Atomic Mutable Data in a Distributed Hash Table

Athicha Muthitacharoen Robert Morris

Abstract

We have developed DHC, a new algorithm that provides atomicity for mutable data in a cooperative DHT. DHC is simple and completely decentralized. We have implemented DHC as an extension of the DHash block storage system. Our preliminary experiments show that DHC is 25% slower than the default DHash implementation. Furthermore, DHC does not incur significant performance degradation even during membership reconfiguration.

1 Introduction and Related Work

Distributed hash tables [1, 9, 7] typically provide good support for read-only data, but have very loosely-defined consistency guarantees for mutable data. DHash [1], for example, supports mutable blocks; these blocks are replicated for availability and fault-tolerance. However, DHash does not guarantee that data is fresh or that replicas are identical. These properties make it difficult to build systems with well-defined semantics on top of DHash. Applications which depend on the freshness of mutable data for correctness include read/write file systems [6] and cooperative decentralized messaging systems [2]).

There are a number of reasons why data consistency is difficult in DHTs. It is not reasonable to assume fail-stop nodes, since nodes may lose network connectivity or suffer network partition. Nodes cannot distinguish a network partition from a failure of the unreachable part of the system; this means that partitioned nodes must retain whatever data they are storing. While partitioned, they may miss writes to that data. For similar reasons, nodes may not be specifically aware that a partition is healing, so nodes that miss writes may unwittingly serve stale data after a partition heals.

A straight-forward solution is to allow only the partition with a majority of the replicas to serve read/write requests. However, unlike traditional replication systems, nodes that fail may never return to life. Thus, peer-to-peer systems must generate new replicas to replace the dead ones. Even a rule that only a majority of replicas can create new a replica is not sufficient. Replicas from the majority may slip into the minority partition one at a time, eventually leaving both partitions with an apparent majority. The key idea to providing consistency is to ensure that a *unique succession* of configurations of replicas exists, and that only the one currently valid configuration serves read and write operations.

Providing atomic consistency for mutable data in a system with dynamic membership is an active field of research. Rodrigues et. al. [8] use a small configuration service to maintain and distribute a list of all live nodes. Since every node is aware of the complete list of nodes, it is easy to transfer and replicate data while ensuring consistency. However, maintaining global knowledge probably limits the approach to small or relatively static systems. Lynch et. al. proposed a solution in [4] and later published RAMBO [5]–a decentralized algorithm that provides atomicity for mutable objects in a dynamic membership setting. RAMBO replicates each data object at a configuration of nodes, and allows multiple configurations per object to exist simultaneously. It allows writes and reads to proceed even during membership change, as long as each operation completes at a quorum in each configuration before returning to the client.

To explore a simpler and more scalable approach, we have designed and implemented DHC, a new atomic update algorithm for DHTs. DHC can cope with fail-stop node failures and network partitions, as long as a majority of the replicas is reachable. DHC ensures that there is only one active configuration of replicas of a mutable data object at any given time, by obtaining consensus among members of the current configuration before moving on to a new configuration. DHC executes each read and write on every node in the configuration and returns successfully if the operation succeeded on at least a majority of the configuration. DHC only needs to know about nodes that are nearby in the ID space of the underlying DHT, and thus is potentially scalable. DHC freezes reads and writes during configuration changes in order to avoid the complexity of concurrent active configurations.

We have implemented DHC as a replacement for the default protocol used to handle mutable blocks in DHash. Our experiments show that DHC is slightly slower than the default DHash implementation. Furthermore, DHC does not incur significant performance degradation even during membership reconfiguration.

2 Background

Our algorithm is an extension to DHash [1], a distributed hash table. DHash uses Chord [10] as a look-up service. We assume the reader is familiar with Chord and DHash. We also assume that nodes in the system are cooperative and behave according to our protocol.

Every Chord node has a unique ID. Chord arranges that each node knows its *successor*, the node with the next highest ID; the ID space wraps around at zero. Thus the nodes form a circular linked list. DHash stores "blocks," or key/value pairs. DHash stores each block on the node that is the block's successor (the node whose ID most immediately follows the block's key).

3 Algorithm

DHC's main jobs are to ensure a single sequence of configurations of replicas, to ensure that a majority of nodes from a configuration gives permission to form the next configuration, and that each read and write operation occurs in a valid configuration.

We extend each DHash mutable block to contain a *version* tag, which is a pair `<sequence number, writer's ID>`. Each mutable block can be updated by multiple writers. DHC provides atomicity for each mutable block. It does not make any guarantees about semantics between different blocks. We refer to a DHash mutable block as simply a block throughout the rest of the paper.

