The PRO key-value store

CS 739, project 1

Markus Peloquin, Lena Olson, Jeffery Roller
{markus,lena,roller-j}@cs.wisc.edu

March 25, 2010

1 Introduction

The goal of the project was to design and implement a consistent and fault-tolerant distributed key-value store. We could decide what trade-offs we thought were worthwhile. We chose to incorporate many of the techniques we read about in class into our implementation, leading to a design similar in many ways to Amazon's Dynamo [1]. These features include consistent hashing, heartbeats, and anti-entropy. Our PRO system (Peloquin, Roller, Olson) is designed to be eventually consistent, available except in the worst cases, and able to survive most partitions. We also chose to implement a stateful client to reduce the work required per transaction.

Our implementation is described in Section 2. We evaluate our implementation in Section 3, give future work in Section 4, and conclude in Section 5.

2 Features

2.1 Consistent Hashing

We used consistent hashing to share the load evenly between a number of servers. As pointed out by the Chord developers and others, consistent hashing is not enough, so each server gets a number of *segments* arranged around the hash circle [2, 4]. To determine these locations, we hash the server address, port, and a segment index number, allowing each server to independently and without communication locate the segments of another server given only its IP address and port. When a key-value pair is stored, the next server clockwise from (greater or equal to) the hash value is the *primary* node, while the next distinct node (*successor*) is the *secondary*. A strong digest algorithm (MD5) was used both to hash server addresses and keys to the hash circle, achieving highly random locations.

We have a bias towards providing fast reads. On a read, either the primary or secondary node returns the value it has stored. All writes require that both the primary and secondary store the result, if both are running; else the write will be performed on just the remaining. Although the servers are most efficient when the client sends requests to the proper node, requests from clients are still forwarded so less intelligent clients can still use the service. This can happen with our own client, which may contact other servers if it believes the primary or secondary nodes are down This is explained more in Section 2.2.

If one of the primary or secondary goes down, the data is no longer being replicated but is still accessible. If both are not responding, the client receives a message about the service being unavailable for any request.

2.2 Persistent client

One of the advantages of allowing the client to maintain state is that then it can keep track of which nodes are up or down at any given time. This allows the client to choose the correct node to send a request to, avoiding nodes that were recently known to be down. Because the client as specified in the project description was stateless, we chose to implement a client daemon that runs on the same machine as the client. When the client wants to store or get a value, it passes the request on to the client daemon, which uses what it knows about the state of the servers to send the requests more efficiently. To assist the client daemon in determining the status of the servers, we use the HTTP *Pragma* field to attach the server's view to responses to requests from the PRO client. It is more efficient to get bad news from servers that are up than from servers that are down.

2.3 Heartbeats

In order for the servers to determine who is responsible for any given key, they need to be aware of which of the other servers are up. Each server will keep timestamps of the latest communication with each

other. If it has been longer than a specific length of time, the server is marked as down. This works well when there is constant communication between the nodes, but often there may be no requests requiring communication, and so a lack of messages from a node does not necessarily indicate that it is faulty.

To prevent such cases, we implemented a heartbeat mechanism. Since it is not critical that every heartbeat is safely received, we chose to implement it using UDP rather than TCP. The servers regularly send heartbeats to all other servers (every 0.5 s), whether or not they are thought to be down. If no messages of any sort are received from another server in a time period which is several times the length of the heartbeat interval (1.25 s), then the server is marked down and it updates its segment table and view.

The rationale for the timeout was to keep it low so as to stay responsive, and the heartbeat interval was chosen so as to not be excessive. Ideally, servers take a long time to be marked as down but a very short time to be marked as alive. This is because unless the server closes a connection when it fails, it is necessary to wait for some interval to determine whether the server is down, or is simply responding slowly, or if some heartbeat messages are being lost and there are no client requests. When marking the server as alive, it is sufficient for it to have sent one message.

2.4 Anti-entropy

Our anti-entropy system was used to correct inconsistencies that arise occasionally, especially in two common scenarios. When a server comes online, it is important for it to receive the segments for which it is the primary node from the secondary servers. For this critical case, anti-entropy was designed to run immediately when a server starts. It is not in this case *eventual* consistency.

The other common case is to get data from the primary nodes to the secondary nodes, and this is accomplished by the periodic anti-entropy messages. When the anti-entropy process begins for the other existing nodes (every 10 s), the secondary node is able to receive the updates. This does provide a healthy window in which requests may be sent to the secondary, though it would be a simple matter for it to reroute the request if it has not yet received the segment from the primary.

