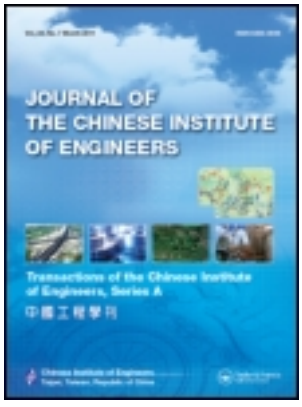

Journal of the Chinese Institute of Engineers
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tcie20

A low-cost, high-availability P2P storage scheme with regenerating code that combines caching methods

Yu-Liang Chen [a b] & Shyan-Ming Yuan [a c]

[a] Department of Computer Science , National Chiao Tung University , 1001 Ta Hsueh Road, Hsinchu 300 , Taiwan

[b] Department of Information Management , Ta Hwa University of Science and Technology, Qionglin Shiang Hsinchu County 307 , Taiwan

[c] Department of Computer Science and Information Engineering , Providence University , 200 Chung-Chi Road, Salu District, Taichung City 43301 , Taiwan

Published online: 13 Sep 2012.

To cite this article: Yu-Liang Chen & Shyan-Ming Yuan (2012) A low-cost, high-availability P2P storage scheme with regenerating code that combines caching methods, Journal of the Chinese Institute of Engineers, 35:6, 735-745, DOI: 10.1080/02533839.2012.701892

To link to this article: http://dx.doi.org/10.1080/02533839.2012.701892

Journal of the Chinese Institute of Engineers
Vol. 35, No. 6, September 2012, 735–745

Taylor & Francis
Taylor & Francis Group

A low-cost, high-availability P2P storage scheme with regenerating code that combines caching methods

Yu-Liang Chen[ab]* and Shyan-Ming Yuan[ac]

[a]*Department of Computer Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu 300, Taiwan;* [b]*Department of Information Management, Ta Hwa University of Science and Technology, Qionglin Shiang Hsinchu County 307, Taiwan;* [c]*Department of Computer Science and Information Engineering, Providence University, 200 Chung-Chi Road, Salu District, Taichung City 43301, Taiwan*

(*Received 28 December 2010; final version received 27 April 2011*)

Peer-to-peer (P2P) storage systems are expected to be fast, fault-tolerant, scalable, and reliable. Thus, high data availability is an essential feature of P2P storage systems. In this article, we present a scheme that utilizes a regenerating code (RC) with caching to improve the performance of content services provided by P2P storage systems. Erasure coding is commonly employed to support high availability, but this technique requires the original file to produce redundant data. In contrast, RC solves the problem by collecting the encoded information. However, RC requires more peers to decode the blocks, which is difficult in a P2P environment because more peers must simultaneously remain active to hold the file blocks. Based on the RC, we store the information for a peer that has recently accessed blocks and utilized the data in the peer's LRU cache to increase the access performance and reduce the encoding cost. We carried out a series of experiments with different cache sizes under various levels of P2P availability. The results show that our scheme can outperform the traditional RC system in terms of access performance, allowing access at least 83% of the time, while also achieving a lower cost.

Keywords: P2P; erasure coding; regenerating code

1. Introduction

There are many designs for peer-to-peer (P2P) storage systems, including OceanStore (Kubiatowicz *et al.* 2000), CFS (Dabek *et al.* 2001), PAST (Rowstron and Druschel 2001), and Total Recall (Bhagwan *et al.* 2004). In P2P storage systems, data availability is the main concern because of the need to provide quality service. Replication and erasure coding schemes are common methods for creating redundant data to achieve high availability. In addition, there is a new application of network coding: regenerating code (RC) (Dimakis *et al.* 2010). Unlike erasure coding, which requires that a full file be stored before encoded blocks are generated, RC can generate a new encoded block by collecting sufficient existing encoded blocks without using the entire file. Although RC offers lower storage costs and bandwidth than erasure coding, it suffers from poor access performance (Dimakis *et al.* 2010). In this article, we propose a method for improving the access performance of P2P storage systems.

Below, we discuss the work done in this area and provide a description of how to solve the problem. Then, in Sections 4 and 5, we present the experiment and measurement results. Finally, conclusions are drawn in Section 6.

2. Related work

Many users may associate P2P networks with well-known P2P applications, such as BitTorrent and Skype. A P2P network connects all participants and allows them to share information and utilize bandwidth as a group. Lua *et al.* (2005) classified P2P networks as unstructured and structured P2P networks. In structured P2P networks, the links and the connected peers form an overlay network according to predefined rules. Structured P2P networks typically use distributed hash-table-based (DHT) indexing to locate peers; for example, they might use the Chord system (Stoica *et al.* 2001). Unstructured P2P networks, in contrast, do not provide any algorithms for use in organizing or optimizing the network connections. Hence, queries may not always be answered in an unstructured P2P network, and they may have to be flooded through the network to as many peers as possible. Ahlswede (2000) initiated a wide variety of

