Efficient Reliable Internet Storage*

Robbert van Renesse

Dept. of Computer Science, Cornell University
rvr@cs.cornell.edu

Abstract

This position paper presents a new design for an Internet-wide peer-to-peer storage facility. The design is intended to reduce the required replication significantly without loss of availability. Two techniques are proposed. First, aggressive use of parallel recovery made possible by placing blocks randomly, rather than in a DHT-based fashion. Second, tracking of individual nodes availabilities, so that it becomes unnecessary to use worst case availability assumptions. The design uses a full membership DHT, and proposes a protocol for replica management.

1 Introduction

In recent years many Internet-wide storage facilities have been proposed that are based on peer-to-peer organizations of personal computers attached to the Internet. These peers organize themselves into a global network overlay and each takes on part of the responsibility of storage. These peers are unreliable—they may regularly detach from the network, or be badly connected in the first place—so aggressive replication is necessary to make sure all stored objects are available at all times. Replication factors of 16 or 32 or more are often suggested in order to deal with this "churn" (frequent joining and departure) problem. One reason that contributes to the high replication factor is that these systems ignore the fact that the availability of these peers is highly variable, and thus the worst case is assumed in which all peers are highly unreliable.

High replication factors result in ineffective use of storage resources, particularly if most objects are rarely accessed. For reading popular objects, the high replication factor can be advantageous for load balancing, but this should really be accomplished by caching.

In this position paper, we propose a different way of building a storage facility that uses much less replication, but maintains high availability by using fast recovery to restore the replication factor quickly in the case of failures [1]. We also propose to track the reliability of each machine and make the replication factor of individual objects dependent on the reliability of the machines on which the replicas are stored, further reducing the amount of replication. How to maintain strong consistency in such a system is outside the scope of this paper.

In our storage architecture, we assume that storage is organized in fixed size blocks. Each block b has a unique identifier $b.id$. We also assume that each peer p has a unique identifier $p.id$, and that identifiers are chosen uniformly at random from a large identifier space.

We start by showing that replica placement is important to reducing recovery time in Section 2. In Section 3 we show how reduced recovery time improves reliability. Section 4 shows a technique for pseudo-random placement that makes fast recovery possible while still allowing replicas to be located efficiently. In Section 5 we present a technique for tracking the reliability of individual peers. Finally, Section 6 presents the full replication protocol. We discuss related work in Section 7, and conclude in Section 8.

2 Replica Placement

One requirement of replica placement is that, given a block b, you can find peers $p_0, p_1, ...$ that store this block. Unstructured p2p networks such as Gnutella (www.gnutella.com) use a location query flooding approach, one that is notoriously inefficient and ineffective for rare objects. Because we are interested in lookup efficiency and availability, we now take a look at how so-called structured p2p networks such as CFS [2] and PAST [3] address this problem.

Structured p2p networks build a routing overlay that allows messages addressed to a key k to be delivered to the peer p whose identifier is *closest* to k (by some metric). If the replication factor is r, a block b is replicated on the r peers with identifiers closest to $b.id$, and the routing overlay is used to place and locate replicas. This leads to a grouping of blocks, because blocks with nearby identifiers are stored on the same peers.

*This work was funded in part by DARPA/AFRL-IFGA grant F30602-99-1-0532, and by the AFRL/Cornell Information Assurance Institute.

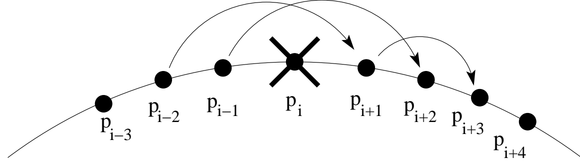


Figure 1: Parallel recovery for DHT-based replica placement. Here the replication factor $r = 3$. Each arrow represents a copy of blocks stored by p_i that the destination does not already have.

If a peer p_i becomes unavailable, then the nearby peers will download replicas in order to restore the replication factor of the blocks that were stored on p_i. If the replication factor is r, this can most efficiently be done in r parallel transfers (see Figure 1). p_{i-r+1} copies to p_{i+1} those blocks that it had in common with p_i, p_{i-r+2} copies to p_{i+2} those blocks that it had in common with p_i but not with p_{i-r+1}, and so on. Finally, p_{i+1} copies the remaining blocks to p_{i+r}.

