

Storage Allocation in Unreliable Peer-to-Peer Systems

John A. Chandy

Department of Electrical and Computer Engineering
University of Connecticut
Storrs, CT USA

Abstract

Peer-to-peer systems provide the opportunity to pool large amounts of distributed resources to enable internet-scale applications. However, the participant nodes are highly dynamic and unreliable. Thus, any shared resource such as file objects must incorporate redundancy to be useful. While many studies have proposed heuristics to determine redundancy levels based on object popularity, there has been little work in determining optimal or near-optimal resource allocation based on node reliability. In this paper, we present a strategy for the allocation of objects in the presence of dynamic and unreliable peers. We have built an availability model of peer-to-peer storage systems based on the bimodal and time-dependent availability characteristics of a P2P node. Using this model, we can select the size of a candidate node set for storage allocation and assign storage objects to maximize availability while still maintaining a balanced distribution of objects.

1 Introduction

Peer-to-peer systems provide the opportunity to pool large amounts of distributed resources to enable internet-scale applications. The most well-known collaborative application on these peer-to-peer networks is file sharing - best exemplified by publicly available systems such as Napster, Gnutella [1], Kazaa [2], Overnet [3] amongst others. Using these file sharing systems as inspiration, many global storage systems have used a peer to peer network framework as the underlying infrastructure to support a distributed storage system. These systems are characterized by a global file system sitting atop an underlying peer-to-peer network that provides routing and location functionality. Examples of such global storage systems include OceanStore/Pond [4, 5], PAST [6], FarSite [7], Ivy [8], CFS [9] and Pangaea [10].

Large numbers of nodes in any storage system mean potentially large number of node failures and thus lower avail-

ability of data, where availability is defined as the probability that the data is accessible at any point in time. Data may not be accessible if the storage node is down for any of a number of transient or permanent reasons. These include network failure, computer crashes or simply that the computer was turned off or the network was disconnected as with a dialup connection. As a result, it is important to have some form of redundancy to preserve data availability. When nodes have roughly similar failure rates, it is relatively easy to model overall system reliability and more importantly, the allocation of data objects does not affect data availability. However, peer-to-peer systems have widely varying failure rates, meaning that where objects are placed can have a significant effect on data availability.

In this paper, we present an allocation strategy to maximize data availability for a peer-to-peer based storage system. The algorithm is based on the realization that different nodes in a peer-to-peer system have different failure rates as well as the fact that those failure rates are time dependent. We have developed a procedure that allows nodes to maintain availability statistics on just a small subset of the entire node set and then use that subset to allocate storage fragments. Using an analytical model, we can then determine an optimal size to this node subset so that a desired availability is achieved.

The remainder of the paper is organized as follows. We present related work in the study of reliability and availability of P2P storage. We follow with a presentation of our storage allocation strategy including a derivation of the analytical model. Finally, we close with simulation results that confirm our analytical model and the efficacy of our allocation algorithm.

2 Related Work

Availability in peer-to-peer systems has been investigated by several researchers. Some of these studies have been experimental as they examined actual peer-to-peer systems in order to characterize system uptime. Saroiu et al. [11] crawled the Gnutella network and collected vari-

ous statistics including bandwidth, latency, and availability. They found that only about 20% of the peers have high (>93%) availability and that the median availability was only 7%. Chu et al. [12] did a similar study and observed time-dependent peer availability, i.e. some nodes were turned off at night and were most likely to be on at around 8 pm. Both these studies were limited by the fact that they used IP addresses to identify nodes and thus likely underreported availability because nodes changed IP addresses due to dynamic IP addresses. Using Overnet client IDs, Bhagwan et al. [13] accurately tracked nodes and found that the median availability was 30% instead of the lower number reported by Saroiu. Another reason for the increase may be that Bhagwan’s study was performed in 2003 nearly two years later than Saroiu’s study. Over that two years broadband usage increased significantly and thus increased the availability of always-on Internet connections. Douceur [14] studied various peer-to-peer systems including Napster and Gnutella and found two classes of availability - one class where the peer was on nearly all the time and another class where the peer appeared to be cycled on and off.

Besides experimental studies, others have investigated strategies to address availability by looking at various forms of redundancy. CFS [9] uses simple replication of objects. In order to increase availability, OceanStore [4] uses m/n erasure codes where an object is split into n fragments and distributed to n peers, but only m fragments are required to recover the data. Neither CFS or OceanStore amongst others considered node availability when placing object fragments or replicas. CFS, for example, places replicas on peer nodes that are closest to the publishing node. Van Renesse [15] proposed a modification of the replica placement algorithm by considering the reliability of nodes and placing as many replicas on peer nodes as needed until the desired availability was achieved. However, this approach did not take into the account the available storage space on each peer nodes leading to an unbalanced distribution whereby highly available nodes contain most of the replicas and can become overloaded.