*Corresponding author. Email: imklchen@gmail.com

ISSN 0253–3839 print/ISSN 2158–7299 online

http://dx.doi.org/10.1080/02533839.2012.701892
http://www.tandfonline.com

research on the application of network coding. Médard and Koetter (2003) and Li *et al.* (2003) demonstrated that linear network coding can be used to achieve maximum multicast capacity. The work of Ho *et al.* (2006) and Sander *et al.* (2003) proves that random linear network coding can be used to achieve maximum multicast capacity when the Galois field is sufficiently large. Moreover, the work of Jaggi *et al.* (2005) includes deterministic polynomial time algorithms used to design linear codes in directed acyclic graphs with edges of unit capacity. An application of network coding to a single-source P2P file-sharing is presented in Yang and Yang (2008). The fundamental purpose of network coding is to allow data mixing at intermediate network nodes to avoid data collision and the process of fusing data in the intermediate nodes subverts the concept of conventional routing processes. Network coding has been used in many fields, including throughput, wireless resources, security, complexity, and resilience to link failures (Fragouli and Soljanin 2007). Emerging applications of network coding including network monitoring, switch operation, on-chip communication, and distributed storage were surveyed by Fragouli and Soljanin (2008). As a means of achieving high availability, three traditional redundancy schemes have been proposed: a replication scheme, an erasure coding scheme, and a hybrid scheme. The replication scheme replicates r copies of a file and distributes the r copies to other peers in the system. The r here is a redundancy factor based on the system requirements. The second scheme divides a block into m fragments and then produces n redundant fragments, where $n > m$. The n encoded fragments are then distributed to the peers in the system (Figure 1). A key property of erasure coding is that for reconstruction of the block, only m fragments are needed out of the n total fragments that resulted from the erasure coding procedure. The redundancy is represented as n over m. Again, redundancy is determined according to the system requirements.

The third scheme is intended to overcome the weakness of erasure coding. In the erasure coding scheme, when a peer with an encoded block leaves or crashes, the system will begin the process of regenerating a new encoded block to maintain file availability. However, to generate a new encoded block, the system must use the original file. This means that the system first needs to collect m encoded fragments to recover every block of the file whenever reconstructing encoded blocks. As a result, the cost of repairing bandwidth in the erasure coding scheme is quite high. Thus, the hybrid scheme stores one full replica (Figure 2) in a special node to eliminate the inefficiency of repairs under the erasure coding scheme.

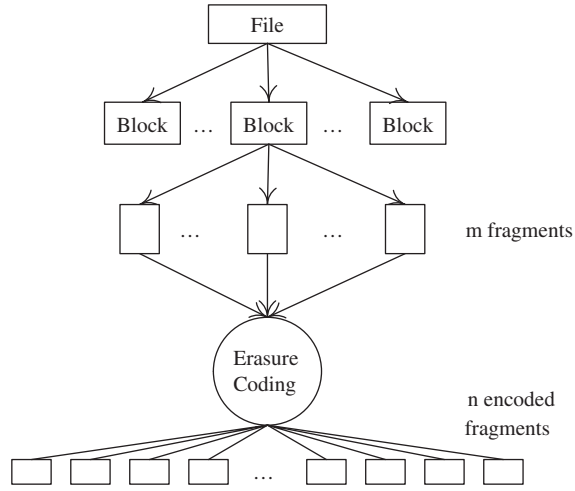

Figure 1. A depiction of how erasure coding is applied to a file to produce n total fragments for a given block.

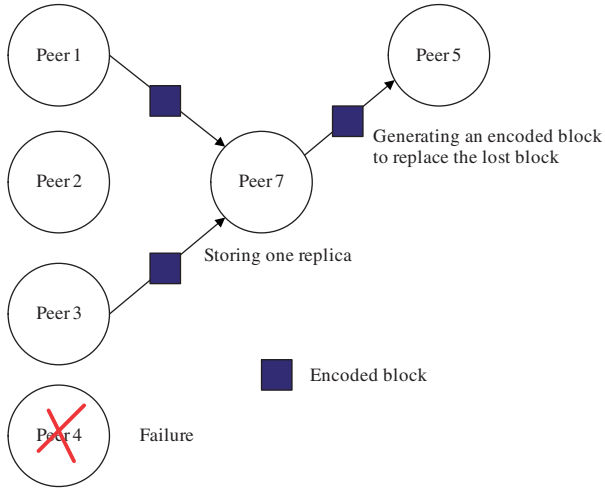

Figure 2. The hybrid scheme recovers a lost block. Note that peer 7 holds a replica of the original file.

Although the erasure coding and hybrid schemes yield better results in terms of mean failure time and bandwidth, they require a high repair bandwidth. When one node fails or leaves, these two systems will need to acquire more nodes to recover the lost information for a new node. This recovery requires the network in question to have a significant bandwidth.

With this in mind, Dimakis *et al.* (2010) introduced the notion of RCs, in which a new node communicates functions of the stored data from the surviving nodes instead of downloading the actual data. The use of RCs can significantly reduce the repair bandwidth, as shown in Dimakis *et al.* (2010). Figure 3 presents a (4, 2) minimum storage RC. Here, the 4 means that

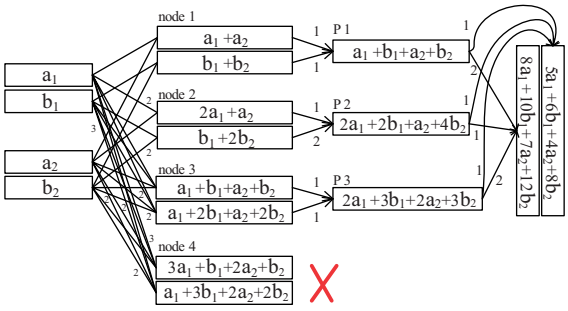

Figure 3. Repair for a (4, 2) minimum storage RC.

