ADAPTIVE ERASURE RESILIENT CODING IN DISTRIBUTED STORAGE

Jin Li
Communication and Collaboration Systems, Microsoft Research
One Microsoft Way, Bld. 113, Redmond, WA 98052
Email: jinl@microsoft.com

ABSTRACT

A challenge of peer-to-peer (P2P) storage is to build reliable data storage from inherently unreliable P2P network, and to do so efficiently. In this paper, we investigate the use of adaptive erasure resilient code (ERC) which changes ERC fragment size according to the size of the file distributed. We show that the adaptive ERC may greatly improve the efficiency and reliability of P2P storage. A number of design policies and strategies for the P2P storage application are also investigated.

1 Introduction

In a P2P application, the peers bring with them resources (network bandwidth and/or hard drive storage) when they join the service. As the demand on the P2P system grows, the capacity of the system grows as well. This is in sharp contrast to a client-server system, where the server capacity is fixed and paid for by the provider. As a result, the P2P system is economy to run and superb scalable.

In this paper, we are particularly interested in P2P storage (distributed storage) applications. In such systems, the peer contributes not only the bandwidth but also the storage space to serve the other peers. The collective storage space contributed by the peers forms a distributed storage cloud. Data may then be stored into and retrieved from the cloud. The distributed storage has been widely investigated, and is the basis of many P2P applications. The OceanStore project [3] builds a utility infrastructure that provides continuous access to persistent information. CAN[4] introduces distributed infrastructure that provides hash table-like functionality on Internet-like scales. CFS[5] designs a P2P read-only storage system that provides guarantees for efficiency, robustness, and load-balance. Kademlia[13] designed a P2P system with proven consistency and performance in a fault-prone environment. Kademlia routes queries and locates nodes using an XOR-based metric topology that simplified the algorithm. It was adopted by both [11] and [12] for trackerless BitTorrent implementation. PAST[7] designs a large-scale, internet-based, global storage utility that provided scalability, high availability, persistence and security. Coral[8] creates self-organizing clusters of nodes that fetches information from each other to avoid communicating with more distant or heavily loaded servers.

Though the peers in the P2P network may act like servers, they differ from commercial web/database servers in one important aspect: the reliability. Because the peer is usually an ordinary computer that supports the P2P application with its spare hard drive space and idle bandwidth resource, it is far less reliable than the server. The user may choose to turn off the peer computer or the P2P application from time to time. Compulsory need, e.g., large file upload/download, may starve the peer from the necessary bandwidth for P2P activity. The peer computer may be offline due to the need to upgrade and patch the software/hardware, or due to virus attack. The computer hardware and the network link of the peer are also inherently much more unreliable than the server computer and its commercial network links, which are designed for reliability. While commercial server/server clusters are designed for "six nine" reliability (with a failure rate 10^{-6}, at that rate, about 30*s* of downtime is allowed each year), a good consumer peer may have only "two nine" reliability (failure rate 10^{-2}, about 15*min* downtime every day), and it is not uncommon for peers to have only 50% (down half the time) or even 10% reliability (down 90% of the time).

Most P2P applications, e.g., P2P backup and data retrieval, want to maintain the same level of reliability for P2P storage as that of the server ("six nine" reliability). The challenge is: how to build a reliable P2P store? Can reliable P2P store be built efficiently, with minimum use of bandwidth and storage resources of the peers? It has been pointed out [2] that to store large amount of dynamic data reliably in P2P network with unreliable peers, we would need huge amount of cross-system bandwidth. To improve reliability in P2P storage and improve efficiency, erasure resilient code (ERC) is a proven technology in P2P store, e.g., [9][10]. Lin et. al. [16], Rodrigues et. al. [17] and Weatherspoon et. al [18] have compared ERC vs. replication in P2P storage, and demonstrated the effectiveness of ERC in P2P. The contribution of this paper is the proposal of an adaptive ERC scheme that switches ERC fragment size for optimal efficiency. We have also investigated into the design of proper policies that improve the performance of P2P storage and balance the contribution and benefit of the peers.