3.1 Inserting a new block

To insert a new block, the writer passes the block's data to the DHash client on the local machine. Once in DHash, DHC marks the block with a *version* tag, which has the form `<version number, writer's ID>`, initially set to `<0, writer's ID>`. Since multiple writers can update a block, the version tag has to include the writer's ID along with the version number. This way, DHC can deterministically order updates to the block when they happen concurrently. DHC replicates the block at k different DHash nodes, where k is the system-wide replication factor. Because the block is new, DHC can directly insert it at the initial replicas. DHC chooses the set of replicas, initially set to be the k immediate successors of the block's ID. We call this replica set a *configuration* of nodes responsible for a block. DHC ensures that there is at most one active configuration at a time, and numbers the successive configurations with increasing sequence numbers. A block's replica knows the number of the configuration in which it is participating. At least a majority of the configuration has to agree on each operation, including forming the next configuration, before it is declared successful. This way, if there is a network partition, there will not be multiple concurrent block versions, since only the partition with the majority of replicas can proceed.

DHC defines the $primary$ in each configuration to be the node with the ID immediately following the block's ID. For each block, a replica keeps a $status$ flag, whose possible values are `idle`, `recon_inprog`, and `write_inprog`; and a $config$ variable, which contains the configuration sequence number and the IDs for the nodes in the block's configuration. Initially, $status$ = `idle` and $config$ = `<0, {initial set of replicas}>`.

3.2 Read operation

To read a block b with key bID, a client sends a `GET` RPC to the node i who is the immediate successor of bID. i checks if it is the current primary of bID and if bID is not going through a reconfiguration or a write operation ($b.status$ = `idle`). If both conditions are true, it sends a `GETBLOCK` RPC to each node in $b.config$. If not, the system may be going through membership reconfiguration, so i returns an error. The client can then retry the operation later when the system stabilizes.

When a node j receives a `GETBLOCK` RPC for bID, it looks in $b.config$ to see if the sender is the current primary of bID and if $b.status$ = `idle`. If both conditions are true, j returns with the block value. If not, it returns with an error.

Node i collects replies from the members of the configuration. When a majority of the replies agree on the *version* tag and value of the block, i returns that value to the client. If there is no such majority after hearing from all the replicas or after a timeout, i returns with an error. The system may be partitioned such that the client and i are in the minority partition.

3.3 Write operation

To write a block b with key bID, a client sends a `PUT` RPC, which contains a new block value, to the node i who is the immediate successor of bID. i consults its local state to see if it believes it is the current primary of bID and if $b.status$ is `idle`. If both conditions are true, i starts the write protocol:

1. i does the following:
   - Sets $b.status$ to `write_inprog`.
   - Increments $b.version$ tag to `<version number+1, writer's ID>`.
   - Updates b's value to the new value.
   - Sends a `PUTBLOCK` RPC to each node in $b.config$.
2. When a node j receives a `PUTBLOCK` RPC for bID, it checks whether the sender is the current primary of bID. If so, j updates the block with the new tag and value, and returns with a positive ack. If not, it returns with an error.
3. When node i receives positive acks from a majority of the replicas, it declares the write successful and returns to the client. If it does not receive a majority of positive acks after hearing back from all the replicas or after a timeout, it returns with an error. In either case, i resets $b.status$ to `idle`.

If i receives a `PUT` RPC while $b.status$ is `write_inprog`, i returns with an error. The client can retry the write later.

3.4 Reconfiguration

Since nodes can leave and join the system, DHC needs to detect and form new configurations of replicas accordingly. Every t seconds, each DHash node i considers all the blocks in its database. For each block b with key bID, if i is the immediate successor of bID or if $i \in b.config$, i checks if $b.config$ matches Chord's idea of the k immediate successors of bID. If the two lists do not match, i takes one of the following actions:

- If i is the immediate successor of bID, it initiates the reconfiguration protocol, described below.
- Otherwise, i sends all the information it has about b ($version$ tag, configuration, etc.) to the immediate successor of bID,

3.4.1 Reconfiguration Protocol

When a node i would like to form a new configuration for a block b with key bID, it needs to make sure that the current configuration will not serve any requests to the block once a new configuration has formed. It also has to make sure that only one new configuration succeeds the current configuration, since there might be more than one node who believe it is the immediate successor of bID (and thus, initiates reconfiguration).

To satisfy the requirements above, i tries to get an agreement on what the next configuration should be by initiating the Paxos [3] distributed consensus protocol, using b's current configuration as consensus participants. Once the current configuration has made a decision, i notifies the new configuration and the nodes in the current configuration becomes the old configuration. We now describe how DHC uses Paxos to decide on the next configuration. All variables mentioned below are local to the block undergoing reconfiguration.

1. i selects a "proposal number" p, which has the form `<n, i's node ID>`, where i increments `n` every time it makes a new proposal. i sends a `PREPARE` RPC with arguments <p , $b.config.seqnum$> to all nodes in $b.config$.

2. When a node j receives a `PREPARE` RPC with p greater than that of any `PREPARE` RPC for $b.config.seqnum$ that it has received, j responds to i with a `PROMISE` RPC, which guarantees that it will not accept any *proposal* with number less than p[1]. The RPC includes the highest-numbered *proposal* that j has accepted, if any, along with b, the current copy of the block. As a result, j stops serving any read or write requests to b.