each file in the system is divided into four encoded blocks and the 2 means that any two of the four encoded blocks can be used to reconstruct the file. As we can see in this example, node 4 has failed, and the RC tries to recover the lost packets. All the packets (boxes) in this figure are 0.5 Mb in size, and each node stores two packets. For example, node 1 contains packets $a_1 + a_2$ and $b_1 + b_2$, node 2 contains packets $2a_1 + a_2$ and $b_1 + 2b_2$, node 3 contains packets $a_1 + b_1 + a_2 + b_2$ and $a_1 + 2b_1 + a_2 + 2b_2$, and node 4 contains packets $3a_1 + b_1 + 2a_2 + b_2$ and $a_1 + 3b_1 + 2a_2 + 2b_2$. Note that any two nodes have four equations that can be used to recover the original data: a_1, b_1, a_2, and b_2. The parity packets $P1$, $P2$, and $P3$ are used to create two packets for the newcomer, and this operation requires a repair bandwidth of 1.5 Mb. The multiplying coefficients are selected at random, and the example is shown in the integers for the sake of simplicity. The calculations are as follows:

$$\begin{aligned} 1P1 + 1P2 + 1P3 &= 1(a_1 + b_1 + a_2 + b_2) \\ &\quad + 1(2a_1 + 2b_1 + a_2 + 4b_2) \\ &\quad + 1(2a_1 + 3b_1 + 2a_2 + 3b_2) \\ &= 5a_1 + 6b_1 + 4a_2 + 8b_2 \end{aligned}$$

and

$$\begin{aligned} 2P1 + 1P2 + 2P3 &= 2(a_1 + b_1 + a_2 + b_2) \\ &\quad + 1(2a_1 + 2b_1 + a_2 + 4b_2) \\ &\quad + 2(2a_1 + 3b_1 + 2a_2 + 3b_2) \\ &= 8a_1 + 10b_1 + 7a_2 + 12b_2 \end{aligned}$$

The key point is that the nodes do not send what they contain but instead generate smaller parity packets of their data and forward them to the newcomer, which further mixes the encoded packets to generate two new packets. In this case, the required repair bandwidth of 1.5 Mb is smaller than the bandwidth that would be required to transmit the original file (2 Mb). In contrast, the erasure coding scheme involves accessing the original file to generate a new encoded block. However, although the use of RC solves the repair bandwidth problems, it cannot support frequent access; in other words, this scheme cannot provide content services. Therefore, Dimakis *et al.* (2010) limited the application of the scheme to archival storage or backup services.

3. System design

To improve the access performance of RCs, we add an LRU cache and store information from peers that have recently requested a file. We assume that all peers in the P2P storage system have two types of storage space: permanent 'database' and temporary space in the form of the 'LRU cache'. The encoded blocks in the database exist in the system until the peer crashes or leaves; however, the blocks in the LRU cache are replaced when the space is full. The replacement rule follows the algorithm for the LRU cache. In our study, we used Chord (Stoica *et al.* 2001) for lookup and query operations with a unique identifier for each peer and each file. The target file availability was set to 99.9%, as is expected by end users of today's web services (Merzbacher and Patterson 2002). Meanwhile, we used $(n.7)$ RC as in Dimakis *et al.* (2010) to encode the files, where n is chosen according to the required system redundancy and 7 according to the traces from real systems (Rodrigues and Liskov 2005, Dimakis *et al.* 2010). We also utilized the results for $(\alpha_{\mathrm{MSR}}, \gamma_{\mathrm{MSR}} = \mathrm{d}\beta) = \left(\frac{M}{k}, \frac{Md}{k(d-k-1)}\right)$, as presented by Dimakis *et al.* (2010), where α denotes the block size stored at each node, d is the number of connections to active nodes, β is the packet size with which a newcomer communicates to any d surviving nodes, and γ is equal to $d\beta$, which is the total repair bandwidth for the minimum storage regenerating (MSR) codes. Finally, M and k represent the original file size and the number of storage nodes that can recover the original file, respectively. If we set $d = 13$, each file will be divided into seven blocks and each block will contain seven packets (Figure 4). Clearly, the total packet number in a file is 49, and these packets can be used to generate the coded blocks. Each packet in the coded blocks also stores its combination coefficients with respect to the 49 packets in the initial file (Figure 5). For the linear combination procedure, we chose a large Galois field to successfully decode the packets. In addition, we assumed that the packet size was sufficiently large for us to ignore the coefficient overhead.

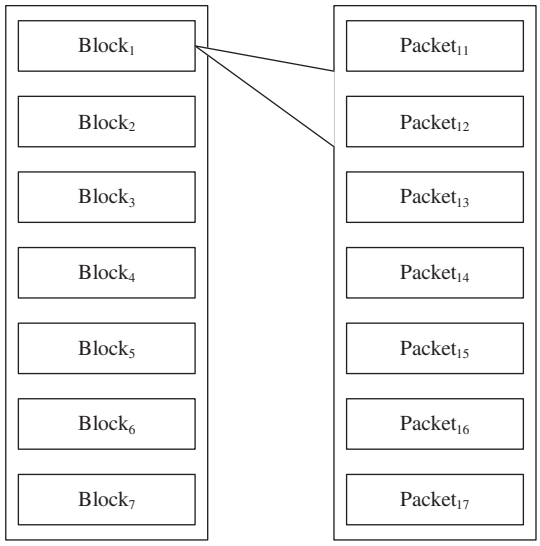

Figure 4. The relationship between the blocks and packets in a file.

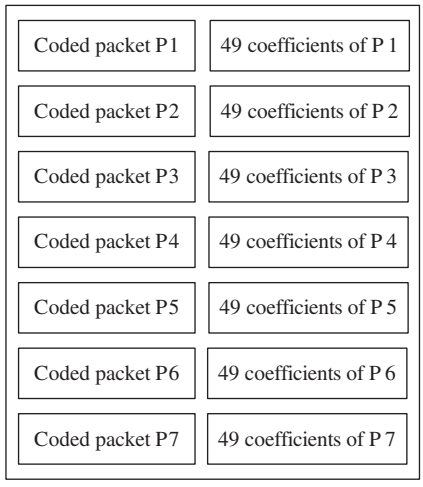

Figure 5. The composition of a coded block.