The amount of time required to do such recovery can be significant. Say that on average each peer contributes 4096 blocks of 64 kilobytes. A typical effective cable upload speed is no more than 10 kilobytes per second. If the replication factor is 16, then we can do 16 parallel copies, which means that it takes approximately 4096 * 64 / 16 / 10 seconds, which is approximately half an hour. This is also the window of vulnerability. If during this window other nearby peers become unavailable, recovery will slow further because fewer parallel transfers will be possible, growing the window. Although this calculation is independent of the replication factor, the window tends to depend on it: if only one of these peers has a slow Internet connection, the window and hence the vulnerability will grow.

If, instead of storing replicas on consecutive peers in the identifier space, the replicas of blocks were randomly scattered across the p2p network overlay, recovery could be done much faster. Assuming the number of peers is much larger than the number of blocks per peer (4096 in this case), as many as 4096 parallel transfers could be started simply by partnering peers storing replicas of each of the 4096 blocks stored on the failed peer with 4096 new peers. In this case, that would bring the recovery time back to 64 / 10 = 6.4 seconds! Also, this would spread the load of recovery much more evenly across all the peers. The Google File System [4] uses this approach, albeit on a small scale within a storage cluster.

The problem with this approach is that if replicas of blocks are scattered randomly across the p2p network overlay, then it will be hard to locate them. One could envision using a directory, but such a directory would add a level of indirection and only push the problem one step away. Instead, we will propose to place the blocks pseudo-randomly with the help of a hash function. First, however, we discuss what effect small recovery time has on availability.

3 Effect of Recovery Time

In Figure 2, produced from simulation data in [5], we show the Mean Time Between Unavailability (MTBU) as a function of the number of peers for various replication/repair strategies and replication factors. Unavailability refers here to the unavailability of any block. In each of these cases, each peer contributes 100 blocks of data to the p2p network. The strategies are:

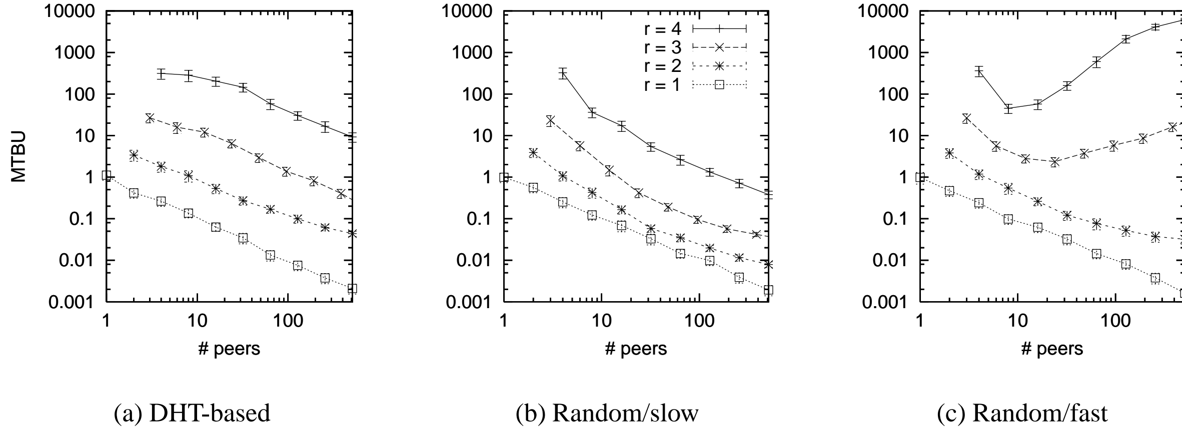
- **DHT-based** replica placement is done using a structured DHT. The maximum number of parallel replica copies is r.
- **random/slow** placement is random, but the maximum number of parallel copies is limited to r.
- **random/fast** placement is random, and maximum parallelism is exploited for copying replicas.

