Achieving Reliability through Replication in a Wide-Area Network DHT Storage System *

Jing Zhao[§], Hongliang Yu[§], Kun Zhang[§], Weimin Zheng[§], Jie Wu[†], Jinfeng Hu[£]
[§]Department of Computer Science and Technology, Tsinghua University; [£]IBM CRL, Beijing
[†]Department of Computer Science and Engineering, Florida Atlantic University

Abstract

It is a challenge to design and implement a wide-area distributed hash table (DHT) which provides a storage service with high reliability. Many existing systems use replication to reach the goal of reliability. However, maintaining availability and consistency of the replicas becomes a major hurdle. A reliable storage system needs to recover lost and inconsistent replicas, but any recovery strategy will lead to extra workloads which affect the throughput of the system. This paper explores these problems and provides a possible solution. We argue that our approach not only keeps eventual consistency of replicas but also quickens the spread of updates. We use an adaptive recovery strategy to guarantee the reliability of replicas as well as bandwidth saving. With a simulation result better than epidemic algorithms, we have also implemented and deployed a DHT system using strategies mentioned in this paper, and integrated it into Granary - a storage system distributed in 20 servers in 5 cities. Granary and the DHT system have run over half a year and provide a reliable storage service to several hundred users.

Keywords: Availability, consistency, DHT, distributed storage, reliability, replication, peer-to-peer systems.

1. Introduction

Replication is an effective and convenient method used to guarantee the reliability of data in distributed storage systems. However, replication brings a new problem: we must guarantee the reliability of replicas. If a user makes an update to some data, all replicas of the data should be able to exhibit this change. When some applications have been updated while others still exhibit an old state, it is called *inconsistency*. Several factors may cause inconsistency of replicas, e.g., churn of DHT nodes, drop of update messages, and network partitions [1, 7, 17]. Replicas may also be lost. When nodes depart from the system, data stored on them will lose a replica. Some new nodes may join in the DHT system after that. If we do not move responding data replicas to these new nodes, data which should be stored on them will also seem to lose a replica.

For a replication storage system, the use of epidemic algorithms [5] is a traditional method of maintaining the reliability of replicas. One site selects another site randomly and periodically, and compares its own data's state with the other so as to determine if there is any loss or inconsistency. To decrease bandwidth cost, they use the hash value of replicas instead of the replica's contents. OpenDHT [12] and DHash [4] both use epidemic algorithms. They employ Merkle trees [9] and transfer the hash summary during the process of anti-entropy. For systems that do not have to handle frequent update operations, or systems that only require eventual consistency and can tolerate some update loss, this method works well. It can guarantee durability and the eventual consistency of data. However, some systems have to handle frequent update operations and require all these updates to be exhibited. For these systems, durability is not enough - they need to guarantee the availability and consistency of all replicas and be able to recover an inconsistent data set as soon as possible.

In this paper, we will provide an example of such a system. We target the reliability of replicas in the name of both availability and consistency. Epidemic algorithms cannot meet these needs well. Since a site chooses a remote site randomly and periodically for synchronization, many synchronization processes will happen between two sites with consistent data sets, which will extend the recovery time for those sites' lost replicas or new updates. Users may read null or inconsistent data from these sites. Therefore, during the recovery time, the availability of replicas will not be guaranteed. A long recovery time may lead to another problem: even if all the replicas exhibit the same eventual state in the end, some replicas may miss a few midterm states, that is, they cannot get a higher version of update (the update may be lost on the network and need to be re-

*This work is supported by NSFC under Grant No. 60433040 and No. 60603071.

covered) before the highest version comes first. In this situation, users may get jumping operation results. A smaller synchronization interval may quicken recovery speed, but that leads to higher bandwidth cost.

We have addressed the above problem in the Granary project [6]. Granary is a wide-area networked storage system. Users can upload their files onto the nodes of Granary, and can download or modify these files. Besides, users can use Granary to share their files with their friends. Granary can be used to backup operating systems or provide file sharing services. It contains a DHT storage system which is also distributed on wide-area networks, and stores three kinds of meta-data into it.