Other replica placement algorithms have also taken object availability into account. Farsite uses replication along with a hill climbing algorithm to progressively improve the file availability by swapping file replicas [16]. Xin et al. [17] expanded on this algorithm by using erasure codes across redundancy groups and considering nodes with cyclic availability behavior. They presume that peers are down for half the day but spread evenly across the globe thus balancing availability throughout the day. This assumption is not true for regional networks and may not even be true for global networks as seen by data collected by Chu, who observed that much of the Gnutella user population is based in the United States. This algorithm also

introduced the idea of candidate sets but didn’t define techniques for determining an optimal size for this set. In this paper, we will define a model to arrive at an optimal candidate set size for a desired availability. Kangasharju [18] developed an integer programming formulation to the object placement problem based on a priori knowledge of full system availability.

3 Storage Allocation Strategy

3.1 Storage Allocation as a Dynamic Programming Problem

Before we discuss the allocation strategy, we formulate the problem as a dynamic programming problem to arrive at a near-optimal solution. The goal is to assign data objects and their replicas to nodes in the peer-to-peer systems such that the availability is maximized. The model is an expansion of that presented in [18]. We assume there are N nodes in the system and there are s_i available storage units on node i . $a_i(t)$ is a time-dependent function describing the availability of node i , i.e. the probability that node i is up at time t . Previous studies [12, 13] have shown that node availability has a clear dependency on time. There are J data objects in the system and for simplicity we assume that each object is equal in size to one storage unit. This is not an unreasonable assumption for block-based storage systems. For generality we can assume that the redundancy method is some form of an erasure code where the object is divided into n fragments with $m(\leq n)$ fragments required for recovery. Other redundancy mechanisms can be expressed using this formulation; e.g. no redundancy is equivalent to the case where $m = n$, RAID5 is equivalent to $m = n - 1$, and simple replication or mirroring is equivalent to $m = 1$.

Since the general m of n redundancy problem can be quite complex, we start with a simpler problem where there is full redundancy or mirroring ($m = 1$). We will revisit the general m of n problem later. The probability, $A_j(t)$ that the data object j is available is:

$$A_j(t) = 1 - \prod_{i=1}^N (1 - a_i(t))^{d_{ij}} \quad (1)$$

d_{ij} is 1 if a replica of object j is on node i and 0 otherwise. If we assume that each object is equally popular we can average across all data objects, and arrive at the overall system data availability given by:

$$A(t) = \frac{1}{J} \sum_{j=1}^J A_j(t) \quad (2)$$

We can assume that $a_i(t)$ is periodic (either daily or weekly), which means that $A(t)$ will also be periodic. We

can, therefore, calculate the average availability, \bar{A} as follows:

$$\bar{A} = \frac{1}{TJ} \int_0^T \sum_{j=1}^J A_j(t) \quad (3)$$

Since the goal is to maximize the overall system data availability, we can set up a dynamic programming problem, where the objective is to choose d_{ij} such that we maximize \bar{A} subject to the following storage availability condition:

$$\sum_{j=1}^J d_{ij} \leq s_i \quad (4)$$

However, since $A(t)$ is time dependent, maximizing \bar{A} is not ideal since a feasible solution may still produce poor data availabilities at certain times of the day. Thus, we also choose a desired minimum availability, \hat{A} , and introduce new conditions with a discretized t as follows:

$$A_j(t) > \hat{A} \quad \forall j, t \quad (5)$$

This dynamic programming problem is solvable by any of a number of techniques. If $a_i(t)$ is constant, the problem is somewhat simplified since zero-one integer programming techniques could be used. However, the solution is dependent on the a priori knowledge of all node availabilities, i.e. we know $a_i(t)$ for all i . In practice, it is difficult for a single node to know the availability probability of all other nodes in a large system. It could be done in a distributed fashion where periodically nodes exchange availability information. More problematic, however, is that once an optimal data allocation is arrived at, it is assumed that we can dynamically reallocate objects to match this allocation. In a live system, though, doing storage reallocation can be very expensive and thus impractical. Therefore, instead of trying to optimize the allocation of all objects across the entire system, we focus our efforts on an algorithm for the allocation of a single object at the point of creation such that overall system availability is maximized. Thus, we can concentrate on a single node that is publishing an object and must now decide where to place the object fragments to maximize object availability with the constraint that each node has limited storage.

3.2 Analytical Model of Storage Allocation at Object Creation

It is in this light that we revisit the original general m of n erasure code allocation problem. Formulating the more general problem as a dynamic programming system is problematic and as shown above even in the simple case, it is impractical. Instead, we restate the goal to be the optimization

of the allocation of a single object j as it is created rather than optimizing the allocation of all objects currently in the system.

