CONSISTENT, AVAILABLE KEY VALUE DATA STORE

ANDREW MULLER AND KIRA GHANDHI
ADVANCED DISTRIBUTED SYSTEMS

Contents

1. Introduction

This project is an implementation of a distributed key value data store built on a simple distributed hash table (DHT) running with Paxos [1] for consistency. The implementation is based on the highly distributed peer-to-peer system, Scatter [2]. Unlike many other DHTs such as Chord [3] and Pastry [4] that aim for high availability, Scatter seeks the best of both availability and consistency in addition to being highly scalable and dealing with high *churn* (frequent adds and drops of nodes). As originally proven in the paper [2], Scatter performs better in terms of availability, performance, and consistency than a plain DHT, like OpenDHT. This project deals mainly with optimizing on key range consistency and routing consistency with an interpretation of Scatter.

2. Scatter Overview

Scatter is a fully decentralized peer-to-peer system that aims for consistency and scalability while maintaining the high availability attributed to DHTs. It provides"linearlizable consistency semantics for operations on a single key/value pair despite (1) lossy and variable-latency network connections, (2) dynamic system membership including uncontrolled departures, and (3) transient, asymmetric communication faults" [2]. In addition it can support millions of nodes and is highly adaptable, a feature this project employs.

Date: June 11, 2014.

2.1. **Architecture.** To achieve its goal, Scatter partitions the DHT into *groups* of nodes who are responsible for a specific key range, $[a, b)$. Within each group the nodes are responsible for a smaller subsection of the group's larger key range. The node is called the *primary* for that key range. While each group elects a leader to perform operations between groups, reads from specific keys in the groups are delegated to the primary for that key.

Each node has knowledge of the members of its group as well as the keys and values for all the keys in its group's key range, this provides replication. Each key and value in a group's key range is replicated among all the members of the group to which the key falls. While only the primary is responsible for carrying out `get` and `set` operations on a specific key, it transfers that power over according to failures, group changes etc. Additionally each node knows its group's key range, its group's left neighbor, its group's right neighbor and its own leader. This way each group cannot communicate with the rest of the network, only its neighbors.

2.2. **Nested Consensus.** Scatter supports opperations between groups, *multi-group operations*:

- *Split:* partitioning the coordinating group into two groups
- *Merge:* creating a new group from the union of two groups
- *Migrate:* moving members of one group to a different group
- *Repartition:* changing the key-space partitioning between two adjacent groups

To accomplish these operations as well as `get` and `set` operations Scatter uses what it calls *nested consensus*. Nested consensus is consensus between the groups as well as within the groups. For the outer consensus, consensus between groups, Scatter implements a two phase commit protocol (2PC). In its adaptation of 2PC, the group who wants to perform the multi-group operation, called the *coordinator* must inform both its neighbors of the change in order for the group and its neighbors to remain consistent. At each step of 2PC the group must reach internal consensus using Paxos. Before sending the next 2PC message, or the initial `READY` message, the group must reach internal consensus about the operation. By the end of 2PC the coordinator changes its internal state and sends a `COMMIT` message to its neighbors, and any other participants who will need to update internally, who update their own internal states.

For the actual data storage data is replicated within a group using Paxos. Each group essentially has a routing table of the nodes in its group and where the keys are stored. For a read or write, the message is forwarded to the correct group according to neighbor knowledge and within the group the message is forwarded to the appropriate primary. The primary is responsible for replicating its key value storage among its group members, it is the Proposer for its keys and values. Additionally Scatter provides leases allowing primaries to satisfy reads without communication with the rest of the group, however this can cause delay of group operations. Further this implementation of Paxos does not require writing to a disk and replicas of a key can answer read requests from local, possibly stale, state.