Granary uses replication to maintain the reliability of data, and its first kind of meta-data indicates which nodes of Granary are storing replicas of a certain file. Before users' files are located, Granary will fetch this kind of meta-data first to learn where users' files are located. These meta-data may be updated by upper-applications of Granary because Granary may migrate the replicas adaptively in order to boost performance and balance loads. The second kind of meta-data includes the information about Granary's users, including user names, passwords and quota information. Granary will fetch this kind of meta-data to check the user's authorization and whether the user has reached the upper limit of his/her quota. This kind of meta-data will be updated when users change their passwords, or upload/delete their files. The last kind of meta-data is users' catalog information, which represents structures of users' files. Granary must fetch this kind of meta-data when users want to list their files. This catalog meta-data is updated the most frequently, since Granary updates this kind of meta-data whenever users make any changes to their catalog, e.g., adding a new file or deleting an old one.

As stated above, we use the replication strategy to maintain the meta-data of Granary. The availability of replicas for the three kinds of meta-data is significant. The DHT system must guarantee that Granary can always get the newest replica, or Granary will behave abnormally or even become unusably. However, the meta-data is mutable, and the last two kinds of meta-data are updated frequently. Considering Granary's wide-area network environment, maintaining the availability and consistency of meta-data's replicas is a big challenge.

The DHT system deployed in Granary is called ConDHT, and we argue that our DHT system can meet the above needs and provide a reliable storage service which ensures data availability and consistency. The target size of the system is not huge - it only contains several hundred nodes. Nodes of ConDHT are relatively steady - they will not leave the system frequently like nodes in most peer-to-peer systems [1, 8]. Although this assumption avoids a high-frequency churn of nodes, we still have to handle problems brought by nodes' arrival and departure: we may add hosts to extend the size of ConDHT, and hosts may depart because of network problems, disk failures, power cut, or a number of other factors. ConDHT is deployed on wide-area networks and has similar characteristics to PlanetLab.

We will represent a recovery strategy in this paper, which we argue can find and recover lost and inconsistent replicas quickly. The strategy uses an adaptive algorithm which saves bandwidth and improves the system's throughput. Both simulation and practical results will be given. We have deployed the Granary system, including ConDHT on 20 machines located on networks across 5 different cities: Beijing, Shanghai, Guangzhou, Wuhan, and Hangzhou. Now that the system has safely run for over half a year, it is mainly used as a backup system and an alternative choice for U-disks on campus at the present time.

The rest of the paper is organized as follows: Section 2 details the architecture of ConDHT and the recovery strategy, as well as an adaptive algorithm which can maintain throughput of the system. Evaluations and measurements will be given in Section 3. Section 4 contains information regarding related works. Finally, we have a concluding remark in Section 5.

2. Architecture of ConDHT

Similar to many previous DHT systems [4, 14, 16], ConDHT assigns a random 128-bit node identifier to each node. The node can be seen as distributed in a 2^{128}-element circle, called nodeId space. For a value being stored into ConDHT with key k, the DHT hashes k (with a consistent hash function like SHA-1) into a 128-bit hashId h. ConDHT demands that the data with hashId h should be stored on a specific number (the replication number of the DHT storage system) of successor nodes.

The interfaces provided by ConDHT are simple, three operations are included: *put(key, value, version), get(key)* and *delete(key, version)*. ConDHT is a versioned storage system: each value owns a version number. Given the same key, a higher-version value will automatically replace the lower-version one. A user invokes the *put* interface and provides a higher version number when updating. A higher version number should also be provided when deleting some data - ConDHT handles *delete* requests similar to *put* requests. For convenience, in the following we call the data stored in a ConDHT system *DHT objects* or *objects*.

2.1. DHT Implementation

In this section, we will detail the implementation of the put interface of ConDHT. ConDHT maintains failing information during the put process, and uses the information for future recovery.

When invoking the *put* interface, a key-value pair and a version number must be provided. The user (here we use "user" to indicate the upper-applications of Granary which invoke ConDHT's interfaces) contacts a node of ConDHT which acts as a gateway node. Every node of ConDHT can act as the gateway. The gateway node is noted as node *G*.

