Distributed Data Storage for P2P and Ad-hoc Networks

Robert Morris, M. Frans Kaashoek, Motonori Nakamura†, and Tomohiro Inoue

Abstract

Future ubiquitous services will be provided in ubiquitous network environments like peer-to-peer (P2P) or ad-hoc networks and will require highly efficient storage systems that are available anytime and anywhere. This paper presents an overview of one such storage system, DHash/Chord, which was developed at MIT LCS within the framework of the MIT-NTT collaboration. We also overview another storage system, CAOSS, being developed at NTT that is flexible in managing the consistency of data in ad-hoc networks environments. They make it easy to provide ubiquitous services and will be effective if used together.

1. Introduction

One of the key concepts of future network services is ubiquitous services, which provide various information and physical interactions to users anywhere and anytime. Ubiquitous services will be provided in a variety of network environments, such as peer-to-peer (P2P) networks and mobile ad-hoc networks. P2P networks are overlay networks over the Internet, and generally consist of user nodes throughout the entire Internet and their logical links. Mobile ad-hoc networks are collections of user nodes with wireless devices, and each user node forwards packets that are destined for other nodes using ad-hoc routing protocols. P2P networks can be formed over mobile ad-hoc networks. User nodes of P2P networks or mobile ad-hoc networks are free to join or leave these networks, so these networks appear anytime and anywhere user nodes exist and can communicate with each other. However, these network topologies tend to change frequently, and one cannot assume that any fixed nodes exist in these networks. We call these dynamically changing networks “ubiquitous networks”.

In ubiquitous networks, we cannot use a fixed storage server, so we need storage systems that are available anywhere and anytime. Ubiquitous storage systems must be available even when some nodes or links in the ubiquitous network fail. One ubiquitous storage system is DHash/Chord, which is being developed mainly for P2P networks. It provides a highly available storage system in ubiquitous network environments. Part of DHash/Chord was developed within the MIT-NTT collaboration framework.

Another ubiquitous storage system is being developed in NTT Network Innovation Laboratories for ad-hoc networks and for flexible data consistency management. It uses a refined version of the primary copy replication technique to achieve strict consistency in distributed data.

In this paper, we overview these two ubiquitous storage systems, which help to make ubiquitous service provision easy. They also work effectively if applied together.

† NTT Network Innovation Laboratories
Musashino-shi, 180-8585 Japan
E-mail: motonori@ma.onlab.ntt.co.jp

2. DHash/Chord

DHash/Chord uses a distributed hash table to store the relationships between keys and their values in a distributed manner. DHash stores and retrieves uniquely identified data blocks and handles their distribution, replication, and caching. To locate blocks, DHash uses Chord [1], which is a P2P lookup service

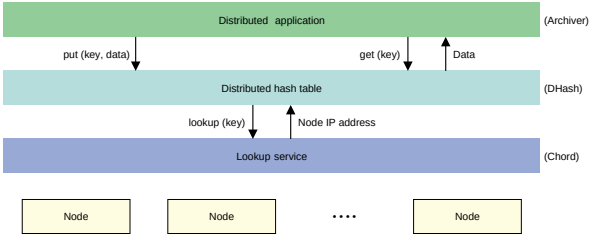

Fig. 1. Position of the distributed hash table.

that finds a node holding the value identified by a given key (Fig. 1).

In this section, we first review Chord and then move onto DHash using Chord. We can use DHash/Chord as a building block of a file storage system. For example, Ivy [2] and CFS (Cooperative file system) [3] provide NFS-like file system interfaces to users. (NFS: Network File System)

2.1 Chord

Chord supports just one operation: given a key, it will determine the node responsible for that key. Chord does not store keys or values itself, but provides primitives that allow higher-layer software to build a wide variety of storage systems.