2.3. **Failure Tolerance.** Although Scatter explicitly does not accommodate Byzantine failures, a scatter group with $2k + 1$ nodes guarantees data availability with up to k node failures. Groups ideally are within 8 to 12 nodes to prevent the failure of an entire group at a time. Nodes who enter the system sample random groups to determine which is most likely to fail, and join that group. Overall Scatter is a solid foundation and an improvement on regular DHTs and fancy DHTs [2]

3. Adaptation

Scatter is outlined broadly so this project is an adaptation of the description provided by the paper. The changes made are not vast, but the key change in this implementation is the idea of the primaries. In this implementation groups are responsible for a key range and each node is responsible for that same key range. `get` and `set` operations can be given to any member of the group, specifically the leader, who then proposes the operation in Paxos for sets and just returns the value for `get`s. This change does not harm the integrity of the implementation as long as group sizes remain relatively small. To accommodate this, the implementation has a `MAX_GROUP` feature that can be changed accordingly.

Further the group operations implemented only include:

- `MERGE`: Two adjacent groups joining into one
- `SPLIT`: One group splitting into two (dividing directly into half
- `ADD`: A single node joining a group

- DROP: Removing a single node from a group, most likely due to perceived failure
- ELECTION: Electing a leader to a group

The key-space is redistributed only in MERGE and SPLIT. This implementation uses heartbeats, in the form of PING and PONG messages, to determine which nodes in a group are no longer active. These nodes are failed using DROP until a message is received from them when they are revived with ADD. Leader elections happen at a set interval to keep the leaders changing, and this is a group operation ELECTION because groups know the leaders of their neighbors.

Each node knows its group which includes, primarily, a leader, a list of members, and the key range the group occupies. Each node also knows this information about the group to its left in the key range and the group to its right. This allows a node to forward messages to its neighboring group by simply forwarding the message to the leader of the group based on the key range or members in the group.

In terms of nested consistency, this implementation uses almost a 3 phase-commit protocol rather than a 2PC as it requires acknowledgments for COMMITs as well as get forwards and set forwards. It requires acknowledgments in order to ensure not only consistency, so that all COMMITs are carried out, but also to ensure all read and write requests are eventually carried out even with node failures.

The other variations are the result of interpretations of Scatter that may be different from the actual Scatter, however these variations were unavoidable based on the room for interpretation left open by the paper. The design choices are detailed in the following section.

4. Implementation

Progress of a node in this implementation is made at the receipt of messages, so the majority of the implementation can be described by the types and reactions to message receiving. However, in order to decide when group operations must happen we also recursively checks the status of the group, in what is called housekeeping. The status includes the liveness of members with PING and PONG messages, the need for redistribution with the size of the group, and leader elections. The majority of the work is in nested consensus required for these group operations. However, based on the specification of the project and the manner of testing, some simplifications were made that can be easily dropped.

4.1. **Simplifications.** First, this project oversimplifies the hashing of keys. The project fixes MAX_KEY as 48 and MIN_KEY as 0 for the ease of testing, but this simplification is intended to be dropped and can easily be changed when a real hashing function such as $\texttt{SHA} - \texttt{1}$ is used at each read and write occurrence. This would distribute the keys evenly. For the ease of testing again, the parser expects the specification of a group into which a node is placed (- -peer-names), its key range (- -key-range), its left group (- -pred-group) with a its key range (- -key-range1), and its right group (- -succ-group) with its key range (- -key-range2). Each key range expected to touch, for example if a group is defined to have key range $0 - 16$ its left group must have a key range ending in the MAX_KEY value and its right group must have a key range starting with 16. Further each key range is expected to be within $[\texttt{MIN\_KEY}, \texttt{MAX\_KEY}]$.

These are all design choices only for the purposes of ease of testing and could easily be relaxed. Ideally this implementation would start with a single group defined who is responsible for the entirety of the hashed key space. Nodes would be added individually to that group until it reached MAX_GROUP (assuming $\texttt{MIN\_GROUP} < \frac{\texttt{MAX\_GROUP}}{2}$ since groups merge when they are too small), at which point it would split and the key space would also be split. This would continue to divide up the key range evenly and keep the groups to the ideal $8 - 12$ size outlined by Scatter.

4.2. **Architecture.** Technically there are two classes. A Node is an object containing attributes that will make it a Proposer and an Acceptor in Paxos. It also contains a list, pending_reqs, containing the pending requests it has been passed, a dictionary, PONG, containing a tally (values) for each member of its group (key) corresponding to the number of PINGs it has not received a PONGs from if it is the leader, and the ability to be a spammer (when it sends spam to the entire network). Finally each node has a group, lgroup and rgroup corresponding to its own group object, its left group's group object and its right group's group object.