We first presume that each node keeps track of the availability of k other nodes in the system. This can be done through active probing or through side effects of natural communication with the node's neighbors. The availability of each node is kept as a vector where each vector entry is a time period. If we accumulate data on a daily basis with an hour granularity, the vector will be 24 entries long with each vector entry representing the availability over an hour time period. The length of the vector is dependent on the expected periodicity of the availability function. In other words, if we expect that the availability function is the same every day, we would accumulate data on a daily basis. On the other hand, the availability function could vary from day to day; e.g. weekend behavior is different from weekday behavior. In such a case, the availability function has weekly periodicity, and we should accumulate data on a weekly basis. In either case, since k will be much less than N , the resources to store the availability vectors should be minimal. These k nodes provide a set of nodes from which we can select n nodes to use for redundancy, and these n nodes should be chosen such that the object availability is maximized. The set of k nodes is similar to the *candidate* set described in [17]. The difference is that we maintain a time-dependent availability vector for each of the k nodes rather than just an overall availability. We also keep track of the available storage at each node so that highly available nodes do not get overloaded.

Since previous studies [13, 14] have shown a bimodal availability characteristic in P2P networks, we place nodes in two availability classes: *always_on* and *cycled*. Describing these classes behaviorally, we call these two classes *servers* and *homePCs*, respectively. The servers are almost always on and typically have larger amounts of available storage. The homePCs, on the other hand are down for large parts of the day and may have less storage available for sharing. We can classify nodes in our availability set based on an examination of the time dependency vectors for each node. When selecting the n nodes from the candidate set, we need to choose between these two classes of nodes - servers and homePCs. To maximize availability, it would seem that we should always choose servers from the candidate set. However, if we always choose servers, the server nodes would run out of storage space. Thus, we choose only g server nodes from the candidate set and the remaining $n - g$ nodes will be homePCs. We will now establish bounds for g dependent on the available storage on a server.

3.2.1 Selection of server nodes from candidate set

If the fraction of total nodes that are servers is f , then there are, on average, fk nodes in the candidate set that are servers. This server percentage is a global value and is assumed to be known by each node a priori. The server percentage can be calculated by background processes that periodically query system availability. If each object fragment is of unit size, $\frac{g}{fk}$ represents the average amount of storage used on the servers. Likewise, there are $k - fk$ homePCs in the candidate set, and of those only $n - g$ nodes are selected to be the redundancy set. The average amount of storage used per homePC is $\frac{n-g}{k-fk}$. This establishes the ratio of used storage between a server node and a homePC node as $\frac{g(1-f)}{f(n-g)}$. Now, if we limit this ratio to r , i.e. we say that each server has r times the available storage of a homePC, then we can arrive at the following inequality. r is a measured quantity based on the assumption that servers will have more storage than a homePC.

$$r \geq \frac{g(1-f)}{f(n-g)} \quad (6)$$

which leads to

$$g \leq \frac{rfn}{1-f+fr} \quad (7)$$

Eq. 7 establishes a limit on g determined primarily by the available storage on the servers. If $r = 1$, i.e. the servers have equal storage to a homePC, then $g \leq fn$. As r goes up, g goes up as well, meaning we can select more server nodes from the candidate set. Since more server nodes selected from the candidate set means higher availability, it makes sense to maximize g by trying to select server nodes from the candidate set until we hit the limit, $\frac{rfn}{1-f+fr}$. We will call this limit \hat{g} .

Therefore, when we allocate n fragments of a storage object, we will assign no more than \hat{g} of the fragments to server nodes¹. However, on average, there are only fk server nodes in the candidate set. Thus, selecting a large enough k is key to ensuring that there are \hat{g} nodes available in the candidate set. However, we do not want an arbitrarily large k since that increases the number of nodes that must be probed for availability.

3.2.2 Selection of candidate set size

Thus, we must determine a minimal \hat{k} that is still sufficiently large enough to provide an acceptable availability for a data object. As before, we will derive an expression

¹If \hat{g} is non-integral, particularly if it is less than 1, we probabilistically allocate the server nodes. For example, if $f = 0.2$, $r = 1$, and $n = 1$, then $\hat{g} = 0.2$. Thus, when allocating fragments, we pick a server node only 20% of the time. For the purposes of this discussion, however, we will assume that \hat{g} is integral.

for the expected availability. Assuming bimodal characteristics, server nodes have 100% availability and homePC nodes have a time dependent availability equal to $a_i(t)$. We start by stating the availability as:

$$A(t) = \sum_{g=0}^{\hat{g}-1} p_f(k, g) \alpha(n, m, g, t) + \sum_{g=\hat{g}}^k p_f(k, g) \alpha(n, m, \hat{g}, t) \quad (8)$$

where $p_f(k, g)$ is the probability that there are exactly g server nodes in a candidate set of size k and $\alpha(n, m, g, t)$ is the availability of the data object given that we have found g server nodes in the candidate set. Since g can be no larger than \hat{g} , the availability expression has two parts for when g is less than \hat{g} and for when g is equal to or greater than \hat{g} .

p_f is defined as follows:

$$p_f(k, g) = \binom{k}{g} f^g (1-f)^{k-g} \quad (9)$$

To derive α , we assume for simplicity that the homePC nodes are equally geographically distributed and that they share a common time dependent availability function. We will revisit this assumption in Section 3.3. With this simplification, we can remove the time dependency and calculate an average homePC node availability as shown in Eq. 10.