Assume that we maintain *R* replicas for a DHT object. In the put process, Node *G* first calculates which *R* nodes should be responsible for storing the *R* replicas according to its routing table. Then it sends the object, including the version number, to these nodes. Upon receiving the object from node *G*, the receiver node uses its routing table to calculate which other *R-1* nodes should store this DHT object and record their nodeIds. When node *G* has received "ACK" messages from all the *R* nodes or timeout reaches, node *G* will send an message to these nodes, informing them of all the nodes' nodeIds which store the DHT object successfully. Thus nodes will know which nodes possibly failed to store the DHT object. They maintain this information into a local data structure named *target list*, and use the *target list* to recover replicas. We detail the recovery strategy in later sections.

We use asynchronous communication in the *put* interface, so that the *put* process is nonblocking. The *put* interface will return a receipt to the invoker, and the invoker can learn the number of successful storing nodes through the receipt. Users can customize the expected number of successful storing nodes. Therefore, users will not waste time waiting for the *put* process to return for slower nodes or delayed messages on network.

2.2. Recovery Strategy

The ConDHT system can handle four kinds of abnormal situations, including node's transient failure, message drop on networks, node's permanent failure and new node's joining. The former two situations may cause replicas' inconsistency and temporary loss, while the latter two will lead to replicas' loss. ConDHT must be able to find and recover the inconsistency and loss as soon as possible.

2.2.1 Push and Pull

As we have mentioned before, all nodes of ConDHT maintain a target list indicating which node lacks which DHT object during a *put* (or *delete*) process. For example, suppose node A lacks a DHT object O. Node B keeps this information in its target list and wants to do the recovery work. Suppose Node A's lack is caused by its transient departure. The system does not know when node A will come back. Thus node B has to ping node A and send it the key and the version number periodically until node A comes back. Then node B sends the object O to node A. All nodes which have object O will send this kind of message to node A periodically. We call this process the *"push"* process.

When node A comes back and receives these messages, it chooses a node with the highest-version object O to fetch it back. We call this the *"pull"* process. In the pull process, we can use delta-coding to send differences. But since the size of DHT object is not large, we send the whole object in our implementation. After node A has gotten object O, it broadcasts a message to all the recovering nodes, telling them that it has object O. Then all other nodes will delete the pair $\{A, O\}$ from their target lists.

This method records all the data loss and failure information in the first two abnormal situations, and recovers the glitches with little bandwidth cost, since most message's sizes are small. It can guarantee that the recovered replica is the newest one. Furthermore, even if some messages containing the recovery information are lost on network, the method still guarantees that the replica can be recovered at last. As we have shown above, the recovery work will not stop until node *A* declares that it has gotten the replica.

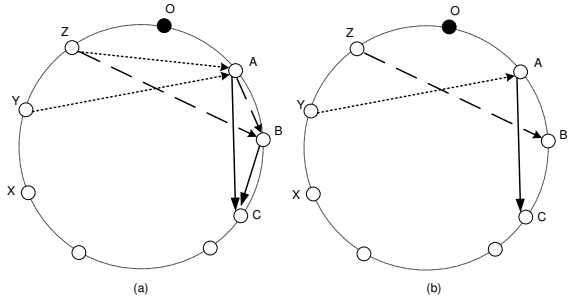

Figure 1. Two recovery methods after node O departs from the DHT.

2.2.2 Eager and Lazy

When a node departs from ConDHT permanently, each DHT object stored in it will lose a replica. Figure 1(a) depicts this situation. Assuming node O departs from the system, we maintain 3 replicas for each DHT object. Before node O's departure, any DHT object whose hashId falls into the region from node X (not included) to node Y (included, and we represent these DHT objects as O_{xy}) has a replica in node Y, Z, and O. After node O's departure, O_{xy} will be stored in node Y, Z, and A, thus node Y and Z will send a replica to node A.

ConDHT implements two methods to complete this task. The first method is shown in Figure 1(b). After node O departs from the DHT, for each DHT object only one node (the first successor) is responsible for the object's recovery work. For example, node Y will recover O_{xy} to node A,

node Z will recover O_{yz} to node B, and node A will recover O_{zo} to node C. The DHT objects will be sent directly. We call this recovery method an *eager recovery* method.

Another method is similar to the *push-pull* process which we have introduced above. As Figure 1(a) shows, node Y and node Z will add the recovery task of sending O_{xy} to node A into their target list first. Then the push-pull process and an adaptive algorithm which is stated in the next section will be used to finish the recovery work. When a gateway node tries to get the object from node A before it finishes pulling, node A will fetch the object and store the object locally. This is the *lazy recovery* method.

