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 this condition is trivially met in our system by only supporting one 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>
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. The implied result of this is that the state of the file is created 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, and operations can be appended to the log as long as a majority of coordinators assigned to each file is responsive, because the paxos group for each file consists of the coordinators.

As compared to regular client-server architectures, appending operations to the log is fairly high latency (since

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 should 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.

The Command Graph

4 Transaction Semantics

We assume that clients will start transactions by calling txstart ¡files to be touched delimited by space¿

So, for example, client 1 looking to perform a transaction on files f1, f2, and f3 would call the transaction like so:

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

The txstart would be added to the file log via paxos, and the 2PC coordinator would learn about the txstart and store this information.

It is worth noting that since we force the client to list all files they wish to transact on, any commands on files outside of that file list, but within the transaction, is not guaranteed to be consistent.

So, 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
```

We do not guarantee that the create of f4 executes, even if the transaction is successfully committed. We consider this to be an error on the client's part if they attempt to access a file within a transaction that they did not explicitly put as part of their transaction file list.

5 Paxos

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

Coordinators could easily detect contention on a specific file and then switch on lead-proposer-mode via a log entry to ensure Paxos liveliness, however at this time we do not support that feature. 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.

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 write TXStart (which include the list of filenames used in the transaction), TryTXCommit, and TryTXAbort messages to the log. TXStart entries implicity lock the log and are required to be written in 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 entire transaction based on learning something about a single file.

If a TryTXCommit or TryTXAbort is learned about, the 2PC coordinator prepares 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 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

Compare guarantees, node usage

Compare Paxos round packet round-trip counts to client server round-trip counts Using paxos for every operation does increase the number of packets that need to be sent for every operation considerably.

Under a standard client-server architecture, 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 would look like:

```
Client Listen File -> Coordinator
Coordinator Added Client as Listener -> Client
Client Prepare  -> Coordinators (1..N)
Coordinators (1..N) Promise -> Client
Client Accept -> Coordinators (1..N)
Coordinators (1..N) Learn -> Coordinators (1..N)
Coordinator Learned -> Client
```

Which would be $3N^2 + 3N + 3$ packets, not including ACKs. Though using paxos for every operation increases the necessary number of packets considerably, it provides consistency and high availability, which we feel is a beneficial tradeoff. It is also worth noting that even though the total number of packets sent is higher, the number of rounds it takes is only 7 for our operation, and 4 for the standard client-server architecture.

Code size and complexity

This setup also eliminates the need for replication, if only because there is no persistent storage for files (i.e. there is nothing to replicate). Paxos ensures that the distributed log is consistent.

8 Packet and Persistent Store Formats

9 Outstanding Issues

The file log currently is never being cleaned. That is, every operation since the paxos group's instantiation is stored in the file, even operations that are unnecessary (for example, if the file has been created, deleted, and then created again, there's no reason to keep track of the first create and delete). One might think of this as a feature - it would be trivial to implement versioning on top of this, where the log line k and all log lines k-1 correspond to version k of the file.