We chose to use the MSR code, a special kind of RC, to implement the system. As described above, we derived a block size and repair bandwidth of $(\frac{M}{7}, \frac{13M}{49})$. The MSR code requires contact with 13 different peers to encode a new coding block. If there are insufficient peers in the system, seven different blocks must be collected. In such a case, the process is the same as in the erasure coding scheme.

Our redundancy scheme is based on the work of Chen *et al*. (2008) with a RC, an LRU cache, and an index of the peers included. Overall, each peer has three roles: a register peer, an index peer, and a maintenance peer. The peers assume these roles while

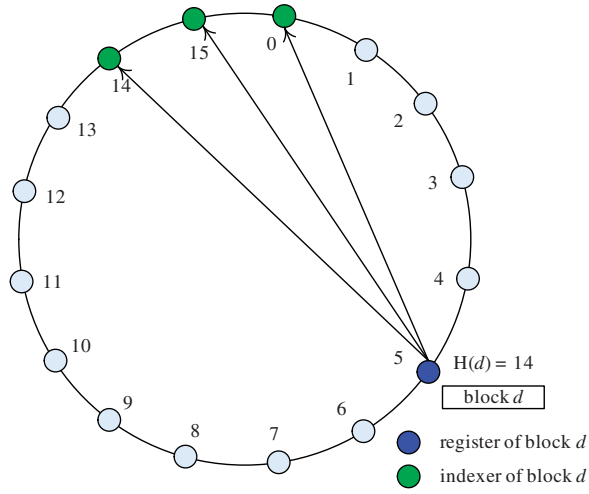

Figure 6. A peer registers block *d* with three indexers.

the system is running. When a peer joins the system, the peer will check to verify that it carries the encoded blocks. If this is the case, the peer will compute the DHT (block ID) to determine which peer it should register with. Thus, the peer acts as a register peer. Each peer may also be chosen as an index peer by a newly joined peer during the system operation. In this stage, the selected peer acts as an indexer. Finally, each peer may store encoded blocks and may periodically determine the number of registered blocks based on the index. If the number is less than the redundancy threshold, the peer will trigger an event that generates new encoded blocks to increase the availability of the file. During this stage, the peer acts as a maintenance peer. As a maintenance peer supported by the Chord protocol in the bottom layer, each peer controls data placement, data lookup, and data availability in the P2P storage system. All index and register peers periodically register the unique file identifiers for the coded blocks in their database with some indexers. Here, the coded blocks that belong to the same file have the same identifier but different coefficients. When the indexers receive a report, they will add a new index entry to their index that includes the IP of the reporting peer and the file identifiers. They will also set a timer for the entry and remove the index entry when the timer expires. In fact, the indexers use the timers to determine whether the register peers are active. The first indexer is determined using the hash function for DHT; the other indexers function as backups and are adjacent and continuous successors of the first indexer, as shown in Figure 6, where H(*d*) is the hash function for DHT based on the Chord protocol. In that figure, peer 5 registers block *d* with index peer 14; peer 15 and peer 0 are also registered by peer 5 as backup indexers.

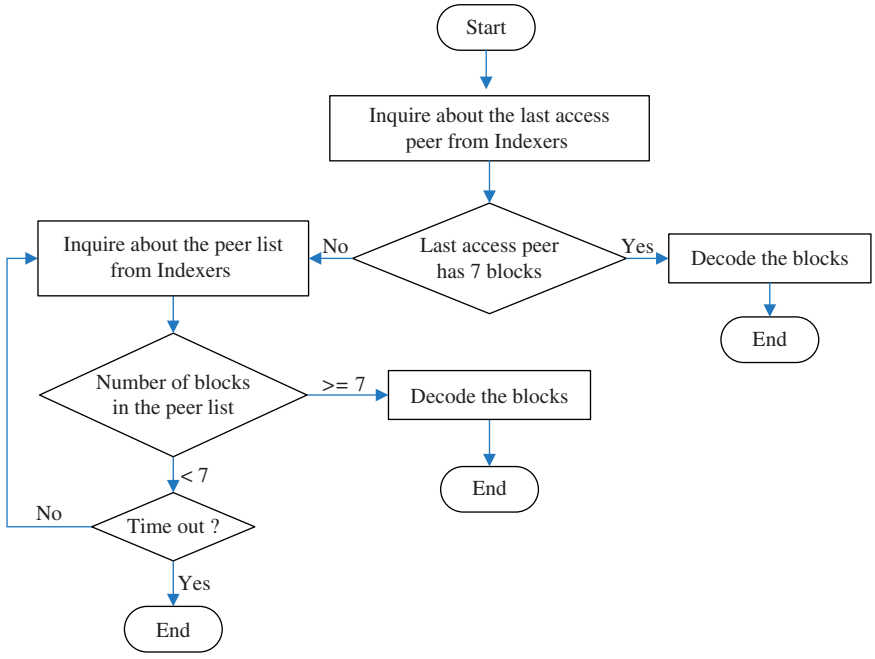

Figure 7. The flow chart for requesting a file.

The next section of this article explains the operations involved in requesting a file and generating a new coded block. In both operations, each peer first checks either its database or its LRU cache to locate the available blocks before requesting information from other peers.

