Title
Resource Reclamation in Distributed Hash Tables

Authors
Tati, Kiran
Voelker, Geoffrey M

Publication Date
2006-07-27

Peer reviewed

Resource Reclamation in Distributed Hash Tables

Kiran Tati and Geoffrey M. Voelker
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{ktati, voelker}@cs.ucsd.edu

1 Introduction

Distributed hash tables (DHTs) are increasingly being proposed as the core communication and storage layer for distributed services in the Internet. This trend spans many traditional distributed applications, including file, storage, and archival systems [2,11,13,15], content delivery systems [3,6,8], databases [9], mail servers [7], messaging systems [10], and distributed naming services [12, 16]. DHTs map keys in a large, virtual ID space to associated values stored and managed by individual nodes in an overlay network. Since DHTs span many unreliable nodes in a wide-area environment, network and node failures are a common occurrence. To cope with such failures, previous work has explored mechanisms and policies for using redundancy to mask failures as users and applications allocate and update data in the DHT.

In this paper, we discuss the problem of storage reclamation in DHTs that use redundancy to provide high data availability. Two general approaches to managing data redundancy in DHTs have emerged, *eager repair* and *lazy repair*. For data stored in the DHT under a particular key, the eager approach maintains redundant data on n successors of the node that "owns" the key [4, 5, 15]. Each node in the system runs a background stabilization process that synchronizes its data with its n successors. This stabilization process conveniently handles both redundancy maintenance and reclamation simultaneously. Nodes propagate redundant data to a successor if the successor does not have it. Similarly, nodes implicitly propagate deletions to their successors so that successors can determine when they can remove the redundant data they store. As a result, the time it takes for deletions to propagate from the owner to all successors is typically n rounds of the stabilization process. Since rounds are short, eager repair will quickly delete all redundant versions of data stored at the owner node and its successors. Note, though, that eager repair implicitly assumes that successors will have the capacity to store redundant data.

Compared with eager repair, lazy repair uses additional redundancy to mask temporary node departures [2]. Eager repair immediately reacts to node arrivals and departures by propagating redundant data to new successors. Since node churn makes departures and arrivals very common, lazy repair uses additional redundancy to delay the propagation of redundant data to new nodes until many nodes storing the redundant data have simultaneously departed. As a result, lazy repair reduces the communication overhead of maintaining redundant data for availability in the face of high node churn by decoupling redundancy maintenance from node departures.

Lazy repair uses a level of indirection to store redundant data. When storing data at a key's owning node, lazy repair creates and stores redundant versions of the data on random nodes in the system. It records and tracks the nodes storing redundant data in a metadata structure stored at the key's owner. To ensure that the metadata is also highly available, lazy repair uses eager repair just for the metadata.

To delete data in a system using lazy repair, an application will send a request to delete data associated with a key to the key's owning node. The owner will then mark the data as deleted and subsequently delete all redundant versions of this data on storage nodes. Once storage nodes delete the redundant

data, the owner can complete the deletion operation by deleting the metadata as well. However, since lazy repair tolerates the temporary departure of storage nodes, storage reclamation is more difficult and requires additional overhead to completely delete all redundant versions of data than eager repair. It requires communication overhead to track when storage nodes return, and it requires storage overhead to maintain metadata until the deletion completes.

Two approaches for performing storage reclamation for lazy repair are active polling and implicit garbage collection. With active polling, a key's owner polls storage nodes until they return to the system, or until a timeout declares the node departed forever. Active polling uses communication overhead to reduce storage overhead by deleting it quickly. With garbage collection, storage nodes locally timeout and delete redundant data, reclaiming storage resources. Garbage collection requires less communication overhead, but deleted data persists longer in the system. In the rest of this paper, we compare the tradeoffs of these two methods.

2 Active Polling

In active polling, the key's owning node handles the application-level delete request. The owner is then responsible for deleting the underlying redundant data used to store the application data and provide high availability. Once the storage nodes have deleted their data, the owner then deletes the metadata. The owner attempts to contact all storage nodes to perform the delete. Storage nodes in the system perform the delete immediately. For storage nodes not in the system, the owner polls them until they appear in the system again or until a timeout declares those nodes permanently departed.

The cost to remove data using this approach has four components: informing the storage nodes to delete the data, polling departed storage nodes until they appear again, using eager repair to maintain the metadata during polling, and removing the metadata to complete the delete.