(1) Consistent hashing

Each Chord node has a unique *m*-bit node identifier (ID), obtained by hashing the node's IP address. Chord views the IDs as occupying a circular identifier space. We call this circular space a "Chord ring" (Fig. 2). Keys are also mapped into this Chord ring, by hashing them to *m*-bit key IDs. Chord defines the node responsible for a key to be the successor of that key's ID. The successor of an ID is the node with the smallest ID that is greater than or equal to it (with wrap-around), much as in consistent hashing [4]. The implementation of Chord uses SHA-1 for the consistent hash algorithm and a 160-bit-wide Chord ring.

Consistent hashing lets nodes enter and leave the network with minimal movement of keys. To maintain correct successor mappings when a node *n* joins the network, certain keys among those previously assigned to *n*'s successor become assigned to *n*. When node *n* leaves the network, all of its assigned keys are reassigned to its successor. No other changes in key assignment to nodes are needed.

Consistent hashing is straightforward to implement, with constant-time lookups if all nodes have an up-to-date list of all other nodes. However, such a system does not scale. Chord provides a scalable, distributed version of consistent hashing.

(2) Chord's lookup algorithm

A Chord node uses two data structures to perform lookups: a *successor list* and a *finger table*. Only the successor list is required for correctness, so Chord is careful to maintain its accuracy. The finger table accelerates lookups, but does not need to be accurate, so Chord is less concerned about maintaining it. The following discussion first describes how to perform correct (but slow) lookups with the successor list, and

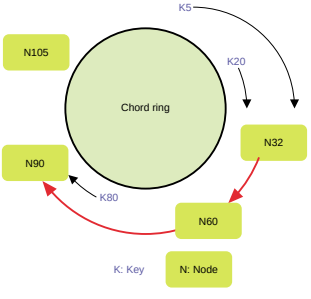

Fig. 2. Example of a Chord ring.

then describes how to accelerate them up with the finger table.

Every Chord node maintains a list of the IDs and IP addresses of its *r* immediate successors on the Chord ring. The fact that every node knows its own successor means that a node can always process a lookup correctly: if the desired key is between the node and its successor, the latter node is the key's successor; otherwise the lookup can be forwarded to the successor, which moves the lookup strictly closer to its destination. A new node *n* learns of its successors when it first joins the Chord ring by asking an existing node to perform a lookup for *n*'s successor; node *n* then asks that successor for its successor list. The *r* entries in the list provide fault tolerance: if a node's immediate successor does not respond, the node can substitute the second entry in its successor list. All *r* successors would have to fail simultaneously to disrupt the Chord ring; an event that can be made very improbable by using modest values of *r*. An implementation should use a fixed *r*, chosen to be $2 \log_2 N$ for the foreseeable maximum number of nodes *N*.

The main complexity involved with successor lists lies in notifying an existing node when a new node should be its successor. The stabilization procedure described in [1] does this in a way that guarantees to preserve the connectivity of the Chord ring's successor pointers.

Lookups performed only with successor lists would require an average of *N*/2 message exchanges, where *N* is the number of nodes. To reduce the number of required messages to *O* (log *N*), each node maintains a *finger table* with *m* entries. Figure 3 shows an example of a finger table. The i^{th} entry in the table at node *n* contains the ID of the successor of $n+2^{i-1}$ on the Chord ring. Thus every node knows the IDs of nodes at power-of-two intervals on the Chord ring from its own position. A new node initializes its finger table by querying an existing node. Existing nodes whose finger table or successor list entries should refer to the new node find out about it by periodic lookups.

Figure 4 shows pseudo-code for looking up the successor of identifier *id*. The main loop is in *find_predecessor*, which sends *preceding_node_list* remote procedure calls (RPCs) to a series of other nodes;

```
// Ask node n to find id 's successor; first finds id 's predecessor,
// then asks that predecessor for its own successor.
n. find_successor (id )
        n' = find_predeccessor (id );
        return n'.successor ();

// Ask node n to find id 's predecessor.
n. find_predecessor (id )
        n'= n;
        while (id ∉ (n', n'. successor()]);
                l = n'. preceding_node_list (id );
                n' = max n" ∈ l s.t. n" is alive
        return n';

// Ask node n for a list of nodes in its finger table or
// successor list that precede id.
n. preceding_node_list (id )
        return {n' ∈ { fingers ∪ successors} s.t. n' ∈ (n, id ]}
```

Fig. 4. Pseudo-code for finding the successor of *id*.