$$\bar{a} = \frac{1}{T} \int_0^T a(t) \quad (10)$$

We can derive α by noting that in order to recover the data object, we must find m fragments of the n data object fragments. Since g of those fragments were assigned to server nodes, we can assume that those fragments are recoverable because the server nodes are always available. The remaining $n - g$ fragments have been assigned to homePC nodes from which we have to find $m - g$ available nodes. Using \bar{a} as the average homePC availability and a construction similar to Eq. 9, we arrive at:

$$\alpha(n, m, g) = \sum_{\mu=m-g}^{n-g} \binom{n-g}{\mu} \bar{a}^\mu (1-\bar{a})^{n-g-\mu} \quad (11)$$

$$= \sum_{\mu=m-g}^{n-g} p_{\bar{a}}(n-g, \mu) \quad (12)$$

Note that α is not dependent on k because $n - g$ is always less than k and as such we are always guaranteed to find $n - g$ homePC nodes to assign the remaining fragments. Putting Eqs. 8, 9 and 12 together, we arrive at the following objective for the overall availability:

$$\hat{A} \leq \sum_{g=0}^{\hat{g}-1} \left[\binom{k}{g} f^g (1-f)^{k-g} \left(\sum_{\mu=m-g}^{n-g} p_{\bar{a}}(n-g, \mu) \right) \right] + \sum_{g=\hat{g}}^k \left[\binom{k}{g} f^g (1-f)^{k-g} \left(\sum_{\mu=m-\hat{g}}^{n-\hat{g}} p_{\bar{a}}(n-\hat{g}, \mu) \right) \right] \quad (13)$$

We can use Eq. 13 to select a k such that our objective is met. Since there is no closed form expression for k , we must use optimization methods to determine \hat{k} . However, since A is strictly increasing with respect to k , finding \hat{k} can be done fairly efficiently. Table 1 shows values for \hat{k} for a variety of redundancy configurations, including no replication, RAID5, mirroring, and a 32/16 erasure code. The availabilities are expressed in units of “nines”, which is defined as $-\log_{10}(1-A)$. It is a measure of the number of leading “nines” in the availability fraction. For example, an availability of 0.999 represents 3 nines. We show the desired availability in the fifth column and the predicted availability in the last column.

As can be seen from the table, in some cases, it is not possible to achieve the desired availability. For example, with no replication or with RAID5, the availability converges to around 0.3 nines (or $A = 0.5$), regardless of how large k is. With no replication, $\hat{g} = 0.5$, which corresponds to an integral value of 0. Thus, no fragments are assigned to server nodes, leaving the total availability equal to the homePC availability. In the table, also note the impact that r has on availability. It is not possible to get 2 nines of availability with an erasure code and $r = 1$. The maximum achievable availability of 1.07 is determined primarily by the fact that \hat{g} is limited by Eq. 7. In other words, the availability is limited because there are not enough servers to allocate fragments to. Increasing \hat{g} past the limit imposed by Eq. 7 will cause the servers storage space to become over-allocated. However, if r is larger, we can get the desired availability, since as r goes up, the available storage on servers goes up meaning that we can assign more fragments to server nodes. Assuming that r is large enough, we can then increase k to get significant improvements in availability as seen in the last three rows of the table.

Note, that we have made an assumption that there is a difference in storage capacity between servers and homePCs, i.e. $r > 1$. If these servers are commercial entities, that assumption is likely. However, if the server nodes are simply homePCs that are on all the time, we may not find a correlation between servers and larger storage space. At this point, there have not been any studies that have measured storage space on nodes in a peer-to-peer system; thus, we can not determine a value for r . Before this algorithm

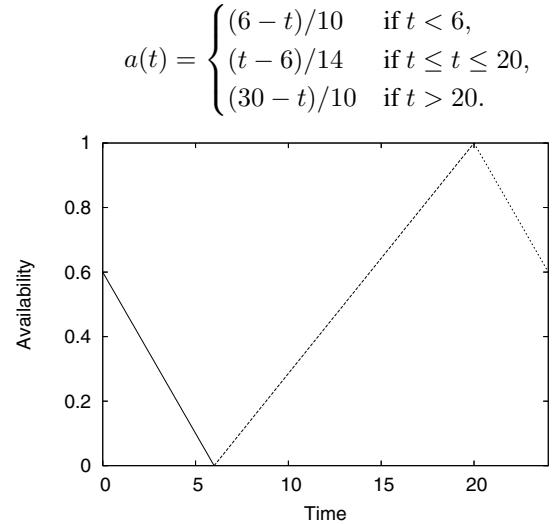


Figure 1. HomePC availability function

can be implemented, further study on typical values of r are needed. We could set r a priori, based on measurements collected from p2p networks or alter r dynamically through monitoring of the p2p network. If we find through these studies that there is no correlation between storage space and servers, then we would set r to 1. In that case, it becomes clear that achieving high availabilities is very difficult. The primary determinant of the availability, \hat{g} , is equal to fn , and f in most peer-to-peer systems is very low as discussed above. Thus, the need to prudently allocate fragments to the servers becomes very important in order to arrive at somewhat reasonable availabilities.

