth.

4.1 Sequential placement

In sequential placement, replicas are restricted to be placed on *k* consecutive bricks in the ordered array of bricks. This restriction leads to only *N* possible placement combinations, i.e. *m*=*N*, which benefits the reliability of the system.

To calculate the repair bandwidth, let *b* be the maximum bandwidth of a brick. When one brick fails, the replicas on the failed brick need to be regenerated on the *k* consecutive bricks after the failed brick. So the repair bandwidth could reach *b*·*k*. However, among these *k* bricks, at least one brick would also serve as the source for data repair (e.g., brick 4 in Figure 1), bringing the effective bandwidth only to a half, i.e. *b*·*k*/2. When *i* concurrent failures occur, this gives *b*·*k*·*i/*2 (we ignore the situation when multiple failures are within the range of *k*, and this makes the result optimistic). Also, the maximum repair bandwidth cannot exceed the backbone bandwidth *B*. Therefore, we have

$$rb(i) = \min(B, b \cdot k \cdot i / 2) \cdot$$

4.2 Random placement

In random placement, replicas are scattered randomly among the bricks in the system. When one brick fails, many other replicas contain the replicas that are lost on the failed brick. So many replicas can act as the source of data repair, vastly bringing up the degree of parallel repair and hence the repair bandwidth.

Quantitatively, the repair bandwidth of the random placement scheme is given by $rb(i)=\min(B, b(N-i)/2)$. The term $b(N-i)/2$ means that when a brick fails, half of the remaining bricks contain the replicas need to be generated, and they copy the replicas to the other half of the bricks, which is a good-case scenario but can be closely approximated.

Comparing $rb(i)$ of the two schemes as listed in Table 1, it is clear that the repair bandwidth of the random placement is much higher than that of sequential placement, for relatively small i's that are mostly relevant to the data reliability. This is the advantage of the random placement scheme.

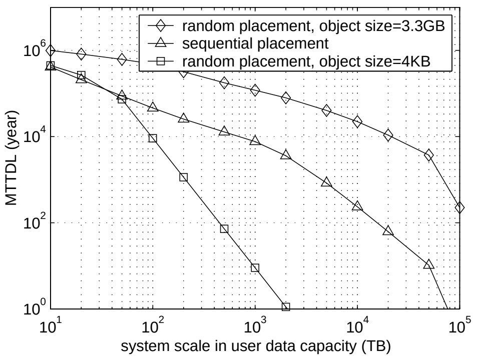
For the possible placement combination m, it depends on the number of objects, which is determined by the size of the object in our environment. Let s denote the average size of an object in the system. When the system has N bricks with the amount of data stored on each brick being c and the replication degree of k, the number of objects in the system is $N{\cdot}c/(k{\cdot}s)$. Thus, the possible placement combination m is given by $\min(C(N,k), N \cdot c/(k \cdot s))$.[5] This value could be much larger than N, the corresponding value of m in sequential placement, especially when the object size is small. This is the major drawback of the random placement scheme that significantly reduces the reliability of the system.

4.3 Comparison

Using the result in Table 1, we can calculate the *MTTDL* given a set of system configuration parameters. Figure 4 shows the analytical result comparing Sequential placement with random placement.

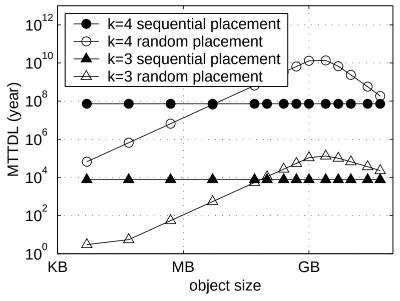
The figure shows several results. First, after the system scale passes a certain point, all schemes essentially stop working: bricks fail so frequently that, with the network bandwidth staying the same, data repair cannot keep up with the brick failures, and the data reliability drops significantly. This means that given a certain backbone bandwidth, the system has a scalability limit.

Second, given the same user capacity, random placement is sensitive to object sizes. The sequential placement scheme is better than random placement for small object size but is worse than that with large object size.