The outline of the paper is as follows. The design of efficient and reliable P2P storage system with adaptive ERC is described in Section 2. We investigate the proper P2P storage policies in Section 3. Conclusions are given in Section 4.

2 Adaptive ERC in Distributed Storage

2.1 Erasure resilient coding in P2P

The adhoc solution to bring reliability to a system with unreliable parts is to use redundancy. If each individual peer on the network has a reliability of p, to achieve a desired reliability of p_0, we may simply replicate the information to n peers:

$$n = \log(1 - p_0) / \log(1 - p), \qquad (1)$$

where n is the number of peers holding the information. Though achieving reliability, the simple replication strategy is not efficient. For example, with peer reliability 50%, we will need to replicate and store the information to 20 peers to achieve "six nine" reliability. This leads to 20 times more bandwidth and storage space to distribute and store the information. Obviously, efficiency has been sacrificed in exchange of information reliability.

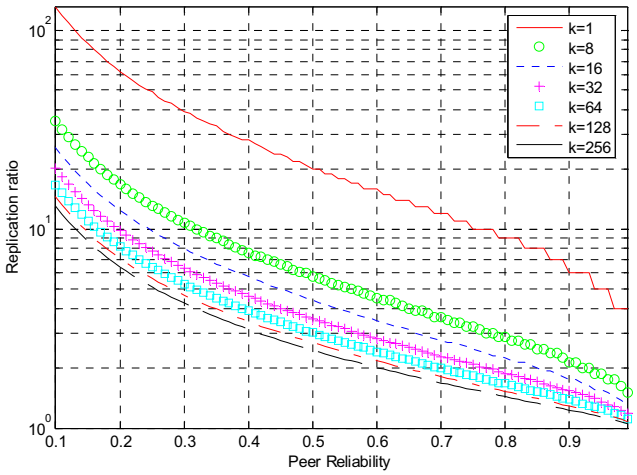

Figure 1 Peer reliability and desired replication ratio.

To improve efficiency while still maintaining the same reliability level, ERC can be used. ERC splits the original file into k original fragments $\{x_i\}$, i=0,⋯,k-1, each of which is a vector over the Galois Field GF(q), where q is the order of the field. Say we are encoding a file that is 64KB long, if we use q=2^{16} and k=16, each fragment will be 4KB long, and will consist of 2K word, with each word being an element of GF(2^{16}). ERC then generates coded fragments from the original fragments, An ERC coded fragment is formed by operation:

$$c_j = \mathbf{G}_i \bullet [x_0 \quad x_1 \quad \cdots \quad x_{k-1}]^t, \tag{2}$$

where c_j is a coded fragment, $\mathbf{G_i}$ is a k-dimensional generator vector, and equ (2) is a matrix multiplication, all on GF(q). At the time of decoding, the peer collects m coded fragments, where m is a number equal to or slightly larger than k, and attempts to decode the k original fragments. This is equivalent to solve the equation:

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} \mathbf{G_0} \\ \mathbf{G}_1 \\ \vdots \\ \mathbf{G}_{m-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{k-1} \end{bmatrix}, \tag{3}$$

If the matrix formed by the generator vectors has a full rank k, the original messages can be recovered.

There are many available ERCs. We have compared different forms of ERC and their performances in the P2P application in [15]. A particularly interesting one is the Reed-Solomon (RS) code [14]. RS code uses structured generator vectors, and is maximum distance separable (MDS). As a result, any k distinctive coded fragments will be able to decode the original fragments. Another advantage of the RS code is that the coded fragment can be easily identified and managed by the index i of the generator vector, thus eases the detection of duplicate RS codes. Reed-Solomon ERC can also be implemented efficiently, e.g., with an encoding/decoding throughput of 200Mbps in [14]. In the following discussion, we assume that RS code is used.