3.2.3 HomePC allocation

To this point, we have not discussed the allocation of storage fragments to the homePC nodes. In the previous section, we developed a model that assumed an average availability, \bar{a} , for the homePC nodes (See Eq. 10). However, this assumption is true only if the data fragments are allocated geographically across the homePC nodes. Therefore, the allocation strategy for homePC nodes must take this into account by ensuring that the nodes are geographically distributed, so that at any particular time there will be some subset of nodes available. We first define a time dependent availability function for the homePCs as shown in Fig. 1. It is an approximation of the data collected by Chu et al. [12]. t is a variable representing the hour in the day and as can be seen, the availability peaks at 8pm. The average availability is 50%.

The goal is to ensure that we select $n-g$ homePC nodes so that they are equally distributed geographically. For the most part, $n-g$ will typically be less than 24. For example,

	n	m	r	$-\log_{10}(\hat{A})$	$\lfloor \hat{g} \rfloor$ (Eq. 7)	\hat{k}	$-\log_{10}(1 - A)$
No replication	1	1	4	1	0	1	0.30
RAID5	5	4	4	1	2	49	0.30
Mirroring	2	1	4	2	1	15	2.06
Erasure code	32	16	1	1	6	39	1.01
Erasure code	32	16	1	2	6	128	1.07
Erasure code	32	16	2	2	10	74	2.01
Erasure code	32	16	2	3	10	212	2.07
Erasure code	32	16	4	3	16	84	3.04
Erasure code	32	16	4	4	16	103	4.00
Erasure code	32	16	4	6	16	138	6.01

Table 1. Candidate size (\hat{k}) for various data redundancy choices and server storage choices ($f = 0.2, \bar{a} = 0.5$).

if we look at the data from Table 1, $n - g$ ranges from 1 to 26. Thus, it is unlikely that we will be able to find in the candidate set one node from each of the 24 time zones. Therefore, we will use a much larger granularity of 4 time zone groups. In other words, each group will comprise of 6 time zones. When selecting nodes for allocation, we will greedily select nodes to balance the groups. For example, when $n - g$ is equal to 16, we will try and select 4 nodes from each time zone group. If the available nodes are not geographically balanced, i.e. there are more nodes from a particular region as was observed by Chu et al. in Gnutella, then it may be impossible to balance the nodes as the rare nodes from outside the overrepresented region get full.

3.3 Storage Allocation Algorithm

Using the model described in the previous section, we can arrive at a maximum value for g and a minimum size for k for a desired availability. The \hat{k} values can be calculated before hand for a variety of desired availabilities and expected redundancy and storage configurations as was done in Table 1. Each node will then monitor the availabilities of \hat{k} nodes in its candidate set. Depending on the size of \hat{k} and the particular routing protocol being used, the candidate set may or may not be a subset of the neighbor set used in routing. Using these \hat{g} and \hat{k} parameters, we can devise an algorithm for allocating storage fragments to server nodes as shown in Figure 2.

4 Simulation Results

We simulated the overall system availability using a set of 16384 nodes with bimodal behavior. Server nodes were assumed to have an availability of 0.99. HomePC nodes have an availability function that is shown in Figure 1. We

```

node_count=0
# allocate up to  $\hat{g}$  servers
for node  $i$  in candidate set of size  $\hat{k}$ 
  if  $i$  is a server and it has space then
    allocate fragment to node  $i$ 
    node_count++
    if ( node_count =  $\hat{g}$  )
      exit loop
    end if
  end if
end for

# allocate remaining fragments to homePCs
for node  $i$  in candidate set of size  $\hat{k}$ 
  if  $i$  is a homePC and it has space then
    tzgroupno = time zone group of  $i$ 
    add node  $i$  to tzgroup[tzgroupno] set
  end if
end for
while ( node_count <  $n$  )
  if ( tzgroup[tzgroupno] is not empty ) then
     $i$  = remove node from tzgroup[tzgroupno++]
    allocate fragment to node  $i$ 
    node_count++
  end if
end while

```

Figure 2. Storage allocation algorithm

The storage algorithm is consisted of two phases. In the first phase, we allocate fragments to \hat{g} servers. In the second phase, the remaining fragments are allocated to homePCs. When allocating fragments to homePCs, we first classify the homePCs by their time zone group. Then, we iterate over the time zone groups to allocate fragments in a round robin fashion, so that the fragments are allocated uniformly across the time zones.