Note: The user data capacity of the system is c·N/k. MTTF=1000days, k=3, B=3GB/s, b=20MB/s, c=500GB.The object size of 4KB in random placement is when m reaches C(N,k).

Figure 4. MTTDL of a system vs. the system scale, with sequential placement and random placement.



Note: The user capacity of the system is fixed at 1PB=1000TB. Other system parameters are the same as in Figure 4.

Figure 5. Comparing sequential placement with random placement when varying object size.

Figure 5 further illustrates the effect of object size on the reliability of the system under the same system scale. In sequential placement, reliability is not affected by object size, because data loss is determined only by the concurrent failures of k consecutive bricks. However, in random placement small object size means a large number of objects, and thus they are more likely

[5] More rigorously, m is the expected number of possible placement combinations for random placement, and it is slightly smaller than $\min(C(N,k), N \cdot c/(k \cdot s))$. We ignore this minor difference in our calculation.

to exhaust the possible placement combinations, giving low reliability. When the object size is large, the possible combinations are small, and the benefit of parallel data repair wins over and thus the data reliability is better than sequential placement. But if the object size continues to grow larger, the benefit of parallel repair diminishes and the reliability in random placement returns to the same level as sequential placement.

The figure also compares the result of replication degree of 3 versus 4. Replication degree of 4 provides close to four orders of magnitude better reliability. In general replication degree of 3 or 4 is suffice to provide enough reliability for most systems.

Overall, no single placement scheme wins in all cases. Random placement gains in fast parallel repair but resulting in too many possible placement combinations when the objects are small, while sequential placement restricts possible placement combinations but is much slower in data repair.

5. Stripe Placement for Near-Optimal Reliability

Ideally, what we want is the optimal reliability provided by the random placement scheme at its optimal object size, but extending it such that the reliability remains at the same level even for small object sizes. To achieve this effect, we need to group small objects together to make them behave like a large object in terms of placement and repair. Then we can significantly reduce the possible placement combinations that the random placement suffers from, while maintaining a good repair speed. This is the *stripe placement* scheme we introduce in this section.

5.1 Stripe placement

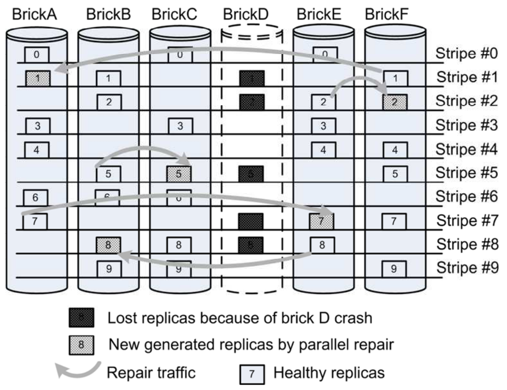


Figure 6. Stripe placement and stripe repair.

We group small object replicas together to form a large *chunk*, which is the unit for placement and repair. The sizes of the chunks are the same and will be determined later. The k replicated chunks of the same set of objects form a set called a *stripe*. A stripe migrates among the bricks in the system with brick failures and data repairs. Figure 6 illustrates the concept of stripe and its repair.

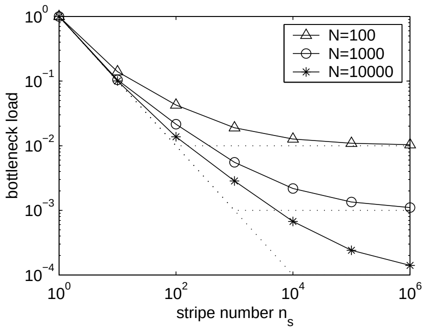
Let *stripe number* n_s be the number of different stripes that can be hosted by one brick. Stripe number determines the degree of parallel data repair. When a brick hosting n_s stripes fails, n_s different chunks need to be repaired and thus the parallel repair degree is at most n_s. We will determine the optimal n_s shortly. Intuitively, n_s should be related to the backbone bandwidth B such that the backbone bandwidth can be fully utilized.