2.2 ERC: fragment size

By using ERC in P2P storage, a data/media file is distributed to more peers, but each peer only needs to store one coded fragment that is 1/k size of the original file, leading to an overall reduction in the bandwidth and storage space required to achieve the same level of reliability, and thus an improvement of efficiency. Let n_1 be the number of peers that the coded fragments needs to be distributed to

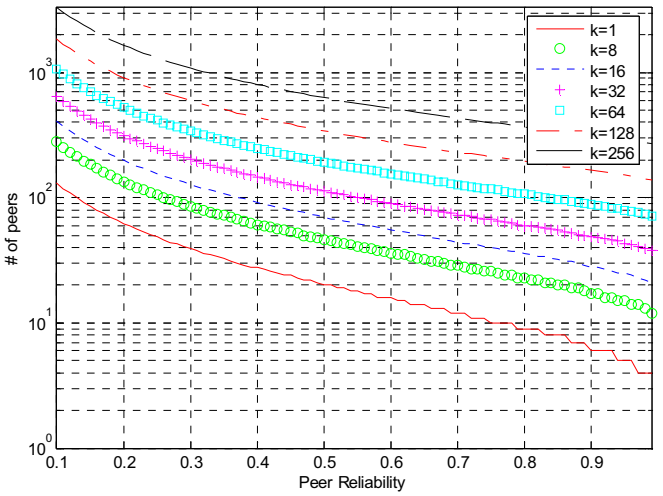

Figure 2 Number of information storing peers to achieve desired reliability of 10^{-6} under erasure resilient coding.

achieve a desired reliability level. Since RS code is MDS code, k peers holding k distinctive coded fragments will be sufficient to recover the original file. In a network of n_1 peers, the probability that there are exactly m peers available can be calculated via binomial distribution:

$$p(m,\ n_1) = \binom{n_1}{m} p^m (1-p)^{n_1-m}. \tag{4}$$

We may thus calculate n_1 from p, p_0 and k as:

$$n_1 = \arg\min_o \left\{ \sum_{m<k} \binom{o}{m} p^m (1-p)^{o-m} < 1-p_0 \right\}. \tag{5}$$

We define the replication ratio r as:

$$r = n_1 / k. \tag{6}$$

The replication ratio r is a good indicator of efficiency, as r copies of files needs to be distributed and stored into the P2P cloud.

We show in Figure 1 the desired replication ratio to achieve "six nine" reliability for different ERC fragment size k. We observe that the use of ERC greatly reduces the required replication ratio. Comparing non ERC (k=1) and ERC of fragment size k=256, the desired replication ratio decreases from r=132 to r=13.1 for peer reliability of 10%, from r=20 to r=2.5 for peer reliability of 50%, and from r=3 to r=1.05 for peer reliability of 99%. ERC may improve the efficiency without sacrificing the reliability.

We also observe that larger ERC fragment size further reduces the replication ratio. With peer reliability of 50%, going from k=8 to 16, 32, 64, 128 and 256 leads to a reduction of the replication ratio from r=5.75 to 4.375, 3.53, 3.02, 2.68 and 2.48. The corresponding efficiency improvement is 24%, 19%, 15%, 11% and 8%, respectively. This seems to suggest that we should use large ERC fragment size for more efficiency.

However, larger ERC fragment size needs more peers to store and to retrieve the coded fragments. Shown in Figure 2, we plot the number of peers that need to hold the coded fragments to achieve the "six nine" reliability. Again with 50% peer reliability, going from k=8 to 16, 32, 64, 128 and 256 increases the number of information storing peers from n_1=46 to 70, 113, 193, 343 and 630. Each doubling of k results in 52%, 61%, 71%, 78%, 84% more peers to store information. Doubling of k also requires at least double number of peers during information retrieval.