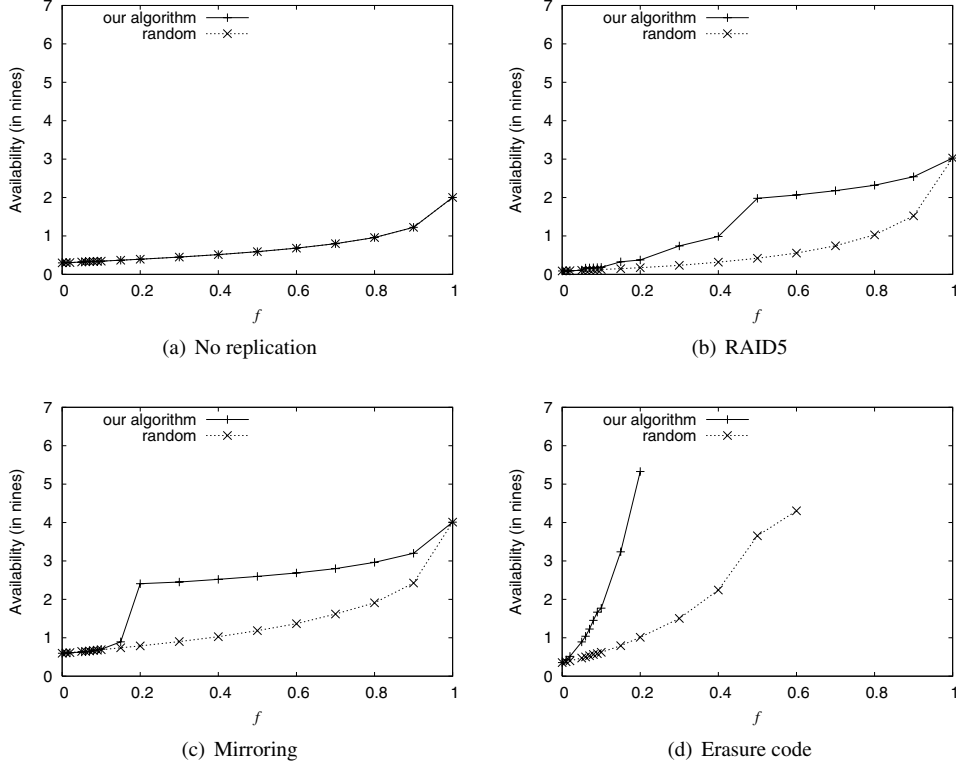


Figure 3. Availability vs. f (world distribution, $r = 4, \hat{k} = 256$).

evaluated the effect on availability for a number of parameters including f , and \hat{k} . We also evaluate the effect of our allocation algorithm vs. a random algorithm that does not take node availability into account. The simulation was run for 3 years in simulated time with requests made at an average 1 per second making roughly 100,000,000 total requests. Availability was measured as the percentage of requests that could be satisfied. We looked at four different redundancy configurations: no redundancy ($n = 2, m = 1$), RAID5 ($n = 5, m = 4$), mirroring ($n = 2, m = 1$), and an erasure code ($n = 32, m = 16$). We also assumed that the storage ratio, r , is equal to 4.

4.1 Effect of storage server percentage (f)

Figure 3 shows the effect of f on availability for four redundancy different configurations. The availability is shown in terms of nines of availability, i.e. $-\log(1 - A)$. As the percentage of nodes that are servers increases, the availability increases since there are more “always on” nodes in the system. Note that data points for high values of f are not shown in Figure 3(d) since the simulations showed no failures during the run, indicating 100% availability or infinite nines.

The graphs also compare our placement algorithm with

a random placement algorithm. When there is no replication, as in Figure 3(a), there is no advantage to our algorithm since our placement algorithm only affects replicas and not the primary copy of the data. For the other redundancy configurations, however, our algorithm performs better, particularly for mid range values of f . It is clear that placing objects with an understanding of the node availability can have a significant effect on availability - in some cases several orders of magnitude. In studies by Bhagwan et al., they observed that the server percentage, f is typically around 20%. It is in this range that the advantage of our placement algorithm is clear. For example, we see that random placement achieves availabilities less than 90% at $f = 0.2$ for all the redundancy configurations. This level of availability is not acceptable for applications that demand high availability. However, the use of our algorithm improves the availabilities to over 99% for mirroring and to over 5 nines when erasure codes are used. This level of availability makes P2P storage acceptable for high availability applications. At higher values of f , while our algorithm is still significantly better, the advantage may not be as critical because random placement may provide sufficiently high availabilities, particularly when erasure codes are used. However, high values of f are not to be expected in general P2P networks as Bhagwan et al. demonstrated.

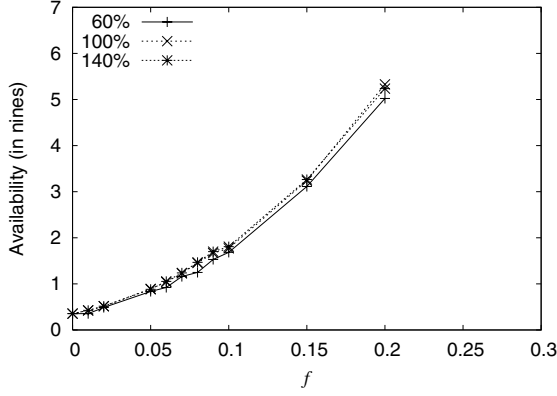


Figure 4. Sensitivity of f (world distribution, $r = 4, \hat{k} = 256, n = 32, m = 16$).