A group contains a key_range which is a tuple of two keys where we take the first to be included in the range and the second to not be included. Next it contains a leader name, a leaderLease (for leader elections), a list of its members, members, and a proposal number for Paxos, p_num.

4.3. **Housekeeping.** On the receipt of a HELLO message the self.loop.add$_c$allback(self.housekeeping) callback is started. This begins the recursive call to self.housekeeping(self) that happens every TIME_LOOP seconds.

In this function if a node is the leader, it checks for heartbeats of its members. In each loop a PING is sent to each member of the group. If the ping is responded to by a PONG, the recipient detracts from the tally corresponding to the responder in self.pong. Once that tally is 0, the name of that node is removed. If that tally ever reaches 4, meaning a node has been PING-ed 4 times without response, the leader proposes to DROP the node from the group, assuming it has died.

Each iteration a node will also check its name is still in the members of the group it believes it is part of. If its name is not in that list, it will propose to the node it believes is leader to be ADD-ed to the group. This takes care of a dropped node coming back to life and wishing to be part of a group.

Next in an iteration of housekeeping if the node is the leader it checks the size of its group. If that size is larger than MAX_GROUP it proposes, as the Proposer of Paxos, a SPLIT. It also checks if the size of the group is less than MIN_GROUP to propose a MERGE. In merge we also check which of the two choices of MERGE, left or right, would result in a more reasonable, smaller, group size before performing the MERGE.

As mentioned earlier we also save pending requests for get, set, and COMMIT operations in the $self.pending_reqs$ list of each node. In housekeeping we check if that node contains any outstanding requests. Pending sets are stored in the form ("*set*", < *key* >, < *value* >), gets in the form ("*get*", < *key* >) and COMMITs in the form ("*commit*", < *key* >, < *value* >, < *dest* >). In the occurrence of any of these in the queue, they are sent along again. These are checked off, removed from the queue, once they are responded to with a GET_ACK, SET_ACK, or COMMIT_ACK.

Finally there is not a leader for a group, any node will propose an ELECTION as the Proposer in Paxos. Once the group learns its new leader, it will begin the 2PC to inform its neighbors of the new leader.

4.4. **Nested Consistency.** The adaptation of nested consistency works as follows. We take for granted that all messages are passed to the leader of the groups, so sending a message to a group refers to sending a message to the leader of a group. There are a number of types of 2PC messages:
{START, START_PAXOSED, READY, READY_PAXOSED, YES, YES_PAXOSED, YES, YES_PAXOSED, NO, WAIT, COMMIT}. In message passing for 2PC messages, we pass these types as the type field of the json. The key field is the type of operation from the set {MERGE, SPLIT, ADD, DROP, ELECTION}. And the value field varies in the set {SPLIT, < *name* >, MERGE_ID, MERGE_REQ, MERGE_FWD}. We will walk through the various group operations below. Each message also contains a parent who is the leader of the coordinating group.

Notice, we make this a blocking protocol. Upon receiving at START 2PC message, a group will block, by sending itself a Paxos LEARN message, every message from a groups other than its neighbors by responding with a WAIT. On receiving a READY_PAXOSED message a group will block all messages not from the parent of the request, similar to how PROMISEs work in Paxos.

Definition 4.1. Paxos
verb: To *Paxos* is for a group to perform the Paxos consensus protocol to reach a decision about an action.
noun: An island in Greece.
noun: A protocol described by Leslie Lamport in [1].

The following table helps to describe how we manage to pass between Paxos and 2PC.

Type	Key	Value	Parent
START	ADD	< *name* >	
START_PAXOSED	DROP	< *name* >	coordinator
READY	ELECTION	< *name* >	name
READY_PAXOSED	MERGE	MERGE_ID	
YES		MERGE_REQ	
YES_PAXOSED		MERGE_FWD	
NO			
WAIT			
COMMIT	SPLIT	SPLIT	

In general the relation between 2PC messages and the message sent to initiate Paxos within this nested protocol is as follows. However, LEARN messages cannot always fit into this formatting as they each signify a different change in state. For most messages the key used is the type of the 2PC message, the value field is the 2PC key, the who field is the 2PC value and the tpcFrom is the 2PC source.