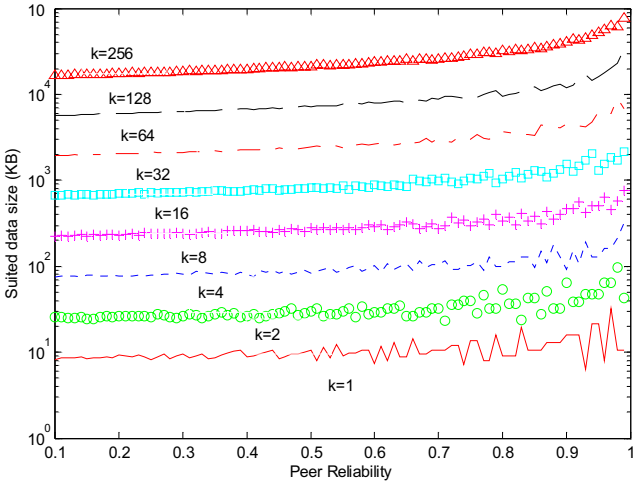

Figure 3 ERC fragment size and its suited file size for information storage in P2P network.

In most practical P2P network, establishing connection between the peers needs a non-trivial overhead. One part of the overhead can be attributed to the retrieval of proper peer identity and finding the proper routing path (e.g., via DHT[1][6]). Another part of the overhead is due to the need to invoke certain NAT traversal algorithm, e.g., STUN (simple traversal of UDP through NAT) if one or both peers are behind the NAT. Assuming that the average overhead to establish connection between the two peers is *overhead* (set to 16*KB* to facilitate the following discussion), we may calculate the overall network bandwidth needed to store a file of size *s* as:

$$store\_bandwidth = s * r + n_1 * overhead. \quad (7)$$

With equ (7), we recognize that larger ERC fragment size does not always lead to the best efficiency. In stead, for small file, small ERC fragment size or even non ERC should be used. We calculate the optimal file size boundary between different ERC fragment size and plot the curves in Figure 3. For example, the bottom curve of Figure 3 shows the file size boundary below which non ERC should be used, and above which ERC with fragment size k=2 should be used. The boundary curves shown in Figure 3 zigzag, as the number of peers required to achieve the desired reliability decreases with the increase of peer reliability. An interesting observation is that the file size boundary is relatively insensitive to peer availability, which greatly simplifies the choice of the optimum ERC fragment parameter. In general, for file smaller than 10KB, ERC should not be used. For ERC with fragment size k=2, 4, 8, 16, 32, 128 and 256 the most suited file size range is approximately 10-33KB, 33-100KB, 100-310KB, 310-950KB, 950KB-2.9MB, 2.9MB-8.9MB, 8.9-26MB, >26MB, respectively.

2.3 Adaptive ERC scheme

An optimal strategy to efficiently store content in P2P reliably is thus to adaptively choose the appropriate ERC fragment size. Using the file boundary curve established in Figure 3, we may adaptively choose to use non ERC, and ERC with fragment size k=2, 4, 8, 16, 32, 64, 128, 256 for different file size. We compare the adaptive ERC approach with fixed parameter ERC, and show the difference in network bandwidth usage in Figure 4, where peer reliability is 50%. Compared with using a fixed ERC fragment size of k=1 (non ERC), 8, 32 and 256, the adaptive ERC method may

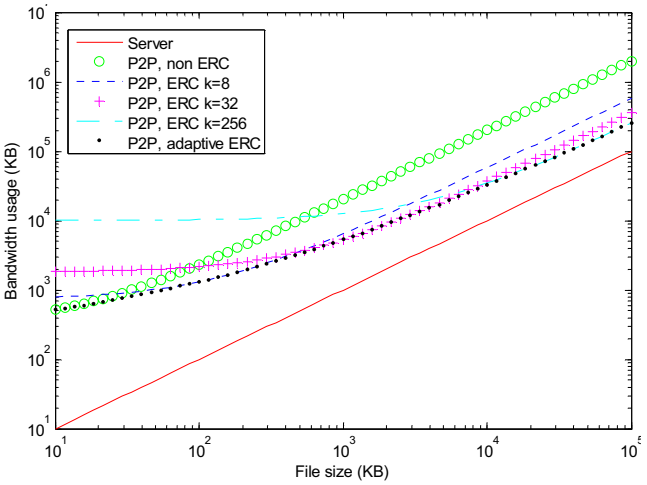

Figure 4 Bandwidth usage between P2P with adaptive ERC and fixed ERC (peer reliability=50%).

improve the efficiency by an average of 61%, 26%, 25% and 50%. The improvement in efficiency is significant.