We use random placement of chunks and randomly selecting chunk repair sources and destinations to manage the stripes, which is simple in a distributed environment to allow the parallel repair degree to be close to n_s in spite of brick failures and repairs. In addition to managing random placement of chunks, stripe placement also needs to manage the grouping of objects into stripes. The grouping needs to guarantee that when a new object is added into a stripe, every chunk within the stripe should have enough space to accommodate one replica of the object. To do so, when the stripe is first created, every chunk in the stripe should pre-allocate enough space for the entire chunk. Other management details are omitted here.

5.2 Finding the optimal n_s

We now need to find the optimal n_s so that the reliability of the system is the highest. To do so, we use the analytical framework to calculate the reliability of the system for different n_s, and then locate the n_s that provides the optimal reliability.

First, for the number of possible placement combinations, we have $m=n_s \cdot N/k$, since each brick hosts n_s stripes and each stripe is hosted by k bricks. Second, for the repair bandwidth, ideally all n_s chunks on the failed brick will be repaired by n_s different pairs of sources and destinations, in which case the repair bandwidth is $rb(i) = min(B, b \cdot n_s)$. However, with random chunk placement the repair load may not be even: Some bricks may have more chunks to repair than others. The *bottleneck brick* is the one with the highest repair load. Let H be the number of stripes to be repaired in the bottleneck brick. We call H/n_s the *bottleneck load*, and denote it as l_b. We use a Monte-Carlo simulation to calculate l_b, and the result is shown in Figure 7. The key result is that when n_s is close to N, the bottleneck load could be one order of magnitude larger than $1/n_s$, the load in the ideal case.



Note: The dotted lines represent the ideal cases where there is no bottleneck brick. The Monte-Carlo simulation is basically throwing n_s balls (chunks) into N-1 slots (remaining bricks) and look for the slot with the largest number of balls.

Figure 7. Bottleneck load vs. the stripe number

Given the bottleneck load, we can have the repair time based on the bottleneck load, which is $c \cdot l_b/b$, where c is the amount of data in a brick, and b is the brick bandwidth. Combining this with the repair time calculation in the framework (formula (2)), we have the following repair time formula for the stripe placement scheme:

$$MTTR(i) = T + \max[D(i)/\min(B, b \cdot n_s), c \cdot l_b / b]$$

Plugging in the above formula in the analytical framework, together with $m=n_s \cdot N/k$, we can compute the reliability of the stripe placement scheme.

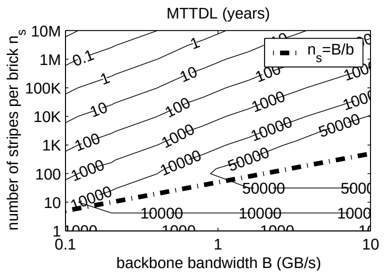
The results of the analysis are shown here as a series of contour plots (Figure 8). In Figure 8(a), we see that when the stripe number increases while fixing the backbone bandwidth (walking up vertically through the contour), *MTTDL* increases first because the repair bandwidth increases when more bricks are involved in repair. After reaching a peak *MTTDL* drops, because the repair bandwidth is restricted by the backbone bandwidth so that repair speed has no further improvement, but the number of possible placement combinations continues to increase as n_s increases. So for each backbone bandwidth value, there is an optimal stripe number to give the best *MTTDL*. From the plot, we can see that the best *MTTDL* values are located along the ridge in the contour plot, and along this ridge, the stripe number n_s increases proportionally to the backbone bandwidth.

Figure 8(b) shows the contour plot of *MTTDL* with different brick bandwidth and stripe numbers. Similar to Figure 8(a), we can see that the optimal *MTTDL* values are also located on the ridge of the plot. However, in this case, the optimal stripe number is reverse-proportional to the brick bandwidth. Figure 8(c) shows that the optimal stripes number does not rely on the system scale.

