Github repository: https://github.com/stephanie-wang/salix

Salix: The large file distributed system

by Leo Liu, Neha Patki, Stephanie Wang

May 11 2014

There are several systems that require storage of large files instead of small key-value pairs. For instance, many living groups on college campuses share a centralized server with large media files. In this project, our goal is to provide a highly reliable and available distributed file system that can handle files too large for memory. Salix is able to perform consistently ordered read and write requests for any file. Salix uses the shardmaster and shardkv architecture from lab 4 as a foundation, but with two optimizations. First, it uses the MultiPaxos protocol to more efficiently come to consensus on operations. Second, it automatically load balances the files between different groups based on their popularity among clients. In this paper, we will first discuss Salix's methods for filesystem operations, the MultiPaxos protocol, and finally automatic load balancing. We will then address Salix's failure recovery model and finish by providing benchmark results and future work.

Filesystem Operations

Filesystem operations are built on top of the shardkv architecture from lab 4. Shards consist of filenames instead of keys. "Put" and "Get" requests in lab 4 can be easily extended to "Write" and "Read" requests. In Salix, we also add a new operation called "StaleRead", which allows clients who do not need the most recent version of a file to get their results quickly.

The most difficult challenge in adding filesystem operations is efficient transfer of files between servers during reconfigurations, while keeping all servers in sync and available. Salix aims to support arbitrarily large files, which means we cannot use RPC requests to transfer files. Instead, we set up buffered connections between servers.

While applying a reconfiguration operation in the Paxos log, a server determines whether it is holding any shards that need to be transferred for the next configuration. The server attempts to transfer shards to the new group until a majority of that group has received all files. We call this majority the shard-holders. The sending server records them in a "Reshard" operation and requests the receiving group to propose the operation. When a receiving replica applies the operation, it uses an RPC call to request missing files from a shard-holder in the same group. We prevent a deadlock between shard-holders in the same group by only designating a receiving server as a shard-holder once it's received every shard from one particular sending group. This ensures that even during a reconfiguration, as long as a majority of a replica group is available and in communication, that replica's files will never be lost.

To transfer a file, the sending server dials a server in the receiving group. The sending server sends the configuration number, file length and filename before sending the file contents. The receiving server checks that its current configuration number matches, then writes the file to a temporary location. Checking for configuration number prevents deadlock between two replicas that are designated shard-holders for different reconfigurations. An RPC call to check for missing files is used to ensure correctness after failed transfers due to unreliable network connections or out-of-sync replica groups.

Shard-holders must keep their copies of the shards in a temporary location until they are sure that all replicas in the group have a copy of the shard. To garbage-collect, each replica records the Paxos sequence number of each successful reconfiguration operation. Once the same reconfiguration has been applied on all replicas in a group, all temporary copies of shards for previous configurations can be removed. This is handled by a goroutine that periodically queries paxos.Min() and clears temporary files for reconfigurations with a lower sequence number.

MultiPaxos Protocol

Our implementation of MultiPaxos builds on regular Paxos. In MultiPaxos, instead of having an N_p (highest prepare seen) for each instance, all the instances share a single N_p. Proposal numbers, which include N_p and N_a (highest accept seen), are no longer <integer, peer_id> tuples. Instead, proposal numbers are single integers. The leader of MultiPaxos is given by (N_p % num_peers). Since each proposal number corresponds to a different leader, proposal numbers are also called "views." Views start at 0 and increase monotonically.

Leader Election

If a peer has not heard from the leader within a specified timeout, the peer tries to elect itself as the leader. To do that, the peer determines the next view number for which it would be the leader and sends a Prepare(view) message to all peers. Each peer replies with a PrepareOk if the view in the message is greater than or equal to its current view and promises to ignore future messages with lower view numbers. A peer becomes the leader when it receives a majority of PrepareOks.

During leader election, the new leader must also learn the accepted values with the highest N_a's for every instance. When the leader sends the Prepare messages, it also includes the index of its lowest undecided slot. When peers send PrepareOk replies, they must include the N_a's and V_a's for all the existing slot numbers greater than or equal to the leader's lowest undecided slot index. This step is necessary so that the new leader will not propose its own value if consensus has already been reached for some instance. MultiPaxos learns all the N_a's and V_a's in batch during leader election instead of learning them one at a time as in regular Paxos. By doing this, MultiPaxos avoids the second RTT.

Accept and Decided work similarly as before. Only a leader can send an Accept with a view number for which it is the leader. Non-leader nodes forward their client requests to the current leader.

Automatic Load Balancing

The automatic load balancing reassigns shards to the groups whenever files of certain shards are more popular than others. We define a popular file to be one with many (fresh) read and write requests. Every server in a group keeps track of the total number of read and write requests it performs per shard. We set a certain timeout value so that at every interval, the servers from each group report their popularity scores per shard to the shardmaster. They also report the highest sequence number in their Paxos log that they have applied. The shardmaster waits to hear reports from all groups. If multiple servers from the same group report different popularity scores, then it takes the report corresponding to the highest Paxos log sequence number, since it contains the most up-to-date popularity information.