The polling and metadata maintenance costs are zero if all storage nodes are in the system when an application removes a file. Since all storage reclamation approaches need to remove the metadata to complete the delete, we do not consider it further in the comparison. Hence, the owner needs to send at least one message to all storage nodes for the redundant data and successor nodes storing copies of the metadata. For temporarily departed nodes, the owner also needs to send messages to poll those nodes until they arrive back in the system.

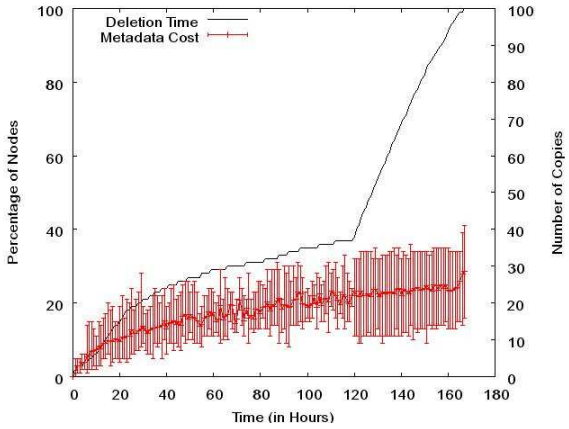

Figure 1: Time to delete data on storage nodes, and number of metadata copies made during the deletion process.

To quantify the overhead of using active polling, we simulated file allocation and deletion in the TotalRecall file system [2]. We used availability measurements from the Overnet peer-to-peer system to model node churn [1], and we used the data creation and deletion statistics of an NFS workload to an instructional server [14] to model file system activity. We created file system traffic from the above statistics for first two days. We started with 32GB data and each day we created 32GB of data, and during each day the workload deletes 39% of the data and overwrites 49% of the data uniformly throughout the day.

We then tracked the time required to fully delete files from the system. Figure 1 presents these results. The x-axis is the time required to remove data on storage nodes and then the metadata on the owner for a file. The y-axis on the left is the percentage of deleted files. A point (a, b) on the curve "Deletion-time" indicates that b percentage of files required a hours to complete the deletion. Overall, only one percent of files are removed immediately, and 18% took less than one day to remove the file. For 60%

of the files the owner times out and declares the remaining storage nodes to have departed the system forever. Note that we present the time to remove files rather than the number of messages to decouple the results from any specific polling mechanism.

Until the owner removes all underlying redundant data used to store a file, it must maintain the metadata representing the file. Assuming the use of eager repair to maintain metadata, maintenance incurs a communication cost to make replicas across successors during host churn. To quantify this cost, we measured the number of replica copies made to maintain metadata during the simulation experiment above. The curve labeled "Metadata Cost" in Figure 1 shows the results of these measurements, also as a function of time. The y-axis on the right indicates the number of metadata copies made as the owner waits for all storage nodes to delete their file data. A point (a, b) on the curve indicates that a file that required a hours to delete its data required b copies to maintain the availability of its metadata during the deletion process. Since for a given deletion time the number of metadata copies varies depending on the underlying host churn, we show an error line for each point. The bottom of each error line shows the minimum number of copies required, the top shows the maximum, and the middle shows the average.

Longer deletion times require more metadata copies. On average, deletions that take a day require 11 copies. Likewise, deletions that span a week require 28 copies. Overall, 74% of files required more than two days to delete data from all storage nodes and at least 16 metadata copies.

3 Garbage Collection

With garbage collection, when the owner receives the application-level delete request it removes the metadata associated with the file immediately but does not contact the storage nodes. Instead, the storage nodes decide to garbage collect deleted data on their own based upon existing communication between owners and storage nodes. When using lazy repair, owners poll their storage nodes to track the overall availability of a file. When the number of available storage nodes drops below a threshold, the owner then repairs data lost to unavailable nodes by propagating redundant data to new storage nodes.

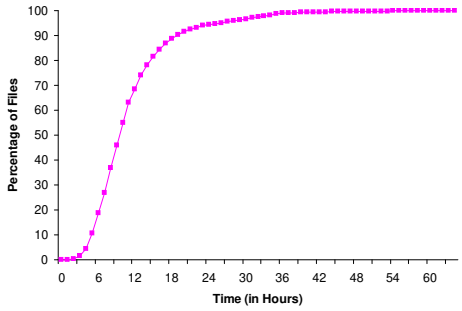

Figure 2: Time to repair.