The knees in the curves at $f = 0.5$ in Figure 3(b) and at $f = 0.2$ in Figure 3(c) correspond to the point where f is large enough that \hat{g} becomes larger than m . At this point, the maximum number of servers in our redundancy group is more than are required to recover the data. When this occurs, we no longer need to depend on low-reliability homePC nodes since we will have at least m servers in the redundancy group to recover the data from.

We also wish to examine whether a difference in the actual f and the pre-measured f that we use in our model affects the observed availability. Figure 4 shows the sensitivity of the methodology to f . The x-axis is the f used in the model to derive \hat{g} , and three plots are shown where the actual f in the simulated system is 60%, 100%, and 140% of the model f . As can be seen, there is very little effect on the availability. However, if the f used in the model does not reflect reality, the servers can become unbalanced as they will contain less or more fragments than they should accommodate. Our simulations show that when the actual server percentage is 60% of the f used in the model, servers contain 63% more fragment storage than they should. At 140%, servers contain 30% less storage than it should. However, if we can determine f with an error of less than 20%, the imbalance can be maintained at less than 20% as well.

4.2 Effect of candidate set size (k)

Figure 5 graphs the availability vs. the candidate set size. Along with the simulation points, the curves corresponding to Eq. 13 are shown for the mirror and erasure code configurations. As can be seen, the simulations and the analytical model curves correspond very well until we hit a knee at which point the simulation shows virtually no improvement in availability as k increases. This difference occurs because the analytical models assume that the servers never

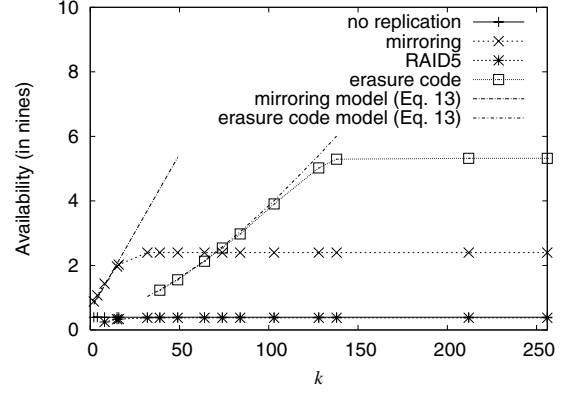


Figure 5. Availability vs. k (world distribution, $f = .2, r = 4$).

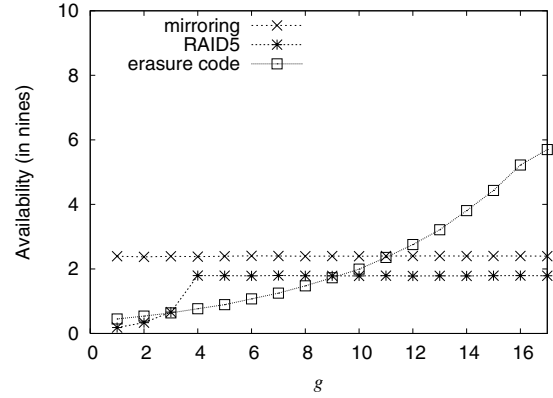


Figure 6. Availability vs. \hat{g} (world distribution, $f = .2, r = 4, \hat{k} = 256$).

fail while the simulation assumes that the server nodes are only 99% available. The knee point occurs when the availability reaches the maximum possible availability with 99% availability servers and that can be calculated as 99.5% (2.3 nines) for the mirrored case and 99.9994% (5.2 nines) for the erasure code case.

We can also use this graph to confirm the Table 1 choices for \hat{k} . The graph shows the points where \hat{k} equals 15 for mirroring and its correspondence to the 2 nines level of availability. Likewise, points where \hat{k} equals 84, 103, and 138 correspond to the 3, 4, and 5 nines level of availability for erasure code. These values correspond to the values predicted in Table 1.

4.3 Effect of max server count (\hat{g})

Figure 6 shows the effect of the max server count (\hat{g}) on availability. In these simulations, \hat{g} is set as a simula-

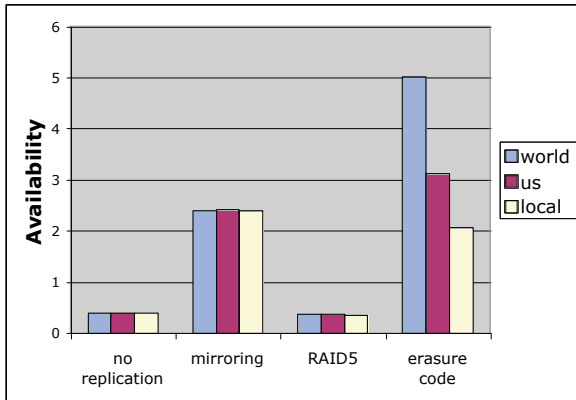


Figure 7. Availability vs. geographical distribution ($f = .2, r = 4, \hat{k} = 256$).

tion parameter, rather than being determined automatically by Eq. 7 as in the other simulations. In general, as \hat{g} goes up, the availability goes up. This is because the number of servers in the redundancy group goes up. However, as we mentioned in the previous discussion on candidate set size, the availability is limited because the simulated servers have an availability of only 99%. There is a knee in the RAID5 curve visible at $\hat{g} = 4$ which corresponds to the knee at $f = 0.5$ in Figure 3(b). The knee is not visible in the mirroring curve because the corresponding knee point is at $\hat{g} = 1$.