To summarize [5], a short recovery time is highly important to providing good availability. The initial dip of MTBU in the **random/fast** strategy is caused by the fact that little parallelism is possible, combined with a high probability that r simultaneous failures will affect the availability of some block. Comparing **DHT-based** with **random/slow** placement shows that random placement adversely affects the MTBU, because in the **DHT-based** strategy only r simultaneous failures of “consecutive peers” (neighbors in the identifier space) leads to unavailability. Nonetheless, with more aggressive parallel recovery, random placement significantly outperforms DHT-based placement.

4 Pseudo-Random Placement

It is possible to place replicas of a block b pseudo-randomly in the identifier space using a DHT as follows: use locations $HASH(b.id, 1)...HASH(b.id, r)$. However, this strategy does not suit our purpose: if a peer p crashes, all the replicas of blocks near $p.id$ would still have to be copied among the r neighbors of p, leading to no improvement in parallel recovery.

Instead, we propose to use a full membership DHT, as discussed in [6]. In such a DHT, every peer knows every other peer, and thus all routing can be accomplished in a single hop. The authors argue that multi-hop routing DHTs are only useful in fairly unusual situations. They



(a) DHT-based (b) Random/slow (c) Random/fast

Figure 2: The MTBU and 99% confidence intervals as a function of the number of peers and replication factor for three different placement strategies: (a) DHT-based placement with maximum possible parallel recovery; (b) random placement, but with parallel recovery limited to the same degree as is possible with DHTs; (c) random placement with maximum possible parallel recovery.

also present a simply full membership DHT, in which peers essentially gossip their membership lists with each other (similar to the approach in [7]).

We construct a pseudo-random placement as follows. For each block b, assign a ranking to each peer p (faulty or not) by calculating $HASH(b.id, p.id)$. Now store the replicas of b on the peers with the r lowest rankings.

Given that every peer knows the identity and location of every other peer, this allows any peer to locate all replicas of any block b. If a peer p were to become unavailable, then each block b that it stored would now be need to be placed on the peer with the $r + 1^{st}$ lowest ranking for that block. As the ranking of each peer is very different for each block, the resulting peers that need to receive copies of p's blocks are pseudo-randomly distributed in the identifier space, thus satisfying our needs.

In order to make aggressive recovery possible, we need to

1. detect the failure of a peer p rapidly;
2. determine what blocks were stored on p;
3. determine the peers that now need to receive copies of those blocks;
4. select peers that have replicas of those blocks to send the copies.

While [6] offers a full membership service, it detects and recovers from persistent failures of peers very slowly (on the order of days). It seems unlikely each peer could maintain recent information on the availability of all other peers: Every failure would need to be broadcast, and as the rate of failures is expected to rise linearly with the number of peers, doing so is unlikely to be scalable.

A different approach is necessary, which we will now present. At the same time we will tackle the reliability heterogeneity problem: we would like to take this into account when determining how many replicas to create.

5 Extended Membership

The main idea in the suggested approach is to extend the full membership at each peer with reliability information about each peer. We will assume that membership will only grow—failed members are not removed, in the expectation that eventually they will recover. In practice, long time failed members will have to be removed, but we do not think that doing so will present complications. We have each peer monitor its r nearest neighbors in the identifier space, and assign a score to each neighbor based on its measurements. For example, the peers could be pinging each other at regular intervals, and the score could be a smoothed average of the fraction of pings to which responses have been received. That way, the score would represent the availability as a number between 0 and 1.

The membership information, slowly but reliably updated through gossip, would contain for each peer q the following information:

- $q.id$: the identifier of the peer;
- $q.address$: the peer's network address;

- $q.scores[r]$: q's scores of its r nearest neighbors in the membership list

Any peer p can calculate a *reliability score* $\mathcal{R}(q)$ for any peer q by combining the scores for q from the r nearest neighbors of q. For example, $\mathcal{R}$ could be calculated by taking the median of the neighbors' scores.

A block b would now be placed on the k lowest ranked peers $p_1...p_k$ for that block, with k the lowest number so that the combined reliability scores for the k peers exceeds the required reliability. For example, say that a certain minimum availability is specified in terms of "the number of 9's". Using the availabilities calculated for each peer, it is possible to find a k so that the probability that at least one of the k lowest ranked hosts is available exceeds the required availability.