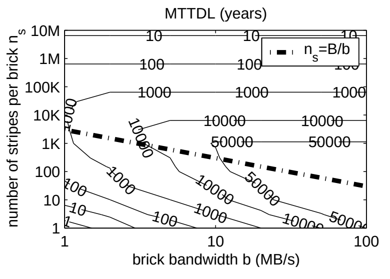
Therefore, from the three plots, the conclusion we reach is that the optimal stripe number n_s should be proportional to the backbone bandwidth B, reverse-proportional to the brick bandwidth b, and not related to the system scale N. Based on this result, we propose that the optimal stripe number n_s can be given by B/b. In the above three plots, the dash-dotted lines correspond to the stripe numbers with B/b. We can see that they are all very close to the ridges, i.e., the best *MTTDL* values.

The formula n_s=B/b is a “guideline” formula. Given a set of system parameters, one can use our analytical framework and some numerical method to find the true optimal n_s, and the result may be a little different from B/b. However, from our analysis, we see that B/b provides near-optimal system reliability. Thus, the optimal chunk size is $c \cdot b/B$ as a function of disk bandwidth, backbone bandwidth and disk capacity.

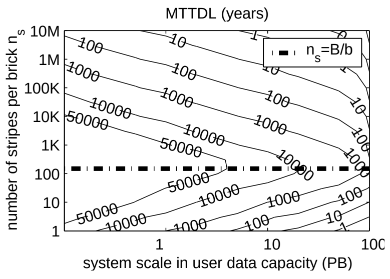
The recommendation of n_s=B/b allows a simple and intuitive explanation. In the ideal situation, n_s pairs of sources and destinations participate in data repair in



(a) Vary with backbone bandwidth. MTTF=1000days, b=20MB/s, c=500GB, k=3, user capacity = 1PB.



(b) Vary with brick bandwidth. MTTF=1000days, B=3GB/s, c=500GB, k=3, user capacity = 1PB.



(c) Vary with system scale. MTTF=1000days, B=3GB/s, b=20MB/s, c=500GB, k=3.

Figure 8. Contour plot for system MTTDL

parallel, and each pair can have maximum bandwidth of *b*. If n_s=*B/b*, then the overall repair bandwidth is $n_s b$=*B*, which means the repair exactly saturates the available network bandwidth. This is the best that one can expect. Therefore n_s=*B/b* provides near optimal reliability.

With n_s=*B/b*, stripe placement provides much better reliability than sequential placement and random placement. As a numerical example, with our typical setting of B=3GB/s, b=20MB/s, *k*=3, *c*=500GB, and total user capacity of 1PB, our optimal stripe placement achieves an *MTTDL* of $9.41 \cdot 10^4$ years, and this does not vary with object size. In contrast, the sequential placement has an *MTTDL* of $7.66 \cdot 10^3$ years, and the random placement has *MTTDL* values worse than sequential placement when the average object size is less than a few tens of megabytes, and its *MTTDL* is only getting close to the optimal value when the average object size is in the gigabyte range, as shown in Figure 5.

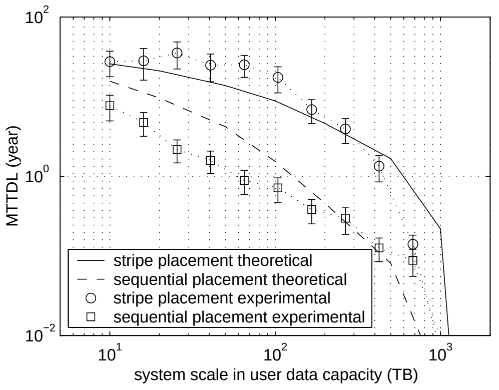
Furthermore, Figure 8 shows that a reasonable range around the optimal stripe number (e.g. within an order of magnitude change) still permits good reliability that is close to the optimal *MTTDL*. This means that the guideline of *B/b* is reasonably robust and may still be applicable even when the network or disks are upgraded over the years.

5.3 Simulation results

To verify our analytical results, we run simulations of a brick storage system to see if the reliability matches with the theoretical prediction and if stripe placement indeed provides better reliability.