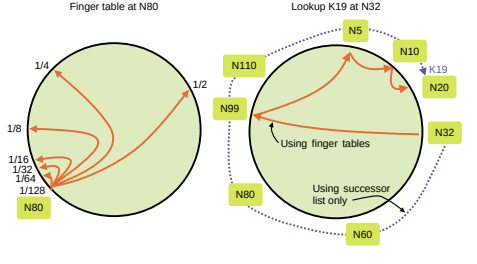

Fig. 3. Example of a finger table and its lookup.

each RPC searches the tables of the other node for nodes that are even closer to *id*. Because finger table entries point to nodes at power-of-two intervals around the Chord ring, each iteration will set *n'* to a node halfway on the Chord ring between the current *n'* and *id*. Since *preceding_node_list* never returns an ID greater than *id*, this process will never overshoot the correct successor. It may undershoot, especially if a new node with an ID just before *id* has recently joined; in that case the check ensures that *find_predecessor* persists until it finds a pair of nodes that straddle *id*.

2.2 DHash

DHash is a distributed hash table that works on top of the Chord lookup service. It maps keys to arbitrary values and stores each key/value pair on a set of nodes determined by hashing the key. This paper refers to a DHash key/value pair as a DHash block. DHash uses Chord to find a node holding a block identified by a given key and replicates blocks to avoid losing them if nodes crash.

DHash ensures the integrity of each block with one of two methods (Fig. 5). A *content-hash block* requires the block's key to be an SHA-1 hash of the block's value (i.e., contents); this allows anyone fetching the block to verify the value by ensuring that its SHA-1 hash matches the key. A *public-key block* requires the block's key to be an SHA-1 hash of a public key, and the value to be the content signed with the corresponding private key. DHash refuses to store a value whose key does not match. Clients check the authenticity of all data they retrieve from DHash. These checks prevent a malicious or buggy DHash node from forging data. Table 1 shows the application programming interface that DHash exposes.

Applications of DHash can use a public-key block as a *root* block of their application data; the root block holds pointers (IDs) to other blocks that store actual content of data or meta data such as filenames. CFS [3] is an example of such an application; it builds a file system that has an i-node metadata structure similar to the UNIX file system.

(1) Block replication

DHash replicates each block on *k* nodes to increase availability, maintains the *k* replicas automatically as nodes come and go, and arranges replicas so that clients can easily find them. It places a block's replicas at the *k* nodes immediately after the block's successor on the Chord ring (Fig. 6). It can easily find the IDs of these nodes from Chord's *r*-entry successor list. DHash must be configured so that $r \geq k$.

This placement of replicas means that, after a block's successor node fails, the block is immediately available at the block's new successor. The DHash software in a block's successor node manages the replication of that block by making sure that all *k* of its successor nodes have a copy of the block at all

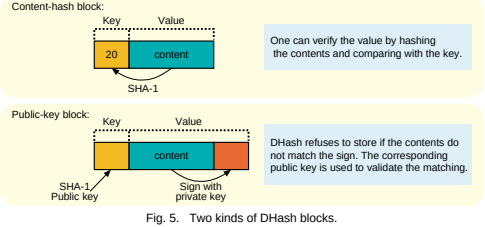

Fig. 5. Two kinds of DHash blocks.

Table 1. DHash client application programming interface.

Function	Description
put_h(*block*)	Computes the *block*'s key by hashing its contents and sends it to the key's successor node for storage.
put_s(*block*, *public_key*)	Stores or updates a public-key block; used for root blocks. The *block* must be signed with the private key corresponding to the *public_key*. The block's key will be the hash of the *public_key*.
get(*key*)	Fetches and returns the block associated with the specified *key*.

