CSE 490h – Project 4: Paxos Writeup

Wayne Gerard - wayger
Zachary Stein - steinz

March 7, 2011

1 Nodes

Our system has three kinds of nodes: clients, coordinators, and two phase commit (2PC) coordinators.

Both the number of coordinators in the system and the total number of nodes in the system are required to be fixed. The user must also specify the number of coordinators to assign to each filename in the system.

Coordinators are expected to take on the lowest addresses in the address space, followed by the 2PC coordinators, and any other clients.

The majority of the coordinator address space should be alive at any given time to ensure responsiveness. A majority of 2PC coordinators must be alive in order for the system to successfully abort or commit any transactions - however our current implementation only supports a single 2PC coordinator. The system's availability is unaffected by other client's liveliness.

2 Commands

Node counts can be configured by passing any node the following commands:

```
coordinators <count>, perfile <count>, nodes <count>
```

Clients support the following commands:

```
FS Commands: create <filename>, delete <filename>, get <filename>,
             put <filename> <contents>, append <filename> <contents>,
             listen <filename>
TX Commands: txstart <filenames>, txcommit, txabort
```

3 File System Semantics

Our file system maintains a consistent, distributed log of operations for each file in the system using Paxos. File state is inferred by parsing the log. The logs are currently only stored in memory, and not in persistent storage. This might appear dangerous, but logs can be recovered through Paxos at any point in time (since Paxos state is stored in persistent storage). However, storing them persistently would be consistent and could speed up node reintegration after brief failure.

Operations are appended to the log via Paxos. Operations can be appended to the log as long as a majority of coordinators assigned to each file is responsive (the coordinators for each file are the Paxos acceptors for that file).

Since we use Paxos, appending operations to the log is somewhat high latency. However, reads from the log are serviced locally (and therefore very quickly) using the log entries learned so far. If a consistent read is desired, it can be wrapped in a transaction.

Commands executed on different files are handled asynchronously. Clients are required to verify the validity of commands they attempt to append to the log. Because of this, multiple operations on the same file are handled synchronously, since the validity of executing commands following the first command in the chain depends on the state of the log after the first command is successfully appended. Transaction commands are also handled synchronously and consistently using two phase commit.

Asynchronous command state is maintained in a DAG known as the Command Graph. The graph is responsible for queuing dependent commands, retrying failed commands, and canceling queued commands should one of the commands it depends on fail.

4 Transaction Semantics

We assume that clients will start transactions by calling txstart ⟨space-delimited list files to be touched⟩

For example, client 1 would perform a transaction on files f1, f2, and f3 like so:

```
1 txstart f1 f2 f3
<commands>
1 txcommit
```

If any of the commands inside the transaction timeout, all queued asynchronous commands are canceled and the transaction is aborted.

Since we force the client to list all files they wish to transact on when the call txstart, any commands on files outside of that file list called before the transaction is aborted or committed are not guaranteed to be consistent.

For example, if client 1's transaction looked like:

```
1 txstart f1 f2 f3
1 create f1
1 create f2
1 create f3
1 create f4
1 txcommit
```

Our implementation guarantees that f1, f2, and f3 are created atomically. However, if the create of f4 is invalid, fails, or times out the transaction will be aborted instead of committed. If any other create fails the transaction will also be aborted, but f4 may or may not be created.

5 Paxos

Clients are all proposers. By not using a lead proposer, we eliminate the need to go through leader election, which reduces implementation complexity. We expect the load to be distributed roughly evenly across the different files in the system.

Coordinators could easily detect contention on a specific file and then use leader election to elect a lead proposer with a lease to ensure the liveliness of Paxos. The selection could be propagated to all nodes listening to the file via a log entry. However, we do not support this feature at this

time. The coordinators assigned to each file act as the acceptors and learners for that file. This means that clients do not learn about operations directly, but rather via a coordinator who treats the client as a "listener", and simply tells the client about all values it learns.

If a client has not heard anything from a coordinator in awhile about a particular file and is not sure whether a coordinator has gone down or if simply no operations have been performed, a client can explicitly request to listen to a file using the "listen" command like so:

```
1 listen f1
```

The client could be added to the listener list twice, but this allows the client to discover whether the coordinator has gone down or not. It would be simple to support a stopListening command to correct the case where a listener gets added to two lists.