The simulation is done in an event-driven model. There are two kinds of events pushing the virtual time forward. One is brick failure events, which are triggered by exponentially distributed and independent brick failures. The other is stripe repair finish events, which are triggered at the time when the first stripe repair session is finished. When any event occurs, simulator re-calculates the repair bandwidth for every remaining stripe repair session based on the brick and backbone bandwidth constraints. All stripes have the same weight when competing for bandwidth.

Figure 9 shows the results of both the simulation and the theoretical analysis on stripe placement and sequential placement. The main results are: (a) stripe placement is better than sequential placement, and (b) for stripe placement, simulation results match well with theoretical results, while for sequential placement, simulation results show much lower *MTTDL* because the theoretical analysis on sequential placement is optimistic.



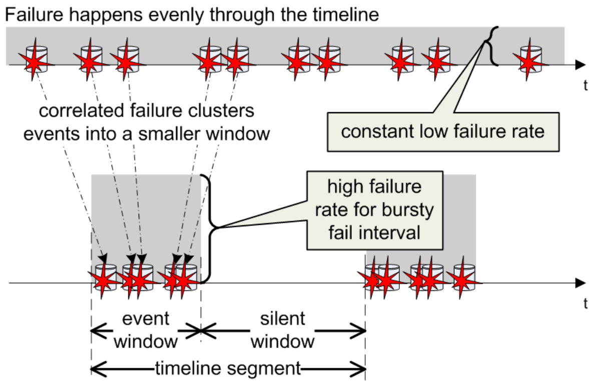
Note: Each simulation result is computed as the average from 50 simulations, with 99% confidence interval shown in the figure. MTTF=30days, B=1GB/s, b=20MB/s, c=500GB, k=3. The artificially small value of MTTF is to shorten the MTTDL of the system so that the simulations can end within a reasonable amount of time.

Figure 9. Simulation result on MTTDL comparing with the theoretical analysis.

For pure random placement, the simulation with large object size is relatively close to sequential placement, while the simulation with small object size is not done because the bandwidth allocation calculation in this case is prohibitively slow.

6. Discussion on Correlated Failures

So far, our analysis assumes that brick failures are independent of each other. However, correlated failures are frequently observed in practice, because bricks are usually from the same manufacturer, and they are operated under the same environment with similar access patterns. The primary difficulty here is how to model correlated failures.



Failure happens evenly through the timeline
t
correlated failure clusters events into a smaller window
constant low failure rate
high failure rate for bursty fail interval
t
event window
silent window
timeline segment

Figure 10: Correlated failure model.

Our assumption is that correlated failures do not change the *MTTF* of each individual brick, because

MTTF is measured in a typical operational environment. However, when looking at a large number of bricks, their failures are correlated in the sense that when one brick failures, more bricks are likely to fail soon. Thus, the overall failure behavior is the clustering of failures each separated by a quiet period with fewer failures.

Accordingly, we convert the failure behavior of the independent failure model into the behavior of correlated failure model as illustrated in Figure 10. Initially, we have a time line spread with independent brick failures. From this time line, for every period of *timeline segment*, we squeeze brick failure events proportionally into a small *event window* and project them onto a new timeline. Then after the event window, there is a silent window the size of which is the difference of timeline segment vs. event window. In the silent window, there is no failure. Therefore, in the new correlated failure timeline, the failure events are more clustered, but the overall failure rates remain the same, which means individual *MTTF* remains the same.

With this model, we can apply the analysis of *MTTDL* easily. First, we define the *correlation degree* r as the proportion of the silent window: r = (silent window length) / (timeline segment length). Thus, r=0 means there is no correlation at all, whereas r=1 means failures are perfectly correlated and a bunch of failures always occur at the same time.

In the correlated timeline, if we drop all the silent windows and concatenate all event windows together, we obtain a *virtual timeline* whose time is compressed to (1-r) proportional to the real timeline. In the virtual timeline failure event rate is 1/(1-r) times of the real timeline, i.e., $MTTF_{virtual} = MTTF \cdot (1-r)$. With $MTTF_{virtual}$ we can compute the data reliability $MTTDL_{virtual}$ in the virtual timeline. Then we map $MTTDL_{virtual}$ back to the real timeline by inflating the time with the ratio 1/(1-r). Thus we get $MTTDL = MTTDL_{virtual}/(1-r)$.