4.4 Effect of geographical distribution

Figure 7 shows the effect of geographical distribution on the availability characteristics. We examined three different distributions - *world* where the nodes are distributed equally across the time zones, *us* where the nodes are concentrated within the four US time zones, and *local* where all the nodes are located within the same time zone. For small redundancy groups such as mirroring ($n = 2$) or RAID5 ($n = 5$), the geographical distribution does not have much of an effect. This is because with small redundancy groups it is easier to map most of the nodes to servers, and the availability of servers is not affected by the geographic location. For the erasure code redundancy configuration, the redundancy group is much larger ($n = 32$), and as such a greater percentage of the nodes are homePC nodes which are affected by the time-dependent and thus geography dependent availability function. The *world* distribution performs best because it is easier to find an available node when you can search the entire globe. We are basically shifting and overlapping the time-dependent availability curve from Figure 1 to arrive at a nearly uniform overall availability with no time dependency. The *us* distribution suffers because we only use 4 time zones to look for nodes that are up. The *lo-*

cal distribution is the worst because all the nodes are in the same time zone and thus provides no opportunity to overlap time dependent availabilities meaning that there are parts of the day where there are no nodes that are up at all.

5 Conclusions

We have presented a methodology for storage allocation in an unreliable peer-to-peer network. The key contributions are the use of time dependent availability vectors and a bimodal availability distribution. The use of an analytical model helps us derive metrics for optimal candidate set size as well as the number of servers to which to allocate fragments. The model also helps us keep the storage allocation balanced so that highly reliable server nodes are not overloaded and thus exhausted of storage space. The algorithm shows significant improvement in the availability of data in a peer-to-peer storage system. In further work, we will update the allocation algorithm to take into account network locality. Doing so would ensure a geographic distribution of storage fragments, so that data is accessible in as few network hops as possible. We will also implement the algorithm in an existing P2P system in order to quantify the impact, if any, on performance.

References

- [1] Gnutella. <http://www.gnutella.com>.
- [2] Sharman Networks. <http://www.kazaa.com>.
- [3] Overnet. <http://www.overnet.com>.
- [4] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An architecture for global-scale persistent storage," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 190–201, Nov. 2000.
- [5] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: the OceanStore prototype," in *Proceedings of USENIX Conference on File and Storage Technologies*, pp. 1–14, Mar. 2003.
- [6] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," in *Proceedings of Workshop on Hot Topics in Operating Systems*, pp. 75–80, May 2001.
- [7] W. Bolosky, J. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs," in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 34–43, June 2000.
- [8] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *Proceedings of the Symposium on Operating Systems Design and Implementation*, Dec. 2002.

- [9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings of the Symposium on Operating System Principles*, pp. 202–215, 2001.
- [10] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam, "Taming aggressive replication in the Pangaea wide-area file system," in *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2002.
- [11] S. Saroiu, P. K. Gummadi, and S. D. Tribble, "A measurement study of peer-to-peer file sharing systems," in *Proceedings of MMCN*, Jan. 2002.
- [12] J. Chu, K. Labonte, and B. N. Levine, "Availability and locality measurements of peer-to-peer file systems," in *Proceedings of ITCOM: Scalability and Traffic Control in IP Networks II Conferences*, July 2002.
- [13] R. Bhagwan, S. Savage, and G. M. Voelker, "Understanding availability," in *Proceedings of the International Workshop on Peer-to-Peer Systems*, 2003.
- [14] J. R. Douceur, "Is remote host availability governed by a universal law?," *Performance Evaluation Review*, vol. 31, pp. 25–29, Dec. 2003.
- [15] R. van Renesse, "Efficient reliable internet storage," in *Proceedings of the Workshop on Dependable Distributed Data Management*, Oct. 2004.
- [16] J. R. Douceur and R. P. Wattenhofer, "Competitive hill-climbing strategies for replica placement in a distributed file system," in *Proceedings of International Symposium on Distributed Computing*, Oct. 2001.
- [17] Q. Xin, T. J. E. Schwarz, and E. L. Miller, "Availability in peer-to-peer global storage systems," in *Workshop on Distributed Data and Structures*, July 2004.
- [18] J. Kangasharju, *Internet Content Distribution*. PhD thesis, Department of Computer Science, University of Nice, Nice, France, Apr. 2002.