The *eager recovery* method is a quicker recovery method for recovery and can do the work well when message drop and network congestion is not severe. However, it may bring some bandwidth bursts. Since one node may store a large amount of DHT objects, to recover them in an eager way leads to a high bandwidth cost. Furthermore, the eager recovery method cannot guarantee that the recovered replica is the newest one. In the contrary, the lazy method can avoid the bandwidth burst and choose the newest replicas to fetch.

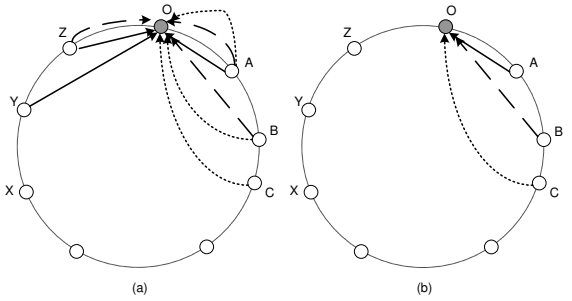

Figure 2. Two recovery methods after *node O* is added to the DHT.

Figure 2 depicts the scenario when node O is added into ConDHT. Some DHT objects should be migrated according to the semantic of DHT. Figure 2(a) and (b) depicts both the *lazy* and *eager recovery* methods. For *lazy recovery*, node Y, Z and A all send recovery messages to node O containing keys and versions of O_{xy}; after node O declares it has got all the replicas of O_{xy}, node A will delete them from its local database. For *eager recovery*, Node A sends replicas of DHT objects O_{xy} to node O; node B sends replicas of DHT objects O_{yz} to node O; and node C sends replicas of DHT objects O_{zo} to node O.

2.3. Throughput Maintenance

A reliability maintenance strategy will cause extra bandwidth cost since the storage system must detect inconsistency and data loss and try to solve it. Most bandwidth resources should be used by upper-level applications of Granary, which process users' files. Too much bandwidth cost in ConDHT may affect the throughput of Granary system, thus in ConDHT we must find a way to reduce bandwidth cost and improve throughput.

Sean Rhea et al. [13] had found that a reactive recovery may create a positive feedback cycle which will cause congestion collapse in network. Therefore they chose a periodical recovery strategy for Bamboo DHT. The discussion was focused on the routing table recovery, but it is also valuable for the storage application. If there is congestion in the network link between two nodes, object transfers during a recovery operation will surely exacerbate the bad situation. The main bandwidth cost should be the replicas' transfer in a recovery process. Knowing when to transfer DHT object replicas is important for the system's performance. We design an adaptive algorithm to determine the transferring time dynamically.

When we use the *push-pull* protocol to do the recovery work, replicas will be transferred during the *pull* process. Suppose node A receives messages indicating that it lacks a DHT object O. The default *pull* strategy of ConDHT is as follows:

1. Node A puts the id of object O and the nodeIds of nodes which have the highest version for object O into a local structure called *source list*.

2. Node A does *pull* operations periodically. Each time node A chooses a portion of its lacking DHT objects to pull according to its source list. Here, the interval between two pulls is dynamic.

Node A checks the recent history of the routing table first. If the routing table tells that few node activities have happened recently, it will conclude that the network condition is good, and choose a short interval. However, if node A finds frequent node changes, it will estimate the number of remaining replicas of the object. In the case that the number is still upon a threshold, node A will choose a relatively long period to pull the object, while if the number is below the threshold, which means the availability of the object is dangerous, the pull will be started immediately. This method is adaptive to node activities.

3. Evaluation and Measurement

In this section we will present the evaluation and measurement results of ConDHT. First, we compare our recovery strategy with the epidemic strategy on a simulation platform. Then we will give measurement results of the deployed practical system.