Coordinators are assigned to files using a simple hash function. The filename's hash code and following integers are used as the addresses of the coordinators. In a multiple data center setting, addresses should be distributed round-robin amongst data centers for maximum reliability. Alternatively, decreased latency could be achieved by assigning addresses sequentially within a data center.

6 Two Phase Commit

Clients can write TXStart (which include the list of filenames used in the transaction), TryTXCommit, and TryTXAbort entries to the log. TXStart entries implicitly lock the log and are required to be written in lexicographic filename order to avoid deadlock.

When any Paxos group is instantiated, the coordinators automatically add the 2PC coordinator as a listener. The 2PC coordinator then monitors the state of transactions through messages sent by the coordinators when a new operation is learned.

The 2PC coordinator tracks transactions by assuming that at any given point, a file can only be involved in one transaction. That is, if client 1 writes:

```
1 txstart f1 f2 f3
```

Then client 2 is unable to start a transaction including f1, f2, or f3. This ensures that at any given point in time, a file is involved in at most one transaction. This also allows the 2PC coordinator to map each individual file to a list of files involved in that transaction. So, in the above example, the 2PC coordinator's file map would look like:

```
f1 -> {f1, f2, f3}
f2 -> {f1, f2, f3}
f3 -> {f1, f2, f3}
```

This allows the 2PC coordinator to draw inferences about the state of the entire transaction based on learning log entries for a single file.

If a TryTXCommit or TryTXAbort is learned about, the 2PC coordinator injects a TXCommit or TXAbort (depending on the message type learned about) to the logs for each of the files listed in the transaction. The 2PC coordinator will indefinitely try to resolve the transaction, and the client will ultimately learn whether their transaction was aborted or committed via Paxos.

If the 2PC coordinator goes down while performing a transaction, it can recover its state by reading the logs when it comes back online.

It is also up to the 2PC coordinator to abort transactions that have been started, but not committed or aborted by their owners. There are any number of reasons this could occur: the client could go down before they have a chance to resolve their transaction, or the client could become unable to talk to the coordinators. Either way, the 2PC coordinator will resolve the transaction by aborting the transaction after a set number of rounds (which should be more than enough to resolve any transaction).

This means that the 2PC coordinator is free to try and inject TXAborts into the log whenever it pleases - the client will learn that their transaction has been aborted and can respond appropriately. However, this mechanism is intended primarily to cleanup after failed nodes - not interrupt them.

7 Comparison to Client Server Architecture

Using Paxos for every operation does increase the number of packets that need to be sent for every operation considerably.

Using our old client-server architecture (without replication), a typical write would look like:

```
Client Write File -> Server
Server Write Forward -> File owner
File Owner write data -> Server
Server write data -> Client
```

Which would be 4 packets, not including ACKs. That same write in our system (assuming the client is already listening to the file) would look like:

```
Client Prepare  -> Coordinators (1..N)
Coordinators (1..N) Promise -> Client
Client Propose -> Coordinators (1..N)
Coordinators (1..N) Accept -> Coordinators (1..N)
Coordinator Learned -> Client
```

Which would be $O(N^2)$ packets in 5 phases, not including ACKs. Though using Paxos for every operation increases the total number of packets sent considerably, most of them are sent in parallel - the number of rounds required is only one higher, and ord old architecture would require additional rounds to perform synchronous replication! The new architecture also provides consistency and high availability, which we feel is a beneficial tradeoff.

This setup also eliminates the need for replication - in fact, there is no explicit persistent storage of files at all. File state is only implicitly stored in the Paxos acceptors persistent state, which ensures that the distributed log is consistent.

8 Outstanding Issues

The persistent Paxos state log currently is never being purged. That is, every operation since the Paxos group's instantiation is stored in the file, even operations that are unnecessary (e.g. if the file has been created, deleted, and then created again, it is not necessary to keep track of the first

create and delete). This could be considered a feature, as the persistent log provides versioning for free - but the log would need to be garbage collected periodically in a real system.