Garbage collection can take advantage of these availability checks to further reduce communication costs. As a result, a storage node can assume that, if it does not receive an availability probe from the owner within an expected time frame (e.g., a small multiple of the probe interval), then the file has been deleted and the storage node can reclaim the storage used for the file. If the storage node mistakenly deletes the data because availability probes are severely delayed or repeatedly lost, then the owner will already consider the storage node unavailable with respect to file availability since the probe failed. Note that the owner can still reconstruct the file from other storage nodes.

To estimate timeout values that storage nodes can use to locally decide that a file has been deleted, we simulated the behavior of using lazy repair in TotalRecall to maintain file availability using the Overnet workload to model host churn. We uniformly distributed 3,600 files during the first day of the simulation, and for each file measured the time for host churn to cause file availability to drop below the lazy repair threshold.

Figure 2 shows the cumulative distribution of time to invoke lazy repair across all files. The x-axis shows the time to repair, and the y-axis shows the percentage of files repaired. The Overnet trace exhibits considerable host churn, and as a result we see that roughly 95% of files require repair within one day and all files require repair within three days. These results suggest that storage nodes can use a long timeout to decide to garbage collect data when they have not received an availability probe from the

owner.

With garbage collection, owners delete file metadata and stop probing storage nodes, and storage nodes eventually timeout and garbage collect the data they store for the file. During this timeout period, the disk storage used for data stored for a deleted file cannot be reused for other files, and represents a temporary storage overhead until garbage collection reclaims it. The amount of storage overhead is the rate at which applications are deleting files times the garbage collection timeout — the garbage collection timeout effectively buffers deleted data. This storage overhead is only an issue if the storage node is at capacity. If it is, then it can refuse requests to store new data and those requests will go to other storage nodes. If the entire system is at capacity, then this storage overhead will cause the system to reach capacity sooner than otherwise. However, it does not reduce the total amount of data that the system can store since the storage will be reclaimed.

4 Discussion

With active polling, owner nodes spend their resources only for deleted files. However, one issue with active polling is that owners have to distinguish between nodes that have left temporarily departed the system (gone offline) and "dead" nodes that have permanently departed. In peer-to-peer environments, it is difficult to make this distinction for all nodes. When using timeouts to declare that a node is dead, two issues arise. The owner might mistakenly declare that a storage node is dead when it has not, in which case the data on the storage node will not be reclaimed (although the owner will still remove the metadata) unless the storage node uses garbage collection or synchronizes with the owner. Or, the storage node has permanently departed and the owner unnecessarily consumed bandwidth by polling the dead node until it timed out.

With garbage collection, the system does not have to determine whether storage nodes have permanently departed since storage nodes are responsible for reclaiming space. Since peer-to-peer environments typically exhibit considerable host churn, dead nodes are not rare when deleting data. From the simulation of file system activity above, dead nodes have 15% of all file data. With garbage collection, the system does not expend resources on this data to remove it. Garbage collection also reduces communication costs since owners remove metadata immediately rather than maintaining it until all storage nodes have deleted the data. Since lazy repair depends upon owner nodes and storage nodes remaining in contact for the purposes of evaluating file availability, lack of contact can indicate that the file has been deleted and the storage node can garbage collect it. As a result, garbage collection is preferable for reclaiming storage in DHT-based systems that use lazy repair to provide high availability.

References

[1] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[2] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. Totalrecall: System support for automated availability management. In *ACM/USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2004.

[3] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[4] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cf cfs. In *Proceedings of the 18th SOSP (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[6] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with coral. In *Proceedings of the 1st*

USENIX Symposium on Networked Systems Design and Implementation (NSDI '04), San Francisco, California, March 2004.

[7] J. Kangasharju, K. Ross, and D. Turner. Secure and resilient p2p e-mail: Design and implementation. In *Third International Conference on P2P Computing*, Linkoping, Sweden, September 2003.

[8] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[9] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P file-sharing with an Internet-scale query processor. In *30th International Conference on Very Large Data Bases (VLDB)*, Aug. 2004.

[10] A. Mislove, A. Post, C. Reis, P. Willmann, P. Druschel, D. S. Wallach, X. Bonnaire, P. Sens, J.-M. Busca, and L. Arantes-Bezerra. Post: A secure, resilient, cooperative messaging system. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Lihue, Hawaii, May 2003.

[11] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[12] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the internet. In *proceedings of ACM SIGCOMM*, Oct. 2004.

[13] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *2nd USENIX Conference on File and Storage Technologies*. USENIX, Mar. 2003.

[14] D. S. Roselli. *Long-term File System Characterization*. PhD thesis, University of California, Berkeley, 2001.

[15] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th SOSP (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[16] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the web from dns. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.