After the peer makes a request of other peers, those peers will reply with information from the relevant file, having located that information in their databases or LRU caches. Two operations are performed when a peer requests a file from the targeted peer. First, the targeted peer compares all of its block identifiers with the requested file identifier to produce a list of block information that contains the 49 coefficients of each block and returns the list to the requesting peer. Then, the requesting peer uses the list to obtain the blocks from the targeted peer. When requesting a file, the peer will choose the first indexer to reach the last access peer and the peer list that includes the coded blocks for the file. Because the last access peer may have collected sufficient blocks from the file, the requesting peer may directly obtain the blocks from the last access peer without further inquiry based on the peer list. However, if the requesting peer cannot obtain the necessary information from the last access peer, it will use the peer list to collect seven independent blocks. If the information is still insufficient, the peer will try to look up the other backup indexers. If these indexers cannot provide useful information, the peer will try again later until the available time has expired according to the timer. When it has successfully collected the seven independent blocks, the request peer will decode the blocks and store them in its LRU cache to facilitate future access by other peers. Finally, the indexer updates the request peer, the last access peer in its index. For a clear visual representation of the steps involved in this process, refer to the flow chart in Figure 7. The creation of new coded blocks is triggered by the index peers. As mentioned previously, the indexer keeps an index that records block identifiers associated with the information from the register peer (e.g., IP addresses and P2P identifiers) and the information from the peer that has most recently inquired about the block. Moreover, the indexer sets a timer for each register peer that is periodically reset by the register. If the timer expires, the register peer will be removed from the index.

Thus, the indexer can determine if the number of registered blocks is below the redundancy threshold. If the number is less than the threshold, the indexer will start creating a new block; it will determine the last access peer in its index. If the last access peer is not a register peer for the new block, the peer will be selected to store the new block. If the last access peer is a register peer for the new block, the indexer will randomly choose a non-register peer to save the new block. The selected peer next examines its cache to see if it has sufficient blocks; if so, the new block will be

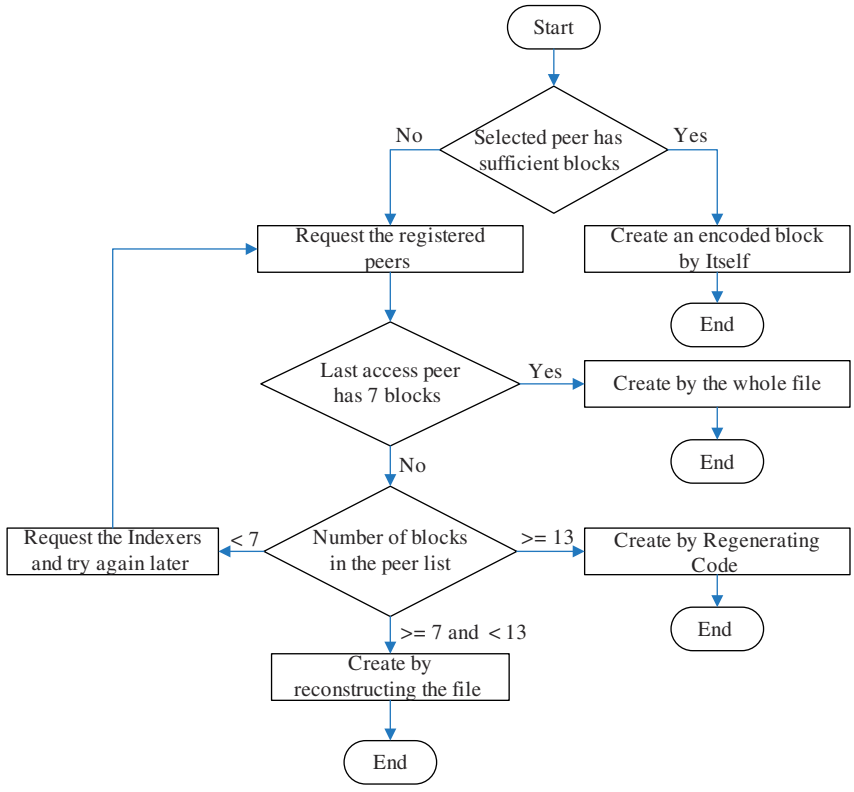

Figure 8. The flow chart for creating a new block.

created by itself; if not, the peer in question will inquire about the last access peer of its indexer. If the last access peer has seven blocks, then the new block will be created from the file reconstructed from the seven blocks. If the last access peer cannot create the file, the selected peer will refer back to the peer list. At that point, if the number of blocks collected from the peer list is less than 7, the selected peer will inquire again later, whereas if the number of collected blocks is less than 13 but greater than or equal to 7, then the new block will be created by reconstructing the file. Alternatively, if the number is greater than or equal to 13, then the new block will be created using the RC. Figure 8 shows the flow chart for creating a new block.

4. Experimental setup

We implemented our scheme in P2PSim (Gil *et al.* 2006), which is a discrete event packet-level simulator that can simulate structured overlay networks. The simulated network is comprised of 1024 peers that may alternately leave or join the network. The interval between successive events for each peer is exponentially distributed with a given mean time frame. In the experiment, each peer has a database, an LRU cache, and an index table. When a peer crashes or leaves, all stored data and indexes will be cleared. Conversely, if a peer rejoins, it will receive a different IP and use a different DHT identifier. There are 1000 different files of the same size in the system, and all files have the same access probability. At the beginning, we divided each file into seven blocks. Thus, there are 7000 raw blocks in total. Then, we used the RC to encode the 7 raw blocks into 14, 21, and 42 encoded blocks that were distributed to randomly selected peers for different levels of system redundancy. Then, we carried out the experiment using different cache sizes (8, 16, 32, 64, and 128 file blocks) and different levels of peer availability (90%, 65%, and 40%). Table 1 presents the probability that at least d peers of n peers will be active for different levels of peer availability. For example, the first row in Table 1 presents that, when peer availability equals 0.9 and at least 13 nodes out of 14 nodes which are the normal redundancy we want to maintain, then the probability to recover from a failed node is 0.5846 which is obtained from $C_{13}^{14} \times (0.9)^{13} \times (0.1)^{1} + C_{14}^{14} \times (0.9)^{14} \times (0.1)^{0}$. As indicated in Table 1, the feasible probability for the RC is row 1 and the last

Table 1. Probability associated with different levels of system redundancy and peer availability.