To begin a MERGE, we send a START message to the coordinator group. The leader of this group Proposes START as a value in Paxos with a key of START. Once consensus is reached within the group, instead of making the group LEARN we send a START_PAXOSED message back to the leader of the group. In general, upon receiving these $-$_PAXOSED messages, the leaders know the decision has reached consensus within the group and it can move on to the next step of 2PC. When the leader receives this message, it sends a READY message with a value of MERGE_ID to the neighbor it *does not* want to merge with. When that neighbor receives the message, it Paxoses the decision within its group and, upon consensus, sends itself a READY_PAXOSED message.

At this point it responds to the coordinator with a YES message. When it gets this message, the coordinator finally sends a READY message with a value of MERGE_REQ for the group it wants to merge with. When that group receives the request, it sends a READY 2PC message with a value of MERGE_REQ to its neighbor on the other side. That neighbor blocks Paxoses the READY and, upon consensus, blocks from everybody except the coordinator. It responds with a YES to the neighbor who sent it. Once the neighbor the coordinator wants to merge with gets this YES with a MERGE_FWD value, it Paxoses the decision to send a YES to the coordinator. Once this is done, it sends a message attached with its other neighbor (not the coordinator) and its own store back to the coordinator piggy-backed on a YES with a value MERGE_REQ. When the coordinator receives this second YES, it Paxoses the store it received from the neighbor it wants to merge with, so each member of the group can extend its own store, then performs the merge that returns a new group. This new group contains no leader. Finally the coordinator sends a COMMIT message containing this new group along to its new side, the side only the neighbor it just merged with knows, and its old side so each can update with a new union neighbor. It also sends a COMMIT message containing both the new group and its old store to the group it has merged with so that group can update its store and group and lgroup and rgroup.

To initiate a SPLIT, a much more straightforward operation, a leader proposes to START similar to as in merge. Once it receives the START_PAXOSED it sends out READY messages to each of its neighbors. They Paxos the READY messages and, when they receive a READY$_{\text{P}}$AXOSED message, send the coordinator their approval in the form of a YES. Once the coordinator receives these two messages, it updates its internal count in $self.okays$ when it receives each YES, it clears its $self.okays$ store and Paxoses the last YES. When it reaches consensus, gets a YES_PAXOSED, it performs the split. The split produces two groups with half the key range and half the members. To each of the members of each of the groups, the coordinator sends LEARN messages containing the updated values of $(lgroup, group, rgroup)$ for them to update. It then sends COMMIT messages to each of the neighboring groups to have them update their internal state corresponding to the SPLIT.

To initiate a ADD or a DROP is fundamentally identical and even less complicated than a SPLIT. Everything in the procedure remains the same except the value field is always the name of the node to be ADD-ed or DROP-ped and the process at the receipt of a YES_PAXOSED.

At this point if the operation is ADD, the coordinator will perform the ADD by adding the new node to the group and sending each member, including the new member, the updated group in a LEARN. Additionally, the store field of the message will contain the store of the group. The new member will update its store to be that store, as will all the other members in the group.

At the point of the receipt of a YES_PAXOSED message, if the operation is DROP, the coordinator will perform the DROP by removing the proposed node from the group.members list and sending the node to remove to all the members of the group in the form of a LEARN message. Each member, upon receiving this message, will remove that member from their $self.group.members$ list.

After sending LEARN messages to all the members of its group, in either ADD or DROP, the coordinator will send its neighboring groups a COMMIT message with a key of ADD or DROP respectively and a value of the node to perform the operation on as well as a which field to indicate the group to which the operation should be performed. The neighbors receive the COMMIT and send every member of their group LEARN messages to update each of their internal states.

