Donut: A Robust Distributed Hash Table based on Chord

Donut is an implementation of an available, replicated distributed hash table built on top of Chord. The design was built to support storage of common file sizes in a closed cluster of systems. This paper discusses the two layers of the implementation and a discussion of future development. First, we discuss the basics of Chord as an "overlay network" and our implementation details, second, Donut as a hash table providing availability and replication and thirdly how further development towards a distributed file system may proceed.

Chord

Introduction

Chord is an overlay network that maps logical addresses to physical nodes in a Peer-to-Peer system. Chord associates a ranges of keys with a particular node using a variant of consistent hashing. While traditional consistent hashing schemes require nodes to maintain state of the the system as a whole, Chord only requires that nodes know of a fixed number of "finger" nodes, allowing both massive scalability while maintaining efficient lookup time (O(log n)).

Terms

Chord

Chord works by arranging keys in a ring, such that when we reach the highest value in the key-space, the following key is zero. Nodes take on a particular key in this ring.

Successor

Node *n* is called the successor of a key *k* if and only if *n* is the closest node after or equal to *k* on the ring.

Predecessor

Node *n* is called the predecessor of a key *k* if and only *n* is the closest node before *k* in the ring.

Finger

Nodes maintain a set of other nodes, called *fingers*, which refer to other points on the ring. The first *finger* of node *n* is always the successor of *n*.

Variables

Variable	Definition
K	the total number of possible keys
k	a particular key in the key-space
m	log(K)
N	total number of nodes currently in the ring
n	a particular node
r	number of successors to keep in the successor list

Basic Design

A node n with key k stores a finger table containing the successors of the keys $k + 2^0$, $k + 2^1$, $k + 2^2$, ..., $k + 2^m$. (Note: it may be the case that one node in two different finger entries). Given this distribution of fingers, the number of lookup of any key's successor, as well as the number of messages to achieve the same, is O(log(N)).

Chord's lookup protocol is defined by the *findSuccessor* procedure. If node n realizes it is the predecessor of requested key k, *n.successor* is returned. Otherwise, the node finds the closest predecessor of k by searching *fingers* and invoking *findSuccessor* on the closest preceding node.

Donut's *findSuccessor* is recursive, meaning each node in the call tree blocks until a result is found. It is also possible to do this asynchronously, where the predeccessor of the key returns its successor directly to the client. Donut is particularly attune to this optimization since the clients are request servers still internal to the system. However, network hops are not a primary performance bottleneck, so this optimization has not been implemented.

Joins

Nodes joining a chord ring must bootstrap using an existing node in the circle. To join existing known node n, joining node n' invokes *n.findSuccessor(n')*. Once n' finds its successor, n'', n' notifies n'' which instructs n'' to set its predecessor to n'. This is done in the *stabilize* routine. The joined is complete once the node immediately preceding n' recognizes that it is the predecessor of n'.

Stabilize and Notify, Fix Fingers and Check Predecessor

With joining and leaving nodes, finger table entries, successors and predecessors must be periodically updated for all nodes. These operations are done at scheduled intervals spanning three procedures: *fixFingers*, *stabilize*, and *checkPredecessor*.

stabilize verifies a node's immediate successor and then notifies that successor to set its predecessor correctly. *stabilize* first queries its successor for its predecessor. If the successor has a predecessor between the two nodes, the node corrects who its successor is. This can occur when a new node joins the ring and its key lies between the stabilizing node and it's immediate successor.

fixFingers sequentially updates the finger table by invoking *findSuccessor*. Specifically, to update the i^{th} entry in the finger table, *findSuccessor* is called on the key $k + 2^i$, where k is the current node's key.

checkPredecessor ensures that a node's predecessor is alive and well. A node periodically pings its predecessor. If the node does not respond, it is assumed dead and the node sets itself as the predecessor, awaiting notification from its new predecessor.

Leaves and Successor Lists

As nodes leave the ring, usually due to system crashes or network errors, the ring stabilizes itself through *fixFingers*, *stabilize* and *checkPredecessor*. Between these three functions the predecessor gets set by *notify*, invalidated by *checkPredecessor*, fingers get updated by *fixFingers*, and the successor gets updated by *stabilize*. However, when a node's successor fails it must be invalidated and set to a new successor in order for the ring to be correct. Unlike a node's predecessor, a node will not be notified of a new, valid successor.

For added robustness, a node keeps a list of r successors from which to remove the failed, old, successor and immediately update to the next successor. More formally, a successor list is the set of r successors to a node. The successor list is populated by the *notify* response from a node to its predecessor.
By using a successor list to handle node leaves, Donut can ensure the overlay network is resilient from up to $r - 1$ concurrent node failures within a defined interval.

Hash table

Interface

Donut implements two levels of hash table interfaces: one between the end-user and the system, and another between the request servers and donut nodes. This separates the implementation details to be abstracted from the user while maintaining flexibility to optimize the interface for system-internal operations.

End-user interface

The external interface, exposed to the end-user is a traditional get/put/remove hash interface. *get* takes a string (the key), and returns a stream of binary data (the value for that key). *put* takes a string (the key) and a stream of binary data (the value to be inserted for that key). *remove* takes a string (the key). The end-user is not exposed to details such as how the key is hashed, replication, or the separation between finding the responsible node:DonutNode and data transfer.

Internal interface

Request servers have a more fine grained interface with the donut nodes than with the end-user clients.

- A call *findSuccessor* is exposed to the request servers.
- The key used in the *get*, *put*, and *remove* routines have two fields: a string and a 64-bit id (in practice the id is encapsulated in such that the key-space could be arbitrary extended). Keeping a copy of the original key (the string) allows the system to deal with collisions.