Peer availability	Value of n	Value of d	Probability
0.9	14	13	0.5846
0.65	21	20	0.0014
0.4	42	41	Close to 0
0.65	21	13	0.7059
0.4	42	13	0.9140

Table 2. Individual parameters for each peer availability level.

Parameter	Value		
Peer availability(%)	90	65	40
Time (hours)	6	(90/65)*6	(9/4)*6
Data redundancy (files)	2	3	6
Data redundancy (blocks)	14	21	42

two rows. Thus, we will choose the corresponding values to perform our experiments.

The results for two variables, peer availability, and cache size, are presented in Tables 2 and 3. The file availability is 99.9%. Without affecting the results, we first chose the level of peer availability and selected different cache sizes. The experiment took 6 h when the peer availability was 90%. To obtain data for 40% peer availability, we extended the experiment time to $6 \times 9/4$ h. Without decreasing the accuracy of our results, we were able to start collecting data during the second half of the experiment.

5. Experimental results

5.1. *Experimental results – access*

When accessing a file, the requesting peer will first check its LRU cache. If it does not have seven blocks, it will request the last access peer. If the last access peer has seven blocks in its LRU cache, the requesting peer will only require one connection to obtain the seven blocks. Tables 4–6 present the access performance levels for the different levels of peer availability assuming various cache sizes when the indexed peers carry different numbers of coded blocks. Although the cache size of 128 can obtain a better performance than the case size of 64, it consumes twice as much memory as the cache size of 64. By comparing the performance of various cache sizes for different levels of peer availability, we find that the cache size of 64 can almost

Table 3. Common parameters for all peer availability levels.

Parameter	Value
Cache size (file blocks)	8, 16, 32, 64, 128
Average requested blocks per peer	280
Target availability (%)	99.9

Table 4. The percentage of successful requests when peer availability = 0.9.

	Block number			
Cache size	The indexed peers have seven blocks (%)	The indexed peers have partial blocks (%)	The indexed peers have no blocks (%)	Other cases (%)
8	45.76	28.44	25.70	0.10
16	69.21	16.29	14.30	0.20
32	86.35	7.41	5.85	0.39
64	91.51	4.62	3.09	0.77
128	91.30	4.55	2.67	1.48

Table 5. The percentage of successful requests when peer availability = 0.65.

	Block number			
Cache size	The indexed peers have seven blocks (%)	The indexed peers have partial blocks (%)	The indexed peers have no blocks (%)	Other cases (%)
8	38.72	30.34	30.83	0.10
16	60.69	19.89	19.21	0.21
32	80.87	10.18	8.58	0.37
64	89.06	5.98	4.20	0.76
128	89.74	5.34	3.41	1.51

Table 6. The percentage of successful requests when peer availability = 0.4.

	Block number			
Cache size	The indexed peers have seven blocks (%)	The indexed peers have partial blocks (%)	The indexed peers have no blocks (%)	Other cases (%)
8	28.00	37.96	33.92	0.13
16	48.54	26.47	24.78	0.21
32	68.97	16.64	14.01	0.38
64	83.51	9.18	6.51	0.81
128	85.25	8.10	5.07	1.58

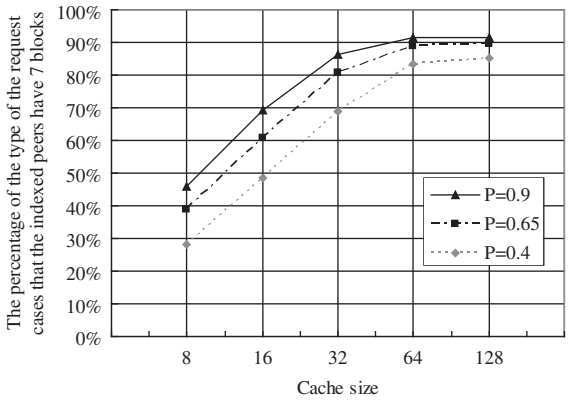

Figure 9. The percentage of successful requests for various peer availability levels and cache sizes for an indexed peer with seven blocks.

reach the performance of cache size of 128 by using only half of the memory. Thus, we choose it as the optimal cache size. The other cases in the three tables are those in which the requesting peer has seven blocks in its own LRU cache. For example, the first row in Table 4 presents that, when we conducted the experiment on the cache with eight blocks, 45.76% of queries would hit the indexed peers with seven blocks, whereas 28.44% of queries would hit the indexed peers between 1 and 6 blocks, and 25.7% would hit the indexed peers with no blocks. Other results were obtained in 0.1% of cases; that is to say, in those cases, the requesting peer had seven blocks in its own LRU cache. As expected, the possibility is very low that a requesting peer will have the seven required blocks in its own cache, regardless of the size of that cache. Figure 9 shows the percentage of successful requests for various peer availability levels and cache sizes for an indexed peer with seven blocks. As indicated in the figure, the optimal cache size is 64, as caches of this size produce an 83% success rate for queries. This means that, in our system, a requesting peer can obtain the required file by collecting the seven blocks in at least 83% of the time. Thus, our system outperforms the pure RC scheme.

In these experiments, we used an LRU cache to improve the access performance. As shown in the above tables, the best cache size is 64, which guarantees the access performance in 83% of the cases. Next, we compared the number of connections in our scheme with the numbers for the pure RC scheme. We only considered connections that yielded at least one block. In contrast, we ignored the connections required to search for information about peers and blocks. In the pure RC scheme, if a peer has one coded block of the file in its database, it will require six more connections

Table 7. Number of connections based on peer availability.