The goal is rebalancing the shards such that the sums of popularities for the shards in each group are equal. Since this problem is NP-Complete, Salix tries to instead minimize the number of shard transfers such that the most and least popular groups have scores that differ by at most a threshold value. It does this by calculating the groups with the highest and lowest total scores, and then attempting to move the least popular shard to the group with the lowest score. It stops whenever moving the shard does not create a better balance (when the minimum shard has a popularity that is less than the difference of popularities of the groups), or when the threshold is reached. This algorithm also ensures that more popular files are less likely to be transferred during the reconfiguration so that the shardkv servers will be able to continue serving this file. The reduced downtime for these files may be important if the files correspond to trending pages with a majority of requests.

Since the number of shards, timeout, and threshold values can be supplied by the network administrator, we find that Salix provides great load balance flexibility. For example, a greater number of shards leads to more even load balancing, while greater timeout and threshold values correspond to fewer file transfers. Salix also allows the administrator to turn off automatic load balancing. This corresponds to giving all shards a permanent popularity of 1, in which case any further load balancing follows the design in lab 4a exactly.

Failure Recovery

Like lab 4, Salix can keep running if a majority of servers in each group is online. However, unlike the lab, Salix also assumes that a single server can silently fail and restart after losing its memory contents. For this reason, all components of Salix use a write-ahead logging system to record in-memory data on disk. In the shardmaster, each configuration is written to disk, as are popularity scores when they are updated. For shard servers, state changes are written to disk as a redo log. To truncate the log ,the entire state is periodically flushed to disk as a checkpoint.

To test for failure recovery, we simulate a server failing and losing its memory upon the clerk calling Fail(). We simulate the server restarting with the call Recover(), which reads information from disk to achieve the same state as before the failure.

Benchmarks

Automatic Load Balancing

The first benchmark tested the efficiency of automatic load balancing. Using 2 groups, 10 shards, threshold=40, and timeout=2.5s, we designed a test that created 10 files, each with a unique hash that assigned them to the 10 unique shards. The first configuration then assigned any 5 shards to one group and the remaining 5 to another. Next, we took 2 shards, *X* and *Y*, that were being served by the same group, and made them more popular by requesting 1500 read/writes to each, after which automatic load balancing reassigned shards *X* and *Y* to two different groups. We then requested an additional 800 read/writes. We repeated this test 50 times on top of regular Paxos with automatic load balancing turned on and off.

However, on average, automatic load balancing performed slightly worse: 26.26s for automatic vs. 26.18s without. In two of the test cases, the automatic load balancing actually did better by 0.5s. Considering the fact that automatic load-balancing requires complete file transfers, which likely puts server groups out of sync, as well as additional writes to disk for our write-ahead-log, we think this is acceptable.

We realize that to see considerable improvements, we would likely require millions of concurrent requests to *X* and *Y*.

MultiPaxos

Our second benchmark tested the efficiency of MultiPaxos in relation to regular Paxos. We compared Salix's MultiPaxos implementation (without persistence) with our implementation of Paxos from lab 3, focusing on TestMany, TestManyUnreliable, and TestPartition test cases. We set the MultiPaxos failure detection to 1s, ran each of the tests 50 times, and analyzed the results.

We found that MultiPaxos outperformed Paxos in the TestMany case, which assumed no failures. MultiPaxos took an average of 0.27s while Paxos took 0.31s. Similarly, for the TestManyUnreliable case, where failures occur frequently, MultiPaxos took an average of 0.45s while Paxos took 0.48s. Though these differences in our test cases may be small, we assume that we'll see an appreciable difference for a longstanding application.

MultiPaxos failed to perform efficiently in the TestPartition case. This is to be expected because MultiPaxos assumes a leader is stable for a significant amount of time. If the current leader is in a minority partition, then MultiPaxos will have the added overhead of electing a new leader. Our MultiPaxos took an average of 12.19s while Paxos took 6.92s. We recommend not using MultiPaxos if changing partitions are frequent. However, in a network file system setting where the servers are likely close together and partitions are rarer, we expect that MultiPaxos will perform well.

Future Work

Currently, Salix only supports a flat directory structure. In the future, this can be extended to work with an entire hierarchical file system since the keys are simply the file names, and directories can also be treated as files. However, more complex queries like "mv" will require further work since it involves multiple files, which may be held by multiple groups.

The load balancing algorithm attempts to reconfigure the shards such that a single group does not get an overwhelming number of requests. However, the shardmaster does not account for the geographical positions of the clients, so a popular file may still be served by a group that is far away from clients that need it. In the future, it may be worthwhile to explore automatic reconfiguration that factors in location.