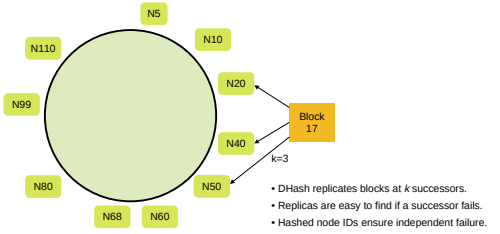

Fig. 6. Block replication of DHash.

times. If the successor node fails, the block's new successor assumes responsibility for the block.

The usefulness of this replication scheme depends partially on the independence of failure and unreachability among a block's *k* replica nodes. Nodes close to each other on the Chord ring are not likely to be physically close to each other, since a node's ID is based on a hash of its IP address. This provides the desired independence of failure.

(2) Block caching

DHash caches blocks to avoid overloading nodes that hold popular data. Each DHash node sets aside a fixed amount of disk storage for its cache. When a DHash client looks up a block key, it performs a Chord lookup, visiting intermediate DHash nodes with IDs successively closer to that of the key's successor. At each step, the client asks the intermediate node whether it has the desired block cached. Eventually the client arrives either at the key's successor or at an intermediate node with a cached copy. The client then sends a copy of the block to some of the nodes it contacted along the lookup path.

Since a Chord lookup takes shorter and shorter hops in the Chord ring as it gets closer to the target, lookups from different clients for the same block will tend to visit the same nodes late in the lookup. As a result, the policy of caching blocks along the lookup path is likely to be effective.

DHash replaces cached blocks in least-recently-used order. Copies of a block at nodes with IDs far from the block's successor are likely to be discarded first, since clients are least likely to stumble upon them. This has the effect of preserving the cached copies close to the successor, and expands and contracts the degree of caching for each block according to its popularity.

DHash avoids most cache consistency problems because content hash blocks are keyed by hashes of their content, so each identified block is immutable. Public-key blocks, however, use public keys as identifiers; a publisher can change a block by inserting new content signed with the corresponding private key. This means that caches of signed public-key blocks may become stale, causing some clients to read old data.

3. CAOSS

NTT is now developing CAOSS (Circumstances Adaptive Online Storage System), an online distributed storage system that uses arbitrary nodes, such as servers and user terminals, on a network for storage. It uses a refined version of the primary copy replication technique [5], which is adjusted to changes in circumstances in ubiquitous networks; i.e., replicas have different lifetimes. Moreover, it provides two kinds of policies for maintaining data consistency.

3.1 Components

CAOSS consists of the following elements (Fig. 7). They exist in arbitrary nodes; i.e., PC, server, PDA, etc.

- Depot: A set of data. It is identified by a depot identifier (DID), which is unique in a network.
- CD (Core Depot): A depot with the authority to perform updating. Only one CD per DID exists in a network.
- RD (Replicated Depot): A depot that can only be read by clients. Zero or more RDs may exist within a network for each DID.
- SRD (Strictly Consistent Replicated Depot): An RD whose data is strictly consistent with the corre-

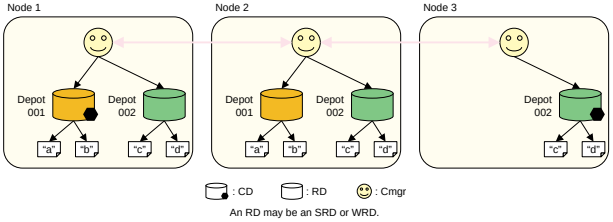

Fig. 7. CAOSS components.

sponding CD.

- WRD (Weakly Consistent Replicated Depot): An RD whose data may differ from that of the corresponding CD because the CD has been changed (i.e., data in WRDs may be old).
- Cmgr (CAOSS manager): An agent that receives client requests and manages distributed replicas of depots (i.e., CDs and RDs).
- Client: An agent that uses CAOSS to access data.