	Peer availability					
	0.9		0.65		0.4	
Cache size	Pure RC	RC/ Cache	Pure RC	RC/ Cache	Pure RC	RC/ Cache
8	6.99	4.22	6.97	4.61	6.92	5.14
16	6.98	2.70	6.97	3.18	6.90	3.85
32	6.99	1.69	6.97	1.97	6.89	2.60
64	6.98	1.44	6.97	1.57	6.88	1.89
128	6.98	1.39	6.97	1.45	6.88	1.70

Table 8. The percentage of new blocks created using various methods when peer availability = 0.9.

	Generating type			
Cache size	By the last access peer (method 1) (%)	By regenerating code (method 2) (%)	By reconstructing the file (method 3) (%)	By itself (method 4) (%)
8	44.95	52.21	0.38	2.47
16	68.92	28.12	0.20	2.77
32	86.08	10.96	0.07	2.89
64	91.59	5.15	0.04	3.22
128	91.58	4.41	0.03	3.98

to collect seven blocks. If it does not have at least one coded block of the file in its database, the peer will require seven connections. In our scheme, in contrast, contacting an indexed peer can help reduce the number of connections because the indexed peer may have more than one block. In the ideal case, the indexed peer has seven blocks, and only one connection will be required to obtain those seven blocks. Table 7 presents a comparison between pure RC and RC using caches for different system settings. As indicated in the table, the number of required connections decreases to less than 2 once the cache size is larger than 64, which is very close to the ideal value 1.

5.2. *Experimental results for generating redundant data*

As Figure 8 indicates, there are four methods that can be used to generate a new block, and we examined the distribution of these methods. Tables 8–10 present the percentage of new blocks created using various

Table 9. The percentage of new blocks created using various methods when peer availability = 0.65.

Cache size	Generating type: By the last access peer (method 1) (%)	By regenerating code (method 2) (%)	By reconstructing the file (method 3) (%)	By itself (method 4) (%)
8	35.99	57.88	0.01	6.12
16	58.03	35.30	0.01	6.66
32	78.57	14.81	0.00	6.61
64	86.67	6.52	0.00	6.81
128	87.58	5.08	0.00	7.34

Table 10. The percentage of new blocks created using various methods when peer availability = 0.4.

Cache size	Generating type: By the last access peer (method 1) (%)	By regenerating code (method 2) (%)	By reconstructing the file (method 3) (%)	By itself (method 4) (%)
8	27.57	67.38	0.01	5.04
16	50.50	45.49	0.00	4.01
32	72.18	24.74	0.00	3.08
64	87.85	9.65	0.00	2.50
128	90.16	6.90	0.00	2.93

methods at different levels of peer availability and with different cache sizes. Among other things, these tables indicate that the percentage is proportional to the cache size for method 1. These results confirm our expectations and justify the addition of a cache and index table to our system. Furthermore, the results show that the optimal cache is 64 for method 1. Figure 10 shows comparative data for method 1 for different levels of peer availability and cache sizes. Next, we compared the cost of creating a new block using the pure RC method versus the cost required using our scheme. (We measured cost in terms of repair bandwidth.) In the second paragraph of Section 3, we derived the value $13M/49$, which is the repair bandwidth for pure RC, whereas in our scheme, there are four methods of creating a new block, and the costs for methods 1, 2, 3, and 4 are $\frac{M}{7}$, $\frac{13M}{49}$, M, and 0, respectively. Thus, we were able to compute the cost as $P1 \times \left(\frac{M}{7}\right) + P2 \times \left(\frac{13M}{49}\right) + P3 \times M + P4 \times 0$, where $P1$, $P2$, $P3$, and $P4$ are the percentages in Tables 8–10. Table 11 presents the maintenance cost

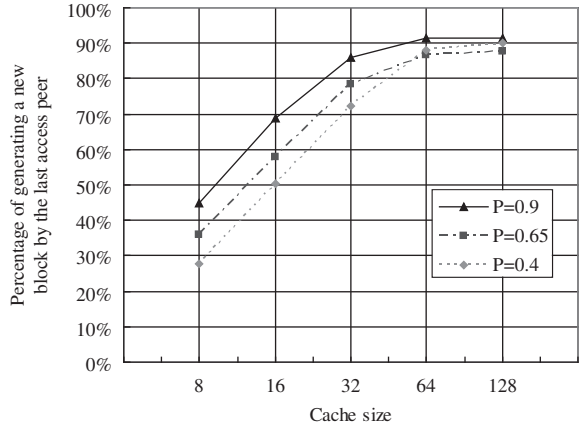

Figure 10. Percentages indicating how often a new block is created using the last access peer for various cache sizes and peer availability levels.

Table 11. Comparison of maintenance costs associated with pure RC and RC/cache methods.

Cache size	Peer availability: 0.9 Pure RC	0.9 RC/Cache	0.65 Pure RC	0.65 RC/Cache	0.4 Pure RC	0.4 RC/Cache
8	0.265	0.207	0.265	0.205	0.265	0.218
16	0.265	0.175	0.265	0.177	0.265	0.193
32	0.265	0.153	0.265	0.152	0.265	0.169
64	0.265	0.145	0.265	0.141	0.265	0.151
128	0.265	0.143	0.265	0.139	0.265	0.147

associated with the pure RC system and that associated with the RC system including caches. As indicated in the table, RC with cache has lower maintenance cost compared to pure RC.

6. Conclusions

Pure RC requires less repair bandwidth than three traditional redundancy schemes (replication, erasure coding, and hybrid schemes). Nevertheless, this system exhibits poor access performance. Hence, pure RC can only be used for archival storage or backup services.