3 P2P Storage: Policies and Design Strategies

3.1 P2P storage cost

In this section, we compare storing a file in a P2P network to storing the file directly in a "six nine" reliable server. The P2P solution always requires more bandwidth to distribute the file into the P2P storage. The increase in the upload bandwidth of the client can be considered a cost of P2P storage system. This cost under different peer reliability and file size can be tabulated in Table 1.

Table 1 Extra cost of increased bandwidth usage in P2P.

	File Size				
Reliability	10KB	100KB	1MB	10MB	100MB
10%	332.9	79.1	29.5	16.5	12.5
50%	51.0	12.11	4.34	2.23	1.56
99%	9.4	1.87	0.65	0.22	0.09

We observe that the cost of using P2P storage is small if the peer reliability is high and the file size is large. For example, storing 100MB of file to peers with reliability of 99% only incurs 9% additional cost. However, when the peer reliability is low and the file size is small, the cost can be significant.

3.2 P2P storage policies

From Table 1, we may derive the following policies of using the P2P storage cloud:

a) We should use the unreliable peers for large files, and use reliable peers for small files.

The cost to the P2P storage will be smaller if we allocate large files to the unreliable peers, and assign smaller files to the reliable peers.

b) We should use unreliable peers for static files, and use reliable peers for dynamic files.

We call those files that do not change as static, and call those files that change constantly as dynamic. To efficiently store small static files, we can bundle multiple static files into a large static file and store the combined file in the P2P storage cloud. The same strategy is not effective for dynamic files, as the change of a single

file requires the update of the entire combined file. As a result, it is better to store dynamic small files in the reliable peers.

A corollary of the policy is that if we use the P2P network to store state of the application (small and dynamic), e.g., torrent file of BitTorrent, peer status information, etc., we should divert the information to the most reliable peers of the network. For example, current trackerless BitTorrent implementation [11] replicates the torrent file to 20 peers for reliability purpose. If we restrict that the torrent file only be placed in high reliable peers (in essence, the high reliable peers will form a sub-network that constitute the cores of the extended P2P network), we may greatly reduce the replication ratio and the cost of updating the torrent, and improve the efficiency.

c) The unreliable peers should be allowed to distribute less, and the reliable peers should be allowed to distribute more.

d) The smaller file should be assigned a higher distribution cost, and the larger file should be assigned a lower distribution cost.

Policies c) and d) are for P2P backup and retrieval applications. A P2P storage network should let each peer balance its contribution and benefit. Say a certain peer with reliability p wants to store an item of size s into the distributed storage. From the item size, we may calculate the optimal ERC fragment size k according to Figure 3. Then, we can calculate the replication ratio r to achieve the desired reliability. To balance the P2P resource consumed, the peer must host $s \cdot r$ amount of content from the other peers. Because the peer with high reliability p requires a small replication ratio r to achieve the same level of reliability, such policy rewards the reliable peers so that they will be allowed to distribute more for the same resource contributed, and punishes the unreliable peers. This may have a positive benefit in P2P economy, as it encourages the user to increase the peer reliability by prolonging the online session, thus improves the overall reliability and reduces the replication ratio required in the entire P2P network. Since the small file uses smaller ERC fragment size, which leads to an increase of the replication ratio under the same peer reliability, the policies also punish the distribution of small files and reward the distribution of large files (or bundling small files into a large combined file for distribution).

3.3 P2P storage with server component support

If server component is used in complement of the P2P network, we may use the P2P storage for large and static files, and use the server for small dynamic files. Since it is the large files that consume most of the server resource, P2P storage complement server well.

An interesting idea in P2P backup with server support is to let dynamic files be backed up to server first. The client and/or the server may then automatically detect those dynamic files that are not changed any more and are turning into static files. These detected static files may then be bundled together into a large file and be distributed with ERC into the P2P storage cloud. This effectively increases the size of the file stored in the P2P cloud. Combined with ERC of large fragment size, this may improve the efficiency.

