CSE 490h – Project 2: Cache Coherence Writeup

Wayne Gerard - wayger
Zachary Stein - steinz

January 27, 2011

Description:

In this project we implemented write-back cache coherence with single copy semantics for use by a small number of clients sharing a single address space coordinated by a single manager. The manger keeps track of the permissions for any given file in the system and assumes that the address space is empty at initialization time. Since the implementations of Transactions and PAXOS in future projects will provide fault tolerance and recovery, we assumed no errors for this project.

Implementation:

We used the standard Ivy cache coherence protocol, creating separate message types for each command (e.g. one for WQ, one for RQ). The manager is implemented on top of a regular client so that it can use the functionality implemented in the first project.

Clients:

When a client calls a read/write command, that client checks its cache to see if it has permissions on that file. If it lacks the permissions it needs to perform the desired operation, it asks the manager for read and possibly write permissions, and queues the action it was trying to take. After receiving the appropriate permission from the manager the action is completed. If an error occurs (e.g. the file requested does not exist) the operation is thrown out.

Managers:

The manager keeps track of client cache statuses using a couple data structures. We could have had the manager poll clients on every request to determine their status and determine who needs what request sent out, but this method is much faster and reduces the number of packets sent. Unless some client has ownership of a file, it is assumed that the manager has the latest revision of any file.

Outstanding issues:

Our deletion and creation mechanism(s) are not as smooth as they could be.

First and foremost, we allow the client to delete files it has RW access to without informing the manager immediately. If another node requests access to this file, then the manager requests the client to send the data back. The client does this by sending a blank file back to the manager, which the manager assumes is an implicit delete. However, this is currently indistinguishable from an empty file. As a result, we are defining empty files to be essentially non-existent. This isn't particularly important except that it would make the API to our library a bit unnatural. We plan to fix this in the near future.

Our plan to fix this problem is summarized briefly as follows. In the event that node 2 deletes file x, and then node 0 requests file x, node 2 will send a special command to the server indicating that it was deleted. The server will then return a FILE DOES NOT EXIST error to node 0, which node 0 should take as an indication that file x does not exist anymore. However, currently node 0 will keep whatever cached copy it has. It will always receive an error that the file does not exist if it asks the server for read and/or write permission, but will still have a cached copy of file x.

Creation is equally troublesome, as currently the manager has no way to distinguish between commands send by the client. If a client does not have proper read/write access to a file, it just sends a generic WQ to the manager. The manager assumes the client would not try to append or put to a file that it did not intend to either create first or that existed somewhere else. In that vein, if the manager cannot find a local copy of the file it will assume that the client intended to create that file, and so it will create a local copy and then send an empty file back to the client. As a result of this, appends and puts can succeed on non-existent files. This could also be fixed by an extra message type.

Additional steps:

One thing that is necessary before each run is to clean the local storage system for each node. Without cleaning the local storage system, the user will likely encounter errors related to file existence. For convenience, we have written a script "clean.sh" that should be run before using CacheCoherenceTester. We do not consider this to be an error, since it is assumed that there are no faults, the client and server should have no shared state between runs.

While the user doesnt need to explicitly make WQ, RQ, etc. requests in order to use our cache coherent client, there is an additional setup step that the user needs to take before using our implementation. In addition to our previous fault tolerance mechanism of handshaking, it is also necessary to explicitly define who the server/manager is for each node. This is done by calling up the command $< NODE >$ manageris $< SERVER >$. So, for example, for three nodes, each script should start with the following set of commands:

```
START 0
START 1
START 2
time
0 manager // establishing that 0 is the manager
1 manageris 0 // inform all clients
2 manageris 0
0 noop 1 // handshaking
2 noop 1
time
time
time
time // allow 3-4 rounds for handshaking to take place to avoid dropping commands
```

Alternatively, if no more than 4 nodes are desired, the PerfectInitializedClient can be used to bypass handshaking and manager establishment. Node 0 will always be the manager in this case. Note that you still have to start the nodes you would like to use.

State Machines:

(See Attached)

Discussion:

Synoptic:

We were successful in generating Synoptic traces of our Protocol running partitioned by node. Unfortunately, these graphs are tall and unwieldy since they maintain the arbitrary high-level operation order imposed by the specific script run as invariants in the final graph.

We are still in the process of generating the ideal Synoptic graph of our protocol partitioned by high-level operation. The CacheCoherenceTester has been written to this end, but is incomplete. It works by performing many random high-level operations (create, delete, get, put, append) at random on random files. It currently works if no attempts are made to perform operations on files that don't exist, create files that do exist, and do other non-sensible things. The user defined Synoptic even it logs currently occurs at an improper time, making it impossible to partition the Synoptic log into the desired traces. This should be fixed soon using Callbacks.

We also considered generating several random scripts, each containing a single high-level operation to be run, and partitioning by log-file generated. Unfortunately, this technique would fail to capture several important characteristics of our protocol since cache data structures would be cleared between each run and we currently have no way to recover them between rounds. The easiest fix for this would be for the server to assume it has the newest version of every file it knows about upon initialization and for the clients to invalidate all of the local files in their cache. However, the CacheCoherenceTester is likely to be more extensible and robust, so we will concentrate our time trying to get it working instead.