3.1. Simulation Results

We evaluate ConDHT on a discrete event-driven simulation platform. This simulation platform uses GT-ITM [18] to simulate the network topology, reads a one-year-long PlanetLab trace (from June 1st, 2005 to May 31st, 2006) collected by the CoMon project [10] supplemented with event logs from PlanetLab Central [11] as input, and simulates about 600 nodes' activities accordingly. ConDHT uses simple timeout to distinguish transient failures from permanent failures, which may cause unnecessary recovery work. However, the method presented in Carbonite [3] can also be integrated into our system, so as to reduce some needless recovery. We set the timeout value to 24 hours, that is, if a node has been out of reach for one day, it will be declared a permanent departure and recovery work will be triggered. A DHT object is put into ConDHT every 20 seconds, which has three sizes: 1KB, 5KB and 20KB. In one year's time, every DHT object will be updated about 20 times on average. 5 replicas are maintained for every DHT object. On this simulation platform we implement two types of recovery methods for ConDHT:

• Adaptive and Eager method. This implementation uses the push-pull recovery strategy. Since CoMon's records are collected on average every 5 minutes, if a node is out of reach temporarily, it will not come back until at least 5 minutes later. Thus in our simulation experiment every node invokes the push operation every 5 minutes. Nodes use the adaptive algorithm to determine when to pull lost DHT objects. After node activity, this implementation uses the eager recovery method, that is, an appointed node will move the corresponding DHT objects directly.

• Adaptive and Lazy method. This implementation employs the same recovery strategy but uses the lazy recovery method after node activity. The threshold during the pull process is set to 3.

We evaluate the bandwidth cost and recovery time in the simulation experiment. As a comparison, we also implement an epidemic synchronization protocol which uses Merkle trees. Two sets of simulation results will be given. One is produced when the synchronization protocol runs every 5 minutes, and the other is produced when the synchronization protocol runs every 10 minutes.

Figure 3 shows the cumulative distribution of bandwidth cost of the system in a year's time of 4 recovery strategies. From the figure we can see that the bandwidth cost will decrease when we choose a larger synchronization interval for the epidemic strategy. Epidemic strategies have the highest value of bandwidth cost - about 5 megabytes per second. This may be caused by the long recovery time: many lost or inconsistent DHT objects have been accumulated and are recovered once. This kind of bandwidth burst can be avoided by modifying the epidemic strategy a bit: when a lot of objects need to be recovered, they can be sent in several times. Both adaptive strategies cost less bandwidth, since they omit the data compared with the epidemic strategies. Compared with the lazy strategy, the eager one costs less bandwidth most of the time, but it has several bandwidth bursts. The highest burst reaches 3 megabytes per second, which can be a great threat to the system's throughput. This is because, after nodes' joining and departure, large amounts of DHT objects will be recovered and an eager recovery strategy chooses a proactive way to recover them. Since the lazy method uses the push-pull strategy which pushes versions and keys first, the bandwidth burst will be avoided, and the recovery time will be extended. This increases the bandwidth cost in the usual time, but the increase will not affect the throughput of the system.

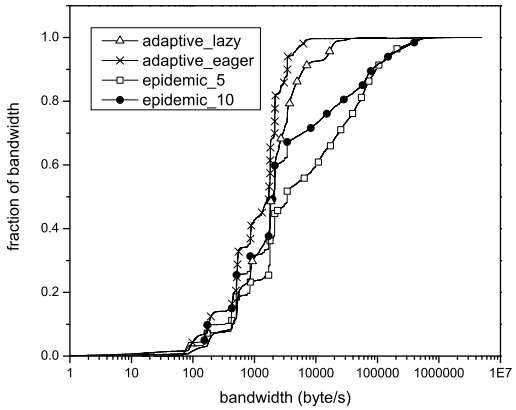

Figure 3. Cumulative distribution of bandwidth cost of 4 recovery strategies.

Figure 4 presents the recovery time used by the 4 recovery strategies. From the figure we can see that the adaptive eager strategy uses much less recovery time than other strategies. A small synchronization interval (5 minutes) for epidemic strategy can quicken the recovery, but it still costs over 15 minutes to repair inconsistency and data loss on average. This period is too long to protect users from getting inconsistent data. Even if we set a smaller synchronization period (which will cause a higher bandwidth cost), it is still possible that a node will take much time synchronizing with other normal nodes before it synchronizes with an abnormal node with inconsistent data set. The adaptive and lazy strategy also takes a lot of time to finish the recovery process. This is because it extends the recovery time for nodes' activities. As a result of the large amounts of DHT objects to recover after nodes' activities, the average recovery time of lazy strategy is long. However, even if the recovery work