4 Conclusions

We investigate the problem of storing data efficiently and reliably in a P2P network. Adaptive ERC is proposed that adjusts ERC fragment size based on the file size distributed. We also propose a number of design policies for the P2P storage. We indicate that the small, dynamic data should be diverted to the more reliable peers or even a server, while the large and static files may be stored utilizing the storage capacity of the unreliable peers. Also, for balanced contribution and benefit, the peer should host the same amount of content as it stored in the P2P network. As a result, the unreliable peers should be allowed to distribute less, and the reliable peers should be allowed to distribute more. Also, smaller file should be assigned a higher distribution cost, and the larger file should be assigned a lower distribution cost.

5 References

[1] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, “Building peer-to-peer systems with Chord, a distributed lookup service", in Proc. 8th Workshop on Hot Topics in Operating Syst., (HotOS-VIII), May 2001.

[2] C. Blake and R. Rodrigues, “High availability, scalable storage, dynamic peer networks: pick two”, in Proc. 9th Workshop on Hot Topics in Operating Syst., (HotOS-IX), (Lihue, Hawaii), May. 2003, pp.1-6.

[3] J. Kubiatowicz, et. al., “OceanStore: an architecture for global-scale persistent storage”, in Proc. ACM ASPLOS, Nov. 2000.

[4] S. Ratnasamy, P. Francis and M. Handley, “A scalable content-addressable network”, in ACM SIGCOMM Conference. ACM Press, San Diego (CA), August 2001.

[5] F. Dabek, et. al., “Wide-area cooperative storage with CFS”, in Proc. ACM SOSP’01, Oct. 2001, Banff, Canada.

[6] A. Rowstron, P. Druschel, “Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems", in Proc. IFIP/ACM Middle-ware, 2001.

[7] P. Druschel and A. Rowstron, “PAST: A large-scale, persistent peer-to-peer storage utility”, in Proc. 8th Workshop on Hot Topics in Operating Syst., (HotOS-VIII), May 2001.

[8] M. Freedman and D. Eres, “Sloppy hashing and self-organizing clusters”, in IPTPS’03, Berkeley, CA, Feb. 2003.

[9] Z. Zhang and Q. Liang, “Reperasure: replication protocol using erasure-code in peer-to-peer storage network”, in Proc. of 21st Sym. On Reliable Distributed Systems, (SRDS’02), Oct. 2002.

[10] F. M. Cuenca-Acuna, R. P. Martin and T. D. Nguyen, “Autonomous replication for high availability in Unstructured P2P systems”, in Proc. 22nd IEEE Int. Symp. On Reliable Distributed Systems, 2003.

[11] The Azureus page: http://azureus.sourceforge.net/

[12] The official BitTorrent page: http://www.bittorrent.com/

[13] P. Maymounkov and D. Mazieres. “Kademlia: A peer-to-peer information system based on the XOR metric.” in Proceedings of IPTPS02, Cambridge, USA, March 2002.

[14] J. Li, “The efficient implementation of Reed-Solomon high rate erasure resilient codes”, in Proc. ICASSP’2005, Philadelphia, PA, Mar. 19-23, 2005.

[15] J. Li and Q. Huang, “Erasure resilient codes in peer-to-peer storage cloud”, in Proc. ICASSP’2006,

[16] W. K. Lin, D. M. Chiu, Y. B. Lee. "Erasure code replication-revisited," in Proc. 4th International Conference on P2P Computing, Zurich, Switzerland, Aug. 2004.

[17] R. Rodrigues and B. Liskov. "High availability in DHTs: erasure coding vs. replication." In Proc. IPTPS 2005, Ithaca, NY. Feb. 2005.

[18] H. Weatherspoon and D. Kubiatowicz, “Erasure coding vs. Replication: a quantative comparison”, in Proc. IPTPS 2002, Mar. 2002.