Elections are slightly different. They only occur because there is no leader for the group, so there would be no node designated to perform the group operation at all. In this case we elect the leader internally with

Paxos before starting 2PC. If there is not a leader for a group, each node who notices this is the Proposer in Paxos. It proposes itself as the new leader and one wins out. In Paxos if the key is `ELECTION` once the Proposer gets a majority of `ACCEPTED` responses, before it sends `LEARN` messages to the group, it sends `COMMIT` messages with `ELECTION` as a key and its own name, the new leader, as the value to its left group's leader and its right group's leader. Upon receiving these `COMMIT` messages, each leader sends `LEARN` messages to all the members in its group who update their internal states.

The `set` calls are now easy in this structure. When a leader gets a `set` message, it Proposes the key and value. In the learning stage each node sets its `self.store[key]` to the value of the message gets a message because the key, presumably a hashed string, does not match any of the 2PC keys.

In summary, the way we incorporate 2PC and Paxos together into a nested consensus is by Paxosing the decision to send 2PC messages within a group. At the end, once the coordinator group has received a total of two 2PC `YES`s, it Paxoses the final `YES` and receives back a `YES_PAXOSED`. It then performs the group operation and sends `LEARN`s to its whole group so each member can update its internal state according to a very complicated `LEARN` message (it is clear enough from the actual code how this implementation treats `LEARN` messages). It then sends out `COMMIT` messages to all the groups affected by the operation. Those group leaders, upon receiving the message, send `LEARN`s to every member of their own group so each of them can update their own internal state accordingly. All these `COMMIT`s are `ACK`-ed and consistency is preserved.

4.5. **Forwarding.** On this architecture, when a node gets a `get` or `set` request it forwards the request as either a `getRelay` or a `setRelay`, with its own name in the `parent` field, to the leader of an adjacent group who is closer to the key in the key space. When a node receives either type that is intended for itself of `Relay` it Paxoses the value if its a `set` request or looks up the value for the key if its a `get`. The node then forwards that value, in the `value` field, back to the `parent` who originally got the request. That node then sends a `getResponse` or a `setResponse` with the appropriate value.

Additionally to keep track of the requests, whenever a node receives either a `get` or a `getRelay` it adds that request to its `pending_reqs` queue and only takes that same request out of its queue when it gets a `get_ack` with the value it stored for that request. Similarly when a node receives either a `set` or a `setRelay` it stores it in its $\texttt{pending}_\texttt{r}\texttt{eqs}$ in the form mentioned above and only removes it once it receives a `set_ack` with the same request.

4.6. **Failures.** This implementation remains as close as possible to Scatter in hopes that failure tolerance will be equally preserved. Clearly Byzantine faults are tolerated in neither this implementation nor Scatter's implementation.

If the side of groups remains in the optimal range it is unlikely that a whole group will fall. Even though Scatter defines this optimal range as 8 to 12 nodes, realistically, in this implementation since the keys and values are replicated among all the members of a group that optimal range will be much smaller. It is still very unlikely that a whole group will fall at once. Single node failures, or double or possibly even triple depending on the size of the group, should be tolerated ideally since every member of a group will have all the keys and values.

Further if leader elections are frequent enough, a single leader will not carry all the weight of the group and will switch off frequently. We also typically redistribute the leader's store to all the nodes during certain group operations.

By keeping the request queue, we also ensure that every node through which a request passes is responsible for making sure that request is eventually carried out. This ensures consistency further than even specified explicitly in Scatter.

Scatter relies heavily on consistency with fail-stop tolerance, and this project implements a strict interpretation of nested consistency, so ideally it would work just as well as Scatter, however let us prove that with some test scripts.

5. Example Scripts

6. Challenges with Implementation

References

[1] Leslie Lamport. "The Part-Time Parliament," ACM Tras. Comput. Syst., 16(2):133-169, May 1998

[2] Lisa Glendenning et al., "Scalable Consistency in Scatter," *Proc 23rd* ACM Symp. *Operating System Principles*, (SOSP 2011), ACM, Oct. 2011

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishan. "Chord: A scalable peer-to-peer lookup service for internet applications," SIGCOMM Comput. Commun. Rev., 31(4):149-160, August 2001

[4] Antony I. T. Rowstron and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Middleware '01, pages 329-350, London UK, 2001. Springer-Verlag