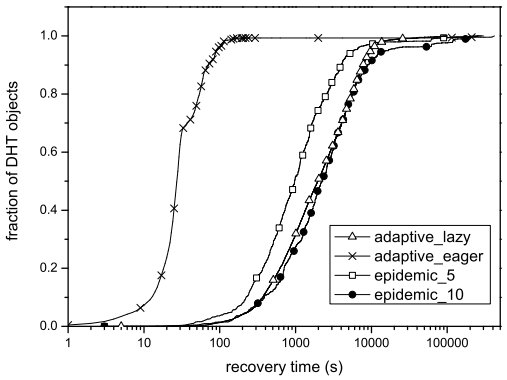

Figure 4. Cumulative distribution of recovery time for DHT objects of 4 recovery strategies.

has not finished, users can still get the most recent replicas from all the nodes. The push process is done in a short time after a node's departure or joining. Thus, although some nodes may have not finished the pulling work, they can still fetch the requested objects from remote nodes according to the information stored in their source lists.

	1	2	3	4	5	Prob.
adaptive_eager	0	0	4	46	49950	99.9%
adaptive_lazy	0	0	7	105	49888	99.776%
epidemic_5	0	0	1	61	49938	99.876%
epidemic_10	0	0	38	380	49582	99.164%

Table 1. Proportion of DHT objects with full replicas among 50000 DHT objects. Each column indicates the number of DHT objects maintaining the specific number of consistent replicas.

Table 1 represents the final state of DHT objects stored in ConDHT. We trace 50,000 DHT objects for each recovery strategy. For the adaptive and eager recovery strategy, 99.9% (49,950 out of 50,000) DHT objects maintain full and consistent replicas, and for the adaptive and lazy recovery strategy, the percentage is 99.776% (49,888 out of 50,000), a little bit smaller than the former one caused by a longer recovery time: several replicas have not been recovered yet. Those incomplete and inconsistent replicas will still be recovered since they are recorded in target lists. Besides, when users try to get a DHT object from a node storing an inconsistent replica, ConDHT will fetch the correct one from remote nodes according to the node's source list.

3.2. Deployed ConDHT System

Besides the simulation experiment, we have also conducted evaluations and measurements on the deployed practical ConDHT system. The practical ConDHT is implemented with over 20,000 lines of Java code and is deployed on 20 machines across networks over 5 cities in China. Among these 20 machines, 5 machines are configured with two Pentium Xeon CPU (2.8 GHz), 3.5 GB memory, two 250 GB SATA disks and a 100 Mbps full-duplex Ethernet connection, and the other 15 machines are configured with two 3.0 GHz Pentium 4 CPU, 2.0 GB memory, two 250 GB SATA disks and a 100 Mbps full-duplex Ethernet connection. The server's OS is Linux, and the kernel version is 2.6.12. The network latency between hosts in the same city is less than 10 milliseconds, as opposed to the several hundred millisecond latency between hosts in different cities.

¿From the above subsection, we can learn that the adaptive recovery strategy can greatly quicken the recovery process. In the practical ConDHT system, we employ this strategy. To avoid bandwidth bursts caused by the eager recovery strategy during node activities which may affect the throughput of the system greatly, we finally choose the lazy recovery strategy to deal with node activities. Five replicas are maintained for each DHT object, and the interval of the push operation is 1 minute. We set the threshold number to three in the pull process. In the pull process, we set an upper limit for the bandwidth cost. Thus, in every pull operation, the node can pull 50 DHT objects at most.

ConDHT has stored a rather large quantity of meta-data as DHT objects for Granary. At present, our deployed Granary system has about 300 registered users, and the number is increasing everyday. Each user is allowed to store at most 2 gigabytes worth of files in the system. Up to now, each user has been maintaining about 800 MB worth of files onto their spaces on average. A typical user's space may contain personal documents, together with some multimedia files. These files will generate about 100 DHT objects representing the files' location, and about 10 DHT objects representing catalog information. Besides, the information of each Granary's registered user will correspond to one DHT object for his username, password and quota information. Table 2 shows the numbers of different types of DHT objects.

Type of DHT Objects	Number of DHT Objects
location of files	28685
catalog information	3270
user-info	281

Table 2. Statistic of DHT objects stored in ConDHT.