In terms of implementation, anti-entropy is always initiated by primary nodes to secondary nodes, and once for each segment. The primary to secondary connection was intended to reduce complexity and allow primary nodes to receive segments from the secondary replicas as soon as possible. A possible optimization might be for anti-entropy to handle multiple segments at once. There might be several contiguous segments owned by a single host, and these could just as well be contained in the same segment. This assumes that our service will never allow a server to dynamically join our system, and we would rather cut ourselves off from that option for the future.

Version vectors were used to decide which version is newer. Where the version numbers conflict, a timestamp is used as the deciding factor. The vector and timestamp is also returned to the client in an HTTP *Pragma* field.

3 Results

Our results are divided into subsections according to the CAP theorem. We also consider how well our system can scale in Section 3.4.

3.1 Availability

Our service usually achieves good availability. Provided that either the primary or secondary node are available, the reads and writes will be performed. Clients will be unable to perform any action on segments owned jointly by a pair of unresponsive nodes.

We considered allowing tertiary nodes to exist, which would take over in the event the primary and secondary were inaccessible. Until the primary and secondary returned, the tertiary node would be the sole owner of these segments (no replication). Clients would still be able to write and read their own writes, but when the primary/secondary returned, any writes would disappear. A better approach is given in Section 4.

3.2 Consistency

In most cases, consistency is provided. There is no commit model for writes like in Paxos [3]. Instead, we use a model similar to Dynamo: we attempt to send the write to all responsible nodes, and return success to the client if any succeed [1].

When a node comes online, the segments for which it is the primary are updated from the secondary as soon as possible. However, the segments it is secondary for could take as much as 10 s to update as it passively waits for anti-entropy to send it the segment's data. Our service could be modified so

that servers request updates from both sides immediately, but it would require switching to more advanced queuing methods.

3.3 Partitioning

There are some well-defined places where partitioning can cause problems in our system. If a request is sent to a partition not containing a primary or secondary, the service will be unavailable to the client. We have found that once a system becomes unresponsive, requests are delayed for around a second. Requests that arrive after heartbeats discover the partition is down return almost immediately. If values are written to both sides of a partition, the system will return whatever side each partition sees as long as the partition exists. As soon as the partition disappears, anti-entropy will choose whichever version had the latest timestamp.

3.4 Scaling

Our design is definitely not scalable, as it only needs to support four servers. It is only a question of how much needs to be done to make it scalable. Like Amazon Dynamo, in our consistent hashing scheme, every node knows about every other node. It would be algorithmically simple to announce a joining node and to offload partial segments to the new node, provided there are not too many nodes. To scale further, Chord's approach with 'fingers' could be useful [4].

Heartbeats would scale in the same way as consistent hashing. Because heartbeats are used only to determine which nodes are up and simplify determining which nodes are responsible for a given key, they are only necessary between neighboring nodes, as the client will usually send requests to the correct node.

Anti-entropy is not a concern, as it only will exist among k nodes for each segment.

4 Future work

Portions of our system can become unavailable when multiple nodes are down or the system is partitioned. If two nodes are inaccessible, the intersection of all segments they are responsible for cannot be accessed. A solution we liked but were unable to implement due to time constraints was to use segment migration. If either the primary or secondary goes down, the acting primary node would send the segment's data to the acting secondary node. Then, when a node more responsible than this 'acting secondary' appears, anti-entropy would move the data to the newly-running server. After some period of time has passed, the backup server would discard its copy of the segment. This would allow data to remain accessible to clients, and consistency would be maintained when the primary servers rejoin the system.

A simple change would be to have data committed to disk. This would keep data in the system even if the primary nodes go down at the same time. Doing this *correctly* would require asynchronous writes to not hinder anti-entropy throughput. Most operating systems should end up performing asynchronous writes anyway, so little effort on our part would be required.

5 Conclusion

We built an eventually consistent and well-performing system that implements some of the more popular ideas we have come across in distributed systems. Consistent hashing is used to distribute data and do load sharing when systems go down. Data is replicated by the successors along the hash wheel. Anti-entropy fills in the gaps in the commit model to make our system eventually consistent. Heartbeats keep our system performing well by eliminating unnecessary communication and notifying nodes of state changes. We believe the only major task to making our model much more useful and available is segment migration. To make it truly scalable, something like the Chord approach to joining and exiting nodes would be needed.

References

[1] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazons highly available key-value store. In *SOSP'07* (October 2007).

[2] Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., and Yerushalmi, Y. Web caching with consistent hashing. In *WWW '99: Proceedings of the eighth international conference on World Wide Web* (New York, NY, USA, 1999), Elsevier North-Holland, Inc., pp. 1203–1213.

[3] Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.

[4] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev. 31*, 4 (2001), 149–160.