CS739 Spring 2011 Project 1

-------------- Draft ---------------

Toposphere: A Topology Aware Distributed Key-Value Store

Kathik Narayan, Rohit Koul, Sandeep Dhoot, Vinod Ramachandran

{knarayan, rkoul, sandeep, vinorama}@cs.wisc.edu

Main Objectives:

Our system aims at achieving the following:

- Eventual Consistency
- Availability
- Partition Tolerance
- Scalability

Design:

At a high-level, our distributed Key-Value store is eventually consistent - more specifically we guarantee read-your-writes consistency combined with monotonic read consistency. In our system, all server nodes are homogeneous. We follow a P2P model rather than a master-slave configuration. Our focus is on a system that is highly available and tolerant to network partitioning. Our system is always write-able - a client can successfully write even when only a single node is live. We assume a system with synchronized clocks and hence conflicts are resolved using timestamps with a simple "last writer wins" mechanism wherein the last-write is the one with the highest timestamp.

Our system guarantees that if no new updates are made to the object, eventually (after the inconsistency window closes), all accesses will return the last updated value. Our system can also achieve read-your-writes if the clients use 'sticky' sessions. We use Anti-entropy to pull only unnoticed updates from other servers. We also use Checkpointing to induce fault tolerance and limit memory footprint of the servers (using a two-phase commit protocol). Our Heartbeat mechanism ensures that servers contact only the nodes that are deemed alive and IP/Multicast and Multicast groups help disseminate packets efficiently and scalably.

We describe the trade-offs of these techniques in somewhat more detail in later sections.

Implementation:

We used Java 1.6 to implement our system for its efficient memory model and multi-threading capabilities. We, however, modified the same client that was provided to us. All our test cases were written in Python.

The servers are multi-threaded. All the requests from the clients are handled by worker threads from the pool of available threads. Background activities like reconciliation and server synchronization, heartbeat message communication and checkpointing are handled by background threads. As already stated we use IP/Multicast to send updates on writes. This ensures that the sender node only sends an update packet only once, even if the message were supposed to be delivered to a large number of servers. This allows our system to add or drop nodes at will and makes it scalable without any significant messaging overhead. Dropped packets (if any) are reconciled via periodic heartbeat message mechanism.

Primary Data structures used at every server:

1. Key-Value Hashmap: < key : value, timestamp>
2. Uptate Table Hashmap (one per server in the system): < update# : key >
3. Vector Clock : list of <update counter> (# of entries = # of servers)

As the servers start, they join a multicast group and read the configuration file “*servers.txt*” to find out which all servers are expected to participate in the system. This information is then sorted to index every server in the range [1, Num_Of_Servers]. Thus, every server has a universal ordering of all the other servers. This ordering is utilized to minimize the number of message passing needed in the system.

It is worth mentioning that by virtue of our design choice of using IP/Multicast, it is perfectly possible for the servers to determine this universal ordering otherwise also, even without having to read the configuration file (e.g each server could send a HELLO packet to the multicast group on startup). We, however, chose to use *servers.txt* as per the given specifications. At startup, the server also tries to load its previously saved persistent state (checkpoint - which is created either in response to a crash or local garbage collection), if available.

The key-value hashmap stores timestamp corresponding to latest update on a key. Every server maintains update tables (corresponding to every server in the system) which are used to track updates that were carried out on a server on client's request. For 4 servers in the system, every server maintains 4 update tables to totally order updates that were carried out on all the servers. Thus, if a server#2 receives an update request from client, server#2 performs the update and puts the key corresponding to that update in update table#2 in the next consecutive slot. Index for update table entries start from 0 and increment on every locally performed update. Whenever any other server knows that server#2 performed an update at N'th index, that server will perform the same update and put the key for that update in update table#2 (that corresponds to server#2). Note that, the ordering of servers, that we generated initially, allows us to use update tables as outlined here. Thus, when everything is running perfectly fine in the system, all the servers will have consistent key-value store and update tables. The update tables also allow to share ONLY those updates with other servers that they have not already seen, instead of sharing entire key-value store or those updates which are seemingly hot in the system. Every server also maintains a vector clock that lists the indices of last consecutive update in update tables. Thus, in the 4-server system, vector clock will hold 4 entries, each of which tell us the index of last consecutive update received from a server. This vector clock is communicated to other servers, as described later, to pull any new/missed updates in the system.

Whenever an update request is received by a server, that request is locally performed and the timestamp of the request is updated in the key-value hashmap. The update is then registered in the appropriate update table, server's vector clock is incremented to reflect the new update in the update table. The update <key, value, timestamp, server#, update index> is then multicast(UDP) to multicast group which is joined by all the servers in the system. On receiving such an update message from other server, recipient server applies that update locally if the timestamp of the update is later than timestamp of last seen update on the key. Recipient server then adds the key of the update in the update table, corresponding to the server who sent the message, at the index mentioned with the update. The vector clock is then incremented to reflect the change in udapte table iff the received update was a consecutive update received from sender server. This takes care of lost updates due to packet drops. The background server synchronization thread pulls all the lost updates from another server to synchronize vector clocks. Since timestamp is the only criteria to decide on applying the update locally, we do not discard new updates even if we have not received any earlier updates that occurred on other servers.