3.2 Management of depots

In CAOSS, the depot generating, updating, and reading processes are performed as follows. A Cmgr generates a CD when a new depot is needed. Clients can save data in the depot after it has been generated (Fig. 8).

When a Cmgr receives a request to read data in a depot, if it does not have the CD or RD of the depot, it reads the data from the Cmgr managing the CD and responds to the request. After that, an RD is created in that node and the RD's lifetime is set (Fig. 8). The RD is either an SRD or WRD depending on the client's read request; if the read request required the newest data, the RD should be SRD. If the request required fast data access, the RD should be WRD.

When a Cmgr receives a request to update data in a depot, if it has the CD, it negotiates with all the Cmgrs managing SRDs. Then, the Cmgr of the CD performs the update, and reflects it in SRDs (Fig. 9). After that, it is possible to read the newest data from SRDs. This process is done according to the primary copy replication technique [5]. If the update succeeds, the Cmgr reports the update to the Cmgrs managing the WRDs, though this information might not

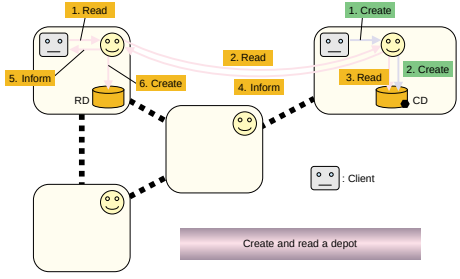

Fig. 8. Creation and reading sequences of CAOSS.

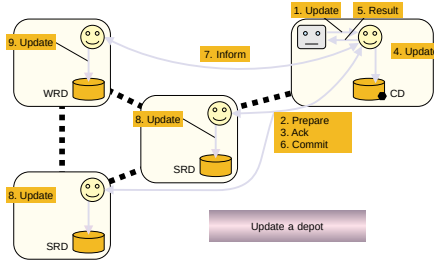

Fig. 9. Update sequences of CAOSS.

be received by the Cmgr because of network failure.

When a Cmgr receives a request to update data in a depot, if it does not have the CD, it sends an updating request to the Cmgr managing the CD, which updates the depot as described above. If the update succeeds, the first Cmgr negotiates with the second Cmgr to move the CD. This movement may not occur in some network situations or for some user access frequencies.

Even if Cmgrs cannot communicate with each other because of a network topology change, clients can still read the newest data from an SRD until its lifetime expires. However, even in a node with a CD, data updating in this depot cannot be performed until the SRD's lifetime has expired. The Cmgr makes an SRD invalid if its lifetime expires before it can communicate with the Cmgr that manages the CD. After that, the Cmgr that manages the CD can update the depot by cooperating with the Cmgrs that manage SRDs.

Thus, we can prevent a fall in availability of updating and reading data with SRDs in ubiquitous network environments by moving a CD to the node that updates the data in the corresponding depot and invalidating inappropriate SRDs. WRDs can improve the availability of reading without affecting the updating availability.

4. Relationship between DHash/Chord and CAOSS

In CAOSS, a client asks a Cmgr to access data in a depot. If the Cmgr does not know the CD's location, it must discover it. However, in a ubiquitous network environment we cannot assume the existence of a centralized server that could tell Cmgr the CD's location. Chord solves this problem: the Cmgr asks Chord using DID as a key and tells Cmgr the CD's location (Fig. 10(a)).

Moreover, a Cmgr of a CD can save its backup in DHash and report its key to a Cmgr that manages the corresponding RD. Thereafter, the latter Cmgr can retrieve the CD in case the former Cmgr becomes unavailable (Fig. 10(b)). This can improve the availability of a depot, but the precise mechanism is for further study.