ConDHT handles frequent put/update operations. When

a user of Granary uploads a single file onto his/her space, a DHT object for the location of the file will be generated and put into ConDHT. Besides, two other objects will be updated: one for the user's catalog information and another for his/her quota. If the user uploads a whole directory which contains files and sub-directories, many more DHT objects will be put/updated. For instance, 307 catalog DHT objects and 613 location DHT objects will be put/updated when we upload the source code directory of Granary project (containing the cvs files) onto Granary. These frequent put/update operations can easily cause inconsistency and data loss on the wide-area network environment of ConDHT.

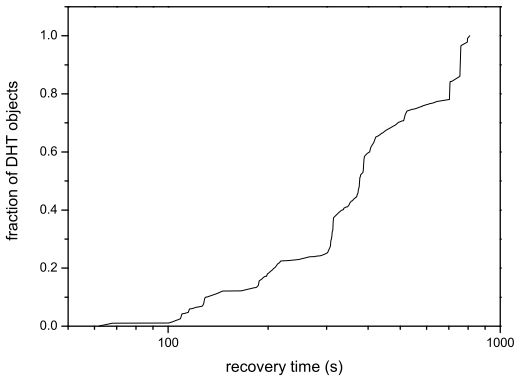

Figure 5. Cumulative distribution of recovery time in the practical ConDHT system.

Figure 5 depicts the recovery time of the practical ConDHT system. The statistical data is collected from two weeks' log of ConDHT system (from October 1st, 2006 to October 14th, 2006). In two weeks' time there is no node joining or departure, but there are still inconsistent and incomplete replicas generated during common put/update operations. As the figure shows, all of the recovery work is done in less than 15 minutes.

We have encountered permanent node failures during Granary's running. On November 3rd, 2006, two sets of BDB files each containing 12,500 DHT objects stored on two nodes were damaged due to misoperation during the system update. Thus, the Granary system had to handle the two nodes as permanent departure nodes. Figure 6 shows the bandwidth costs of a node which should pull damaging replicas before and after two nodes lost their data permanently in two days' time. After nodes' departure, ConDHT took 30 seconds to finish the push process. In this period, some meta-data might not be received from some nodes since the data had not been migrated to them yet. But after the push process, these nodes could know where to fetch the data from remote nodes according to their source lists. Therefore although the whole pull process lasted about 19 hours, there was no influence from users. A long pull time can avoid bandwidth bursts. From Figure 6 we can see that during the recovery work, the bandwidth cost was always around 70 KB/s, which is not a big burden for the system. This upper limit of bandwidth cost is due to the upper limit of the number of DHT objects in each pull operation.

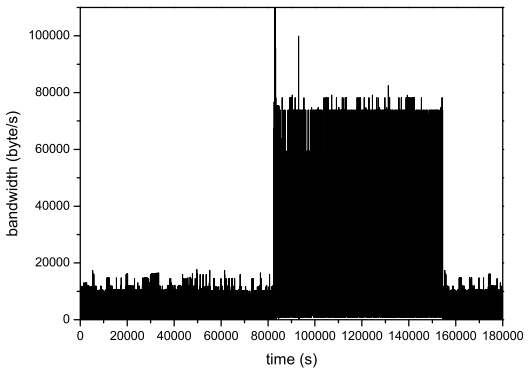

Figure 6. Bandwidth cost of a recovering node when two nodes lost their data permanently.

4. Related Work

OpenDHT [12] is a public DHT service which is now deployed on PlanetLab. OpenDHT provides a *put* and *get* interface. Before a user want to update his/her file, he/she must remove the old file first, then upload the new one. OpenDHT uses epidemic algorithm for replica synchronization. When inconsistency of replicas happens, the conflict resolution procedure is provided by users. The system assumption simplifies replicas' consistency problem. OpenDHT focuses more on how to provide a general DHT service and a simple interface, thus its storage application is less flexible. OpenDHT only provides eventual consistency for its data. In the case of network partitions or excessive churn, OpenDHT may fail to return values that have been put or continue to return values that have been removed. Imperfect clock synchronization in the DHT may also cause values to expire at some replicas before others, leaving small windows where replicas return different results. Thus, when we develop a system where availability and consistency of replicas are significant, OpenDHT's synchronization strategy is not enough.