To improve the access performance and reduce the maintenance cost, we added a cache to the system. We also added an index table that records the last access peer and the register peers. This system allows access at least 83% of the time, which means that regeneration coding need not be limited to archival storage or

backup services. Instead, it can be used for content services. This discovery is the main contribution of our research.

Although our scheme exhibits a significantly improved maintenance cost, it also helps to enhance the access performance. The experimental results show that an index table and an appropriately sized cache help to improve the request service, allowing the system to reach a performance level of no less than 83% when the cache size is 64.

7. Future work

In the future, we will attempt to explore other ways of reducing maintenance costs and for applying our work to multimedia content discovery and delivery network (mCDN) services, implementing a P2P storage system in that context.

Acknowledgments

This study was supported by the ITRI under Project Code 8352B11200.

Nomenclature

α_{MSR}	the block size stored at each node by using minimum storage RC
β	the packet size with which a newcomer communicates to any d surviving nodes
γ_{MSR}	the total repair bandwidth for MSR code
BitTorrent	a software application that uses P2P communications protocol for file sharing
Chord	a P2P protocol and algorithm for distributed hash tables
D	the number of connections to active nodes
DHT	a type of distributed system that provides hash table-like functionality
H(d)	the hash function for DHT based on the Chord protocol
IP	internet protocol
K	the number of storage nodes that can recover the original file
LRU	least recently used
M	the original file size
Mb	Megabit = 1,000,000 bits
mCDN	multimedia content discovery and delivery network
OceanStore	a distributed storage utility on PlanetLab
P2PSim	a multi-threaded, discrete event simulator to evaluate, investigate, and explore P2P protocols
PAST	a large-scale P2P persistent storage utility
Skype	a software application that allows users to make voice calls and chats over the internet
Total Recall	a P2P storage system

References

Ahlswede, R., *et al*., 2000. Network information flow. *IEEE transactions on information theory*, IT46 (4), 1204–1216.

Bhagwan, R., *et al*., 2004. Total Recall: system support for automated availability management. *In*: *Proceedings of the 1st symposium on networked systems design and implementation*, 29–31 March 2004 San Francisco, CA. Berkeley, CA: USENIX Association, 25–25.

Chen, G.H., *et al*., 2008. Redundancy schemes for high availability in DHTs. *Chinese journal of computers*, 31 (10), 1695–1704.

Dabek, F., *et al*., 2001. Wide-area cooperative storage with CFS. *In*: *Proceedings of the eighteenth ACM symposium on operating systems principles*, 21–24 October 2001 Banff, AB, Canada. New York: ACM, 202–215.

Dimakis, A.G., *et al*., 2010. Network coding for distributed storage systems. *IEEE transactions on information theory*, 56 (9), 4539–4551.

Fragouli, C. and Soljanin, E., 2007. *Network coding fundamentals*. Boston: Now Publishers.

Fragouli, C. and Soljanin, E., 2008. *Network coding applications*. Boston: Now Publishers.

Gil, T., *et al*., 2006. *P2Psim: a simulator for peer-to-peer protocols*. Available from: http://www.pdos.lcs.mit.edu/p2psim/.

Ho, T., *et al*., 2006. A random linear network coding approach to multicast. *IEEE transactions on information theory*, 52 (10), 4413–4430.

Jaggi, S., *et al*., 2005. Polynomial time algorithms for multicast network code construction. *IEEE transactions on information theory*, IT51 (6), 1973–1982.

Kubiatowicz, J., *et al*., 2000. Oceanstore: an architecture for global-scale persistent storage. *In*: *Proceedings of the 9th international conference on architectural support for programming languages and operating systems*, 12–15 November 2000 Cambridge, MA. New York: ACM, 190–201.

Li, S.Y.R., Yeung, R.W., and Cai, N., 2003. Linear network coding. *IEEE transactions on information theory*, IT49 (2), 371–381.

Lua, E.K., *et al*., 2005. A survey and comparison of peer-to-peer overlay network schemes. *IEEE communications surveys and tutorials*, 7 (2), 72–93.

Médard, M. and Koetter, R., 2003. An algebraic approach to network coding. *IEEE/ACM transactions on networking*, 11 (5), 782–795.

Merzbacher, M. and Patterson, D., 2002. Measuring end-user availability on the web: practical experience. *In*: *Proceedings of the performance and dependability symposium*, 23–26 June 2002, Bethesda, MD. Washington, DC: IEEE Computer Society, 473–477.

Rodrigues, R. and Liskov, B., (2005). High availability in DHTs: erasure coding vs. replication. *Lecture notes in computer science*, 3640, 226–239; *In*: *Proceedings of the 4th international workshop on peer-to-peer systems*, 24–25 February 2005 Ithaca, NY. Piscataway, NJ: IEEE Educational Activities Department, 226–239.

Rowstron, A. and Druschel, P., 2001. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *In*: *Proceedings of the eighteenth ACM symposium on operating systems principles*, 21–24 October 2001, Banff, Canada. New York: ACM, 188–201.

Sander, P., Egner, S., and Tolhuizen, L., 2003. Polynomial time algorithms for network information flow. *In*: *Symposium on parallel algorithms and architectures (SPAA)*, 07–08 June 2003 San Diego, CA. New York: ACM, 286–294.

Stoica, I., *et al*., 2001. Chord: a scalable peer-to-peer lookup service for internet applications. *In*: *Proceedings of the (2001) conference on applications, technologies, architectures, and protocols for computer communications*, 27–31 August 2001 San Diego, CA. New York: ACM, 149–160.

Yang, M. and Yang, Y.Y., 2008. Peer-to-peer file sharing based on network coding. *In*: *Proceedings of the (2008) the 28th international conference on distributed computing systems*, 17–20 June 2008 Beijing, China. Piscataway, NJ: IEEE, 168–175.