Note that our protocol does not guarantee that all peers agree on the reliabilities of the peers, and therefore the peers will not agree exactly on how many peers should be used for replication of a block. However, we expect that there will be a sufficiently large overlap, and the location protocol of the next section will be immune to such inconsistencies.

6 Aggressive Recovery

Let us return now to solving the problems to aggressive recovery. The first problem is fast detection of failures. We can solve this problem because each peer p is closely monitored by its r neighbors. Each of these r neighbors, besides determining a score for p, can also determine whether p is currently available or not. We allow for false suspicions—doing so is not a big problem as our recovery mechanism is cheap and fast. Thus for simplicity, if any neighbor suspects p, then we can invoke recovery at that time.

The next problem is determining which blocks were stored on p. We need additional mechanism for this as well, as in our approach so far there is nobody who knows except p itself. We propose that each peer p notifies each of its r neighbors of the identifiers of each of the blocks it stores. As a result, each peer q knows the identifiers of the blocks stored by each of its r neighbors. Thus, if q suspects p, it knows which blocks were stored by p.

Next, q can determine, for each block b stored by p, the $k+1^{st}$ lowest ranked peer t for b, and send a message to t notifying the failure of p with respect to block b. Peer t, if not already storing a copy of b, can proceed to download a copy from one of the peers storing replicas. Although simple, this by itself might not be sufficient. There might be more than one of the original replicas that have become unavailable, and even if there were only one, t might not be reliable enough to restore the required availability for the block, and more copies may be required. Worse yet, t might currently be failed itself.

Instead, q employs the following protocol. It contacts each of the peers in order of ranking for b, starting with the lowest (and skipping p). Each time q receives a response from a peer t, it incorporates the corresponding reliability $\mathcal{R}(t)$ into a total score for b. Once the combined result exceeds the required reliability, q stops. On the receiving side: when a peer t receives such a request from q, it returns a positive response. Then it check to see if it is already storing a copy of b. If not, t locates a replica using the hash function and the full membership list, and downloads a copy. This completes the aggressive recovery protocol.

Eventuallly, this will lead to too many replicas being used. We also need a mechanism for peers to drop unnecessary replicas. A simple technique could be as follows. Periodically, a peer q selects a block b it stores for which it is not one of the k lowest ranked peers. (Remember that k is a function of the block identifier and the membership information.) q then proceeds to run a protocol much like the aggressive recovery protocol above, scanning the peers in order of ranking. If q does not reach itself because enough reliability is established before then, q can drop the block. This protocol also causes recovering peers to download blocks they are missing automatically.

Note that the initial block installation and the lookup protocol would look much like the recovery protocol as well, scanning peers in order of ranking for the block, although the lookup protocol would stop at the first positive response. Yet another instance is when the reliability of certain peers is reduced over time, requiring more replicas to be created. Therefore, the scanning protocol is the basic building block for both for maintaining and locating replicas.

7 Related Work

Previous work on replica placement has focussed on achieving high throughput and/or low latency rather than on supporting high availability. Acharya and Zdonik [8] advocate locating replicas according to predictions of future accesses (basing those predictions on past accesses). In the Mariposa project [9], a set of rules allows users to specify where to create replicas, whether to move data to the query or the query to the data, where to cache data, and more. Consistency is transactional, but no consideration is given to availability. In [10], Wolfson et al. consider strategies to optimize database replica placement in order to optimize performance. The OceanStore project also considers replica placement [11, 12] but from the CDN (Content Distribution Network, such as Akamai) perspective of creating as few replicas as possible while supporting certain quality

of service guarantees. There is a significant body of work (e.g., [13]) on placement of web page replicas as well, all from the perspective of reducing latency and network load.

In FARSITE [14], Douceur and Wattenhofer investigate how to maximize the worst-case availability of files while spreading the storage load evenly across all servers [15, 16]. Servers are assumed there to have varying availabilities. The algorithms they consider repeatedly swap files between machines if doing so improves file availability. The results are of a theoretical nature for simple scenarios; it is unclear how well these algorithms will work in a realistic storage system.