DHash++ [4] uses a similar reliability and consistency strategy with OpenDHT, except it uses an erasure code strategy to realize redundancy. The erasure code is a good method of maintaining reliability, but it generates too much workload when handling lots of read and update requests.

Total Recall [2] is a file storage system which can be built on DHT. It provides a dynamic and hybrid way to guarantee reliability. Total Recall uses a lazy strategy and erasure coding to maintain large files, and an eager strategy and replication to maintain small files. Total Recall uses these approaches to decrease workloads of redundancy and repair mechanisms. We have borrowed some ideas from Total Recall's strategies. In Total Recall, an update is not successful until all the replica hosts acknowledge their writes to their local databases. Thus the system does not need to maintain the consistency of replicas after an update like ConDHT.

Proactive replication [15] is a way to guarantee the durability of data. Before any glitch happens, the system does some recovery work forwardly so that when a glitch happens, a bandwidth burst can be avoided. This method works well for seldom-updated storage services, but when the update process is frequent, proactive replication will be inapplicable.

5. Conclusions

In this paper, we have shown methods which were used to achieve the reliability in a distributed storage system built on DHT. Our system, ConDHT, is designed, implemented and deployed on wide-area networks, and disposes frequent update requests. We argue that the recovery strategy of ConDHT can increase the speed of recovery and guarantee the availability and consistency very well for replicas of the meta-data stored on it. The adaptive algorithm which we use can decrease the bandwidth cost and maintain the throughput of the system.

We have evaluated our recovery strategies on a simulation platform and get better results compared with epidemic ones. More important, we have introduced how we currently deploy this system. Measurement results from the deployed system are also given, which may be helpful to related researches. The system is running stably and we plan to extend it to a larger scale in the near future.

References

[1] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb, 2003.

[2] R. Bhagwan, K. Tati, Y. C. Cheng, S. Savage, and G. M. Voelker. Total Recall: System support for automated availability management. In *Proc. of the USENIX Symposium on Design and Implementation (NSDI)*, San Francisco, CA, 2004.

[3] B. G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *Proc. of the USENIX Symposium on Design and Implementation (NSDI)*, San Jose, CA, 2006.

[4] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proc. of the USENIX Symposium on Design and Implementation (NSDI)*, San Francisco, CA, 2004.

[5] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry Epidemic algorithms for replicated database maintenance. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 1987.

[6] Granary Project Homepage, 2006 http://hpc.cs.tsinghua.edu.cn/granary/granary.html

[7] K. P. Gummadi, S. Saroiu, and S. Gribble. A measurement study of Napster and Gnutella as examples of peer-to-peer file sharing systems. Multimedia Systems Journal, 9(2):170-184, Aug. 2003

[8] K. P. Gummadi, R. J. Dunn, S. Sariou, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, NY, USA, Oct, 2003.

[9] R. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Proceedings of the Annual International Cryptology Conference (CRYPTO)*, pages 369C378. Springer-Verlag, 1988.

[10] K. S. Park, and V. Pai. CoMon: a mostly-scalable monitoring system for PlanetLab. In *ACM SIGOPS Operating Systems Review 40*, 1 (Jan. 2006), 65-74. http://comon.cs.princeton.edu/.

[11] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. of the 1st HotNets Workshop*, Princeton, NJ, Oct, 2002. http://www.planet-lab.org.

[12] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. of SIGCOMM'05*, Philadelphia, PA, Aug. 2005.

[13] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proc. of the USENIX Annual Technical Conference*, June 2004.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware'01*, Heidelberg, Germany, Nov. 2001.

[15] E. Sit, A. Haeberlen, F. Dabek, B. G. Chun, H. Weatherspoon, R. Morris, M. F. Kaashoek and J. Kubiatowicz. Proactive replication for data durability. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Santa Barbara, CA, February 2006.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of SIGCOMM'01*, San Diego, CA, Aug, 2001.

[17] H. L. Yu, D. D. Zheng, B. Y. Zhao, and W. M. Zheng. Understanding User Behavior in Large-scale Video on Demand Systems. In *Proc. of the ACM Eurosys Conference (Eurosys)*, Leuven, Belgium, Apr, 2006.

[18] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee How to Model an Internet. In *Proceedings of IEEE INFOCOMM*, 1996.