Figure 11 shows the reliability of the system when the correlation degree varies from 0 to 1. We can see that the reliability of the system drops about an order of magnitude from r=0 to r=0.5, and drops even faster with higher correlation degree. In actual systems, it is possible to collect past failure statistic to obtain an estimate on the correlation degree based on the clustering behavior of the failures. Then one can use the above model to calculate the reliability of system, or to estimate the bandwidth of the root switch needed for supporting a certain level of system reliability.

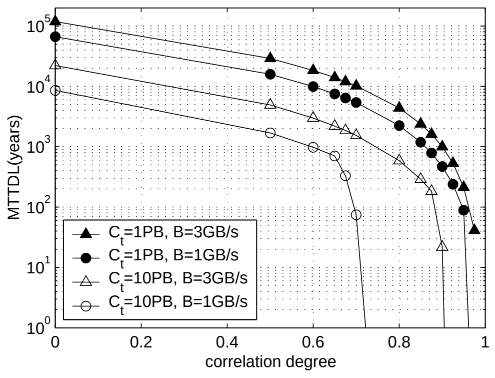


Figure 11: MTTDL when correlation degree varies. MTTF=1000days, b=20MB/s, c=500GB, k=3.

7. Related Work

Reliability is one of the key aspects of storage systems, and it has been studied extensively, especially for disk arrays like RAID systems [15]. Reliability studies on disk arrays investigate the impact of disk organizations on *MTTDL*, and typically use Markov models to study the system with independent and exponentially distributed disk failures, e.g. [2] [20]. Our work can be viewed as an extension of these studies into distributed brick storage systems as a generic object store. The important new addition to previous models is the consideration of bounded network bandwidth to data repair.

Many studies [12][1][8][22][14][18] are on similar brick storage systems, but their focuses are not on the systematic study of data reliability. Many of them do address replica placement issues for various reasons. The sequential placement strategy has been widely used. For example, Petal [12] uses chained declustering [10] mainly as a way to improve load balancing, and many P2P systems such as PAST [17] and CFS [4] use it to simplify management. GFS [8] uses random placement to improve data repair performance, but it does not provide a study on the resulting reliability of the system. In our paper, we study the tradeoff to reliability between sequential and random placement.

Several works have investigated the impact of replica placement on availability, not reliability. In Farsite [1], Douceur and Wattenhofer studies dynamic replica placement strategies that improve the overall availability of files [5][6]. In [18][19], van Renesse and Schneider study DHT-based placement (which is categorized as sequential placement in this paper) and random placement and their effects on the availability of a distributed storage system. They use simulation methods and do not consider the effect of available network bandwidth on data reliability.

We have proposed the stripe placement policy with near optimal configuration parameters. Grouping objects is not a new concept. The 64MB chunk size in GFS is based on its workload and read/write performance considerations, while the work in [18] groups a set of objects into volumes but it is not clear how the number of volumes is determined. This study points out that, from a reliability point of view, the chunk size should be a function of available bandwidth, disk bandwidth and disk capacity.

Many coding schemes, in particular Reed-Solomon coding [16], are used in RAID-like storage systems. We do not incorporate such coding schemes into the analysis of reliability, partly because brick storage systems can typically afford more bricks to support simple replication and avoid the complexity and performance penalty associated with the coding schemes.

Some previous work (e.g. [3]) studies the effect of correlated failures, but we do not find a generic model that can facilitate the analysis of system reliability. Our model is simple enough to incorporate into our framework for reliability study, but its effectiveness needs to be further validated in practice.

8. Concluding Remarks

With an analytical framework that incorporates available network bandwidth consideration, we study the reliability of distributed storage systems with different replica placement schemes. We show that both sequential placement and pure random placement have their drawbacks and propose the stripe placement scheme to achieve near-optimal reliability.