Internally, finding the node responsible for a key and propagating the data are separate tasks. On receiving a request from the end user, a request server hashes the key into a 64-bit number. It then calls *findSuccessor* to find the node responsible for the key and invokes the applicable hash table procedure on that node.

Replication

Replication is done on two levels. Internally, nodes in the ring replicate data to their successor to guarantee availability of data when nodes leave the ring. A second level or replication is implemented by the client. On this level data is replicated to different keys across the ring. This level of replication can guarantee that stale reads can be detected.

Chord level

A donut node replicates every write request (both *put*s and *remove*s) to the next *R* successor nodes. This guarantees that when a node leaves the ring, it's successor – which becomes responsible for it's data – maintains a consistent view of that section of the data.

Replication is implemented in a way that guarantees that all successors have the data change before a write is considered finished. Specifically, a *replicatePut* or *replicateRemove* is sent recursively to the *R* successor nodes. The actual commit (update or removal of data) happens on the way back up the call stack. The originating node will not receive a response unless all nodes replicated successfully. The goal is not to guarantee that stale reads will not occur, or that the client can in all cases ascertain whether the write was successful. Rather, the goal is to guarantee that if a particular write was not successful the client will know. This means there are scenarios in which all nodes replicating a key have the updated data, but the client would be notified that the write was not successful.

Client level

There is another level of replication that is done on a higher level, by the request servers. The method used is based on a design in Dearle, Kirby, Norcross[1]. There is a global constant *R* which specifies the total number of replicas (including the master).

For a given key *k*, $R - 1$ additional keys are computed to replicate the data on such that $k(i) = k_0 + i * K / R$, where *i* is the i^{th} replica. Given a reasonably distributed set of nodes, the successors for these keys will be a set of *R* unique nodes. If the chosen *R* is such that the total number of replicas is odd, this method can be extended to ensure that stale reads are identified.

Each write is done to all *R* keys. All subsequent reads of that key are also done on all *R* keys. If there is a consensus (a majority of the replicas hold the same value) than the read is correct and up to date. Otherwise the read is stale. Several scenarios can produce the stale read state. For a discussion of some of these scenarios, as well as some suggestions for possible ways to recover from stale reads.

Future Directions

Handling Stale Reads

Currently, Donut can detect stale reads but has no mechanism for alleviating stale reads. Several concurrent writes to the same key can leave that key in an inconsistent state between replicas.

For example, suppose that $R = 3$ (keeping three replicas at the service level). Two or more concurrent writes may begin but can fail before writing to all *R* replicas or the writes can interleave. Both cases potentially could leave the *R* replicas in three different states, *v* (data before the writes), v' (data from one writer) and v'' (data from another writer). In this case, a consensus about the current value does not exist, preventing Donut from providing future successful reads of the value.

One solution to interleaving writes is utilizing a deterministic ordering of writes to the replicas. For example, write to the hashed key *k*, then $k + i * K / R$ ... $k + (R - 1) * K / R$, where *i* is the i^{th} replica. Ordering writes allows Donut to guess at the most recently agreed upon consensus. The middle key in the order of writes will be the last key to have successfully written to over half of the *R* nodes. In a stale state when $R = 3$, the first node is the most recently written, the second is the most recently written with a consensus, and the third is the only write that completed writing to all nodes.

Ordering client level replication requires the *put*s to be completed serially, effectively tripling the time to write.

Once a stale read is detected, it can be correct by the request handler retrying the write request. Even with the proposed write order above, the procedure to *get* for a conflicted key should return a notification identifying that corrective action must be taken to bring the key into a stable, non-stale state.

Byzantine Fault Tolerance

In researching the direction the Donut, we began to address the problem of byzantine fault tolerance and transient inconsistencies of finger tables (or dynamism). Chord inherently relies on the predecessor of a key to correctly return the final successor. In several of the papers we perused, the common fix is to implement a routing mechanism that ensures multiple paths to reach each node. Therefore, to implement byzantine fault tolerance Donut would most likely need to switch its overlay network from Chord to a more robust solution such as Kademlia or Inverse de Bruijn overlay network[8].

References

1. Alan Dearle, Graham NC Kirby, and Stuart J Norcross, Hosting Byzantine Fault Tolerant Services on a Chord Ring, University of St Andrews 2007, Abstract, PDF.
2. Amos Fiat, Jared Saia, and Maxwell Young, Making Chord Robust to Byzantine Attacks, In European Symposium on Algorithms (ESA), 2005. PDF.
3. Antony Chazapis and Nectarios Koziris, Storing and locating mutable data in structured peer-to-peer overlay networks, Proceedings of the Panhellenic Conference on Informatics 2005. PDF.
4. Apu Kapadia and Nikos Triandopoulos, Halo: High-Assurance Locate for Distributed Hash Tables, Proceedings of the 15th Annual Network & Distributed System Security Symposium (NDSS '08), February 2008. PDF.
5. Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan, Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service, Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 2001. Abstract, PDF.
6. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, ACM SIGCOMM 2001, San Deigo, CA, August 2001, pp. 149-160. Abstract, PDF.
7. Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications, IEEE Transactions on Networking, February 2003. Abstract, PDF.
8. Ying Chen and Kai Hwang, Byzantine Fault Tolerance of Inverse de Bruijn Overlay Networks for Secure P2P Routing, Technical Report, USC Internet and Grid Computing Lab (TR-2006-4), October, 2006. PDF.