CSE 490h – Project 2: Cache Coherence Writeup

Wayne Gerard - wayger
Zachary Stein - steinz

January 27, 2011

Notes:

There is some overlap in what I wrote below and what you had written. I wrote it partially to try to get it straight in my mind. We should probably reorganize / merge some things (the discussion of replication for instance). I'm also not sure we need to do replication for project 3, only for project 4, so we could put of implementing this and note our plan here.

Commands:

Our system has two kinds of nodes: managers and clients.
Managers currently don't support any commands.
Clients support the following commands:

```
Configuration Commands: manager, manageris <addr>, handshake <addr>
           FS Commands: create <filename>, delete <filename>, get <filename>,
                        put <filename> <contents>, append <filename> <contents>
           TX Commands: txstart, txcommit, txabort
    Dev Debug Commands: debug, noop <addr>
```

All operations performed outside of transactions are semantically implicity wrapped inside of a txstart and txcommit by the manager. However, in our actual implementaion, different code runs if the client is or isn't currently performing a tx for performance reasons.

File System Semantics:

A file is defined to exist and is accessible iff some node in the system created the file outside of a tx, or inside of a tx that has been committed. This file will cease to be available if it is deleted outside of a tx or deleted by a committed tx.

If a file exists, then it is guaranteed that the manager has some version of the file in its persistent storage (although, it is not guaranteed to be the newest version). As with our previous project, we allow one client at most to have ReadWrite access to a file.

Since we are not using write-through cache coherence, we expect clients to maintain responsibility for files they have RW access on. We expect that clients will use replicas to appropriately clone data so that the data can be recovered if the client goes down. This also means that the manager will not request the updated version of these files until another client requests them. At this point, the manager will ask the client to send its changed version of the files forward and to revoke its local permissions on the file(s) (and the manager will revoke the client's permissions on the file(s) as well).

In the event of a client failing while that client still has RW access on a file(s), the manager will transfer ownership to the client's replica and then request the data again.

If no client has ReadWrite access to a file, any number of clients can have ReadOnly access to that file. The manager explicitly invalidates these permissions before granting anyone else ReadWrite access to that file.

Serialization:

We employ client-side file-level locking to prevent a client from requesting permission to the same file twice in a row. For example, say a client doesn't have ReadWrite access to the file test.txt and the client receives the commands:

```
put test.txt I'm about to delete this file!
delete test.txt
```

Since the client gains RW access on test.txt after the first command, there is no need to contact the manager to perform the second command, so we queue the second operation until the fist completes.

The manager, in turn, assumes the client will queue commands until they are ready for the next request to be serviced. If a client requests access to the same file twice, the manager will grant them the appropriate level of access.

Transaction Scheme:

We decided to use two-phase locking manager-side to ensure serializable transactions for the following reasons:

Framework setup Our framework is already suited towards two-phase locking. We already have a locking scheme in-place for cache coherency (project 2), and so the process of locking files was simply expanded to lock files for the duration of a transaction, instead of for the duration of a request.

imistic concurrency We thought about using optimistic concurrency, however this requires the server to validate all requests at the end of a transaction. This would require a log on the server recording transactions. Further, this log would have to timestamp all transactions in order to decide whether two transactions conflicted or not. Lastly, this would also required the server to be sent a copy of the client's transaction log with every commit, which we thought was unnecessary.

Versioning Further, using optimistic concurrency would require some form of file versioning, in order to support rollback. We thought this approach required more space allocation than was necessary or ideal.

Deadlock:

One potential problem with 2PL is the possibility of deadlock.

We require clients to operate on files within a transaction in filename order to avoid deadlock. So, if a client wants to perform the following operations:

```
        Get g => x
Put x => f
```

They have to actually perform these operations:

```
Append "" f
Get g => x
Put x => f
```

Following this procedure, clients are guaranteed not to dead lock while performing a transaction.

The server currently makes no guarantees to clients if they do not perform requests in filename order. Clients can always abort themselves, which results in all locks freeing eventually. It would not be difficult for the manager to ensure clients lock files in order, but we haven't implemented it yet.

Transaction Semantics:

A client is considered to be transacting until it receives a TX_SUCCESS or TX_FAILURE from the manager. The client will queue all commands it is given until it receives a tx response from the manager for its outstanding transaction.

Replication:

In the middle of transactions, in order to avoid potential situations where the most current copy of the file is lost, we decided to implement replication. Whenever a client changes a file locally, it will also replicate this action via RPC on another client (we chose a simple replication scheme, where client 1 replicates on client 2, ... client k replicates on client k+1, ... client N replicates on client 1). This ensures that the latest copy of a file will never be lost, even if a client fails (see below for further discussion on that topic).

Failure Handling:

Server Failures Currently, clients block on server failures. This is within the specification of the assignment, and so we did not implement any handling of this scenario. This will be relaxed by PAXOS in assignment 4.

Client Failures While a client is in the middle of a transaction, they are periodically sent pings by the server (heartbeat pings). If the client ceases responding to this heartbeat, then after a set number of rounds the server will assume the client went down (this functionality was largely borrowed from project 1, where servers were required to deal with client failures). If the server detects that a client went down, it will immediately release all locks this client had on any files and abort their transaction.

The server will then immediately change ownership of any files that client had ownership of to its replica. If there were pending permission requests for this file, then the server will immediately forward that request to the appropriate replica. TODO: HIGH: NOTE: I think we might want to implement this w/ RPC or a different message type for implementation simplicity.

TFS Log Entry Format:

```
<client_address>
<Operation type>
<filename>
<contents_line_count>
<contents>
```

contents_line_count is -1 for operations that don't have contents there is a line separator after each $< entry >$ including the contents tx ops don't have a filename or anything after