5. Conclusion

This paper overviewed two ubiquitous storage systems, DHash/Chord and CAOSS. We think that they will make future ubiquitous services practical and easy to develop. Part of DHash/Chord was developed within the framework of MIT-NTT collaboration. NTT is now evaluating the performance of CAOSS by simulation and by testing a prototype system to confirm its feasibility and is considering connecting DHach/Chord and CAOSS.

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," ACM SIGCOMM 2001, San Diego, U.S.A., Aug. 2001.

[2] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, "Ivy: A Read/Write Peer-to-peer File System," Fifth Symposium on Operating Systems Design and Implementation (OSDI), Boston, U.S.A., Dec. 2002.

[3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," ACM SOSP 2001, Banff, Canada, Oct. 2001.

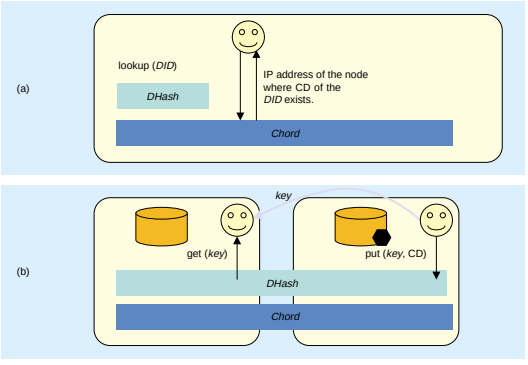

Fig. 10. Collaboration between Chord/DHash and CAOSS.

[4] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," 29th Annual ACM Symposium on Theory of Computing, pp. 654-663, 1997.

[5] P. A. Alsberg and J. D. Day, "A Principle for Resilient Sharing of Distributed Resources," 2nd Int. Conf. on Software Engineering, pp. 562-570, San Francisco, California, Oct. 1976.

Robert Morris
Associate Professor in MIT's EECS department and a member of the Laboratory for Computer Science.
He received a Ph.D. from Harvard University, U.S.A., for work on modeling and controlling data networks with large numbers of competing connections. He co-founded Viaweb, an e-commerce hosting service. His current interests include modular software-based routers, analysis of the aggregation behavior of Internet traffic, routing in multi-hop rooftop 802.11 networks, robust server architectures, distributed hash tables, and peer-to-peer file storage.

M. Frans Kaashoek
A professor of computer science and engineering in MIT's department of electrical engineering and computer science and a member of the MIT Laboratory for Computer Science. Before joining MIT, he was a student at Vrije Universiteit in Amsterdam, the Netherlands.
He received a Ph.D. degree in 1992 from Vrije Universiteit for his thesis on group communication in distributed computer systems, under the guidance of Andy Tanenbaum. His research interest is computer systems: operating systems, networking, programming languages, compilers, and computer architectures for distributed, mobile, and parallel systems. His current research is focused on peer-to-peer systems. In 1998 he cofounded Sightpath Inc, which was acquired by Cisco Systems in 2000. He also serves on the board of Mazu Networks Inc. In 2001 he received the first ACM Mark Weiser Award.

Motonori Nakamura
Senior Research Engineer, Network Intelligence Laboratory, NTT Network Innovation Laboratories.
He received the B.E. and M.E. degrees from Nagoya University, Nagoya in 1990 and 1992, respectively. In 1992, he joined NTT Communication Switching Laboratories, Tokyo, Japan. After that, he researched distributed systems, a name service for large communications systems, distributed resource management, and ad-hoc network protocols for fixed wireless access systems. His current interest is distributed storage systems and seamless network services. He is a member of the Institute of Electronics, Information and Communication Engineers (IEICE) of Japan and the Information Processing Society of Japan (IPSJ).

Tomohiro Inoue
Research Engineer, Network Intelligence Laboratory, NTT Network Innovation Laboratories.
He received the B.E. and M.E. degrees from the University of Tokyo, Tokyo in 1999 and 2001, respectively. In 2001, he joined NTT Network Innovation Laboratories, Tokyo, Japan. His current interest is distributed storage systems in dynamic network environments. He is a member of IPSJ.