We wish that this study could serve as a guideline for system designers and administrators to determine a number of system parameters when building such brick storage systems, including the root switch bandwidth, the stripe number, the replication degree, etc. We believe that, even though our calculations are based on an idealized framework, the recommendations derived (e.g. using *B/b* as the stripe number) are applicable to many practical situations, because a reasonable range of values around the computed values provide the same level of reliability as shown by our results. Furthermore, after removing the backbone bandwidth constraint, the framework should be able to adapt to wide-area peer-to-peer storage settings.

As the next step, we plan to implement stripe placement in BitVault [21], a data retention platform built with a large number of storage bricks. Future research also includes the study on the reliability with heterogeneous brick components, and the reliability of placement schemes that are aware of network topologies.

References

[1] A. Adya, W. J. Bolosky, M. Castro, et al, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment", in *Proc. of the 5th OSDI*, December 2002.

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

[3] P. Corbett, B. English, A. Goel, et.al., "Row-diagonal parity for double disk failure correction", in *Proc. of 3rd Usenix conference on File and Storage Technologies (FAST'04)*, April 2004.

[4] F. Dabek, M. F. Kaashoek, D. Karger, et al, "Wide-area cooperative storage with CFS", *Proc. of the 18th ACM Symposium on Operating System Principles*, 2001.

[5] 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.

[6] J. R. Douceur and R. P. Wattenhofer, "Optimizing file availability in a secure serverless distributed file system", in *Proc. Of the 20th Symp. on Reliable Distributed Systems*. IEEE, 2001

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

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System", in *Proc. of the 19th ACM Symposium on Operating System Principles*, Oct. 2003

[9] G. A. Gibson, D. F. Nagle, K. Amiri, K., et al. "A Cost-Effective, High-Bandwidth Storage Architecture", ASPLOS, October, 1998.

[10] H-I. Hsiao and D. J. DeWitt, "Chained declustering: a new availability strategy for multiprocessor database machines", Technical Report CS TR 854, University of Wisconsin, Madison, June, 1989.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, et al, "OceanStore: An Architecture for Global-Scale Persistent Storage", ASPLOS 2000.

[12] E. K. Lee and C. A. Thekkath, "Petal: Distributed Virtual Disks", ASPLOS 1996.

[13] Q. Lian, W. Chen, and Z. Zhang, "On the impact of replica placement to the reliability of distributed storage systems", Microsoft Research Technical Report, to appear.

[14] J. MacCormick, N. Murphy, M, Najork, et.al, "Boxwood: Abstractions as the Foundations for Storage Infrastructure", In *Proc. of OSDI'04*, Dec. 2004.

[15] D. A. Patterson, G. Gibson, R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)", In Proc. of the *1988 ACM SIGMOD international conference on Management of data*, 109 - 116, 1988.

[16] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software --- Practice and Experiene*, 27(9):995-1012, Sept. 1997.

[17] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large scale, persistent peer-to-peer storage utility. In *Proc. Of the 18th ACM Symp. On Operating Systems Principles*, October 2001.

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

[19] R. van Renesse, "Efficient Reliable Internet Storage", Workshop on Dependable Distributed Data Management. Oct., 2004.

[20] 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

[21] Z. Zhang, Q. Lian, S. D. Lin, et al, "BitVault: A highly reliable distributed data retention platform", submitted for publication.

[22] Z. Zhang, S. D. Lin, Q. Lian, 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.

Appendix

A. Main symbols used in the paper

symbol		Sample values
MTTDL	Mean time to data loss of the system	1~10^5 years
$MTTDL_{obj}$	Mean time to data loss for a specific object	>10^8 years
k	Replication degree	3
N	Site-wide number of bricks	60~600,000
m	Possible replica placement combinations	*N*~*C*(*N*,*k*)
B	Backbone bandwidth	3GB/s
b	Brick bandwidth	20MB/s
MTTF	Mean time to permanent failure of a brick.	1000 days
c	Average amount of data stored in a brick	500GB
s	Average size for a single object	4KB~10GB
T	Detection delay, from the time a brick crashes to the time other bricks are notified about the failure	10 seconds
n_s	Number of stripes hosted on one brick	*k*~*C*(*N*,*k*-1)
r	Correlation degree of failures	0~1