The server ordering that we mentioned earlier, allows us to significantly reduce the background communication among the servers. We conceive the servers as being present on a ring in the order known to every server. Every server periodically sends a HEARTBEAT message to its successor on the ring. The vector clock of sender server is sent along with the heartbeat message. From sender's vector clock, the recipient server determines if he or sender himself are lagging behind in terms of updates known to them. If sender is lagging, recipient sends him a SYNC message with all the extra updates that he knows but sender does not. If recipient also finds out that sender know more updates than him, then a HYBRID message is sent which holds all the updates from recipient and also recipient's vector clock. Response to HYBRID message is a SYNC message with all the extra updates that a sender server knows. The SYNC/HYBRID message is divided into multiple UDP packets when the large number of updates being transferred overshoots the maximum size of UDP packet (64KB). Loss of any one of these packets does not lead to inconsistency in the system. Thus, the two consecutive nodes on a ring participate in a pull based mutual synchronization process. The successor node communicated on the ring is the next live node in the ordering. To determine which next node is live, we could have used timeout for a response from the next server. But, instead periodically we multicast KEEPALIVE message to all the servers. Thus, whenever a server has not received any (UPDATE / SYNC / HYBRID / HEARTBEAT / KEEPALIVE) message from any other server in the system within KEEPALIVE delay, that server is presumed DEAD and hence not contacted during HEARTBEAT based synchronization process. If any message is received from a presumably dead server, that server is deemed alive and can be contacted by its alive predecessor. This setup allows us to restrict the synchronization communication with only one server in presence of no failures, and also to pick appropriate server who can respond to synchronization request in case of failures. The mutual synchronization, described here, does not require strong connectivity among the servers. Even in case where we have single point-to-point link from a server to only one other server, it allows us to propagate all the lost updates to all the servers. This setup also allows to tolerate partitions in the system. Servers in multiple partitions in the system contact each other in multiple rings as long as partition is present. When the partition is removed, the periodic KEEPALIVE multicast message allows us to join the rings for HEARTBEAT communication.

We also use persistent checkpointing mechanism to limit the size of our update tables and network communication needed when a crashed node comes back up. Our two-phase-commit Garbage Collection (GC) mechanism, periodically invoked by servers, allows us to clear up update tracking data from the update tables that are known to all the servers in the sysetm. The GC thread on a server checks if the number of locally received updates requests (after last checkpoint) on a server has reached GC_UPDATE_COUNT value. If yes, garbage collection mechanism clears GC_UPDATE_COUNT number of entries from update thable of the server which initiated GC. At the end of GC, every server chekpoints its entire Key-Value hashmap and Checkpoint Vector Clock to a file. The checkpoint vector clock is the vector clock such that every entry in the vector clock tells the number of updates that were received by all the servers in the system and thus there is no need to track updates older than that.

Checkpointing is also performed on TERM and INT signals to the server. A server reads the checkpoint file, if present, to bootstrap itself. For any reason, if a new server reports its vector clock in HEARTBEAT message which is older than Checkpoint Vector Clock, then the HEARTBEAT recipient reads its checkpoint file and sends the checkpoint to that server over a TCP connection so that the new server can initialize its key-value hashmap and vector clock to bare minimum checkpoint stage. All the updates in the system that happened after the checkpoint are received in the manner described above. Thus, this combination of Garbage Collection and Checkpointing allows us to keep our update tables as small as desired and to reduce the network communication when a server comes alive after a crash.

Testing / Evaluation:

Below we describe some of the test cases used in evaluating our key value store.

1. Throughput

In this test case we perform 100 read/writes for a certain number of iterations from 10 different processes. We compute the read-only throughput and the write-only throughput. In the case of reads, we have seen throughput in the range of **600 to 1000 reads/second**. In case of writes we observed throughput of at least **530 writes/second**.

2. Base Consistency

In this case we write 40 unique keys for 10 iterations to different servers and the keep track of the (old value,new value) chain of updates for each key. Our system returns consistent results as the (old value,new value) chain of updates is preserved.

3. Break Consistency Test Case

In this case we write 10 queries of the same key with different values to different servers. We do this for 40 keys for 10 iterations. In other words for each iteration we perform 400 writes and every 10 consecutive writes are for the same key across different servers. We then check for the preservation of (old value,new value) chain similar to the base consistency test. In this test we observe that our system does not always preserve the (old value,new value) chain of updates as the network latency for message propagation across servers seems to be higher.

4. Manual Partition

In this case we partition the servers into two partitions. We write 100 keys to partition 1 and 100 keys to partition 2. We then unpartition and then check if the keys get synchronized. Since our keep alive message frequency is 4 seconds we sleep our test script for 3-4 seconds after unpartition before checking for consistency among partitions.

5. Stress Partition

In this case we let the client send write requests to different servers, in between we partition then after sometime we unpartition and then perform reads for the keys that are written and check if the values are consistent across all four servers. Since our keep alive message frequency is 4 seconds it takes a couple read iterations for us to get consistent results from all four servers.

6. Failure

In this case we check if the service is tolerant to failures. We write some data to server that we plan to kill. We then kill this server and then write data to other working servers. We then read the data of the failed server from other servers and then check if we can get consistent results. We then wake up the failed server and then read data written to other servers during its failed period. Since our heartbeat message frequency is 2 seconds we sleep for about 3 seconds and check if these reads are consistent. We can extend this test for 2 failures or 3 failures.

Conclusion:

Thanks to CAP theorem, It is quite impossible to build an ideal distributed system to solve every problem. Often, trade-offs have to be made to build systems that are particularly good at doing certain things and perform reasonable well in other areas. In our Key value store, we sacrifice strict consistency to achieve high availability and partition tolerance.

There are of course, a number of areas of future work for our system. e.g. although our system has a reasonable throughput, 'smart' clients that do consistent hashing and have some sort of sticky sessions would perform better and also provide stronger consistency guarantees. We could also improve the performance of the system further if we could reduce the number of messages exchanged during heartbeats and liveness/beacon signals and/or by persisting the update tables and vector clocks as well . Also, better ways to store the update map values, and data compression could further improve the performance.