Storage requirements can be reduced by using erasure coding, and this has been proposed for the OceanStore system [17]. However, erasure codes require a minimum number of replicas in order to be effective, and are mostly used to increase availability for the same storage requirements, not the reduce storage requirements at the same availability like our technique. Having said this, we believe that erasure coding is orthogonal to our approach, and the two can be combined to provide higher availability at lower storage costs.

Finally, the Recovery-Oriented Computing project (ROC) [1] also aims to improve availability by speeding up recovery. In the ROC project the emphasis is on fast rollback in order to recover from such failures as misconfigurations or bad upgrades, not on replication and roll-forward recovery.

8 Conclusion and Future Work

This position paper proposes a new approach to replica placement in a large scale p2p storage facility in order to increase efficiency. The two major innovations of the approach are:

- pseudo-random placement in the identifier space, rather than placement on consecutive peers in the identifier space, leading to the potential for extremely fast recovery from failures. This in turn allows a smaller number of necessary replicas in order to establish a desired minimum level of availability of objects;

- using a variable number of replicas based on the measured reliabilities of the peers; This also results in reducing the number of necessary replicas, as it is no longer necessary to choose the number of replicas based on worst case peer reliability assumptions;

We have been silent on the issue of consistency. The easiest approach would be to build an immutable storage facility, avoiding the issue altogether. We believe, however, that the chain replication approach of [5] is adaptable to our environment. The replicas in chains would be organized by ranking, with the lowest ranked peer forming the head and the highest ranked peer the tail for a chain. Given that replicas can find each other using the full membership DHT, they can recover from failures.

Acknowledgements

We would like to thank Fred Schneider for many suggestions for improvements, and Mark Linderman for discussions on the random placement strategy.

References

[1] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery oriented computing (roc): Motivation, definition, techniques, and case studies," Tech. Rep. UCB/CSD-02-1175, UCB, Mar. 2002.

[2] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, Oct. 2001.

[3] 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*, Banff, Canada, Oct. 2001.

[4] S. Ghermawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. of the 19th ACM Symp. on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.

[5] R. van Renesse and F.B. Schneider, "Chain replication for supporting high throughput and availability," in *Proc. of the 6th Symp. on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[6] R. Rodrigues and C. Blake, "When multi-hop routing matters," in *Proc. of the 3rd Int. Workshop on Peer-To-Peer Systems*, San Diego, CA, Feb. 2004.

[7] R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Proc. of Middleware'98*, The Lake District, UK, Sept. 1998, IFIP, pp. 55–70.

[8] S. Acharya and S.B. Zdonik, "An efficient scheme for dynamic data replication," Tech. Rep. CS-93-43, Brown University, Sept. 1993.

[9] M. Stonebraker, P.M. Aoki, R. Devine, W. Litwin, and M. Olson, "Mariposa: A new architecture for distributed data," in *Proc. of the 10th Int. Conf. on Data Engineering*, Houston, TX, 1994.

[10] O. Wolfson, S. Jajodia, and Y. Huang, "An adaptive data replication algorithm," *ACM Transactions on Computer Systems*, vol. 22, no. 2, pp. 255–314, June 1997.

[11] D. Geels and J. Kubiatowicz, "Replica management should be a game," in *Proc. of the 10th European SIGOPS Workshop*, Saint-Emilion, France, Sept. 2002, ACM.

[12] Y. Chen, R.H. Katz, and J. Kubiatowicz, "Dynamic replica placement for scalable content delivery," in *Proc. of the 1st Int. Workshop on Peer-To-Peer Systems*, Cambridge, MA, Mar. 2002.

[13] L. Qiu, V.N. Padmanabhan, and G.M. Voelker, "On the placement of web server replicas," in *Proc. of the 20th INFOCOM*, Anchorage, AK, Mar. 2001, IEEE.

[14] A. Adya, W.J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, and R.P. Wattenhofer, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," in *Proc. of the 5th Symp. on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002, USENIX.

[15] 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 Reliable Distributed Systems*, Lisbon, Portugal, Oct. 2001.

[16] 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.

[17] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proc. of the 1st Int. Workshop on Peer-To-Peer Systems*, Cambridge, MA, Mar. 2002.