   > A *proposal* has the form <p, cur_seqnum, $newconfig$>, where p is the number of the proposal, cur_seqnum is the current configuration sequence number, and $newconfig$ is the next proposed configuration. j also sets $b.status$ to `recon_inprog`.

3. If i receives `PROMISE` RPCs from a majority of $b.config$, then it sends a `PROPOSE` RPC to each node in $b.config$. This RPC includes the proposal <p, $b.config.seqnum$, $newconfig$>, where $newconfig$ is the configuration from the highest-numbered proposal among the `PROMISE` RPCs. If none of `PROMISE` RPCs include a proposed configuration, then i sets $newconfig$ to the k immediate successors of bID.

4. When a node j receives a `PROPOSE` RPC for a proposal numbered p, it accepts the proposal by sending an `ACCEPT` RPC to i, *unless* it has already sent a `PROMISE` RPC in response to another `PREPARE` RPC with proposal number greater than p. In the latter case, j returns with an error.

5. If i receives `ACCEPT` RPCs from a majority of $b.config$, then consensus is reached. i sets the $version$ tag and value of b to the most recent value received in the `ACCEPT` RPCs. Then, i sends a `NEWCONFIG` RPC to each node in $newconfig$ to notify that they are the new configuration of the block. A `NEWCONFIG` RPC includes <b, $b.config.seqnum$, $newconfig$>. If i heard from all the nodes and did not get `ACCEPT` from a majority, or if a timeout has passed, i will have to decide whether to initiate another round of the protocol.

6. When a node j receives a `NEWCONFIG` RPC for b, it checks if the block received in the RPC is newer or same version as the one it has in the database (if any) and if the cur_seqnum in the RPC is newer than the one it holds for b (if any). If both conditions are true, j updates b's information its local database, which sets $b.status$ to `idle`. After that, it can immediately start serving read and write requests for b even if not all the nodes in $newconfig$ have been notified. To the client, reads and writes will not complete until at least a majority of the nodes have responded successfully to request.

During reconfiguration, the current replicas will not respond to read or write requests. We assume the case where reconfigurations happen less frequently than reads and writes.

4 Evaluation

We tested DHC on 16 DHash virtual nodes, spread evenly among 3 machines. For fairness, we set the replication factor to 5 for DHC and 3 for DHash, since DHash waits for writes to complete at every replica which DHC only waits for the majority. In the WAN experiments, the machines are located in

[1]We can determine how proposal numbers are ordered separately, as long as every node uses the same ordering.

Experiment	Read (ms)	Write (ms)	Recon (ms)
DHash LAN	4.55	73.2	–
DHC LAN	6.01	76.8	166
DHash WAN	27.5	155	–
DHC WAN	36.2	212	276

Table 1: Real-time in milliseconds for DHC to complete each operation, compared with the default DHash implementation.

New York, Utah, and California; each pair of machines has an RTT of about 55 ms. The latency for each read and write operation is the average of 100 trials. In the reconfiguration experiment, we killed one of the 3 machines and record time from the start till end of the reconfiguration protocol.

Table 1 shows the latency of each operation. DHC has higher read latency than DHash, since DHC needs to hear back from at least a majority of the replicas before returning to the client. For the WAN write experiment, DHC is slower than DHash due to the extra roundtrip it uses for writes. Finally, the reconfiguration latency in the DHC WAN experiment is larger than the write latency by about 1 RTT, which reflects the extra roundtrip in the reconfiguration protocol.

5 Discussion

The reader may notice that designating a node as the primary of the configuration weakens the fault-tolerance of DHC. We argue that the primary keeps the algorithm simple, enables normal write operations to complete in two roundtrips, and only poses some delay in rare cases. For example, if the primary fails after sending `PUTBLOCK` RPCs to the replicas, the write operation will never return to the client. After the timeout t (Section 3.4), the system detects that the primary has failed and reconfigures (which took less than a second in our preliminary WAN experiments). The client can then retry the operation. We can set t to reflect any granularity of freshness that the client desires.

6 Summary

We designed and implemented DHC, a simple and decentralized algorithm that provides atomicity for mutable blocks in DHash. The preliminary experiments show that DHC is 25% slower than the default DHash implementation, and incurs a small performance loss during reconfiguration.

Acknowledgement

We thank Nancy Lynch, Seth Gilbert, and members of the Chord project for invaluable suggestions to improving DHC; David Andersen for access to the RON wide-area testbed.

References

[1] F. Dabek, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[2] Alan Mislove et. al. POST: A secure, resilient, cooperative messaging system. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, May 2003.

[3] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[4] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *Proc. of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[5] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service. In *Proceedings of the 16th International Symposium on DIStributed Computing (DISC '02)*, Toulouse, France, October 2002.

[6] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, pages 161–172, August 2001.

[8] Rodrigo Rodrigues, Barbara Liskov, and Liuba Shrira. The design of a robust peer-to-peer system. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[9] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[10] I. Stoica, R. Morris, David Liben-Nowell, D. Karger, M. Frans Kaashoek, Frank Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Transactions on Networking*.