Abhishek Choudhary
Ryan Oman
Brad Bicknell

Project 2 Write Up

CSE 490h

Description:

In our implementation of cache coherency, we added a 'cache coherency' layer. The purpose of this layer is to handle all RPCPackets to keep the file permissions consistent between all the clients. This layer acts as a bridge between DistNode and RIOLayer, and when a node takes an action which requires changing of permissions between client we handle it here. The cache coherency layer then passes an RPCPacket to the RIOLayer which wraps that packet in a RIOPacket and sends it out. It uses RPCPackets to communicate with the other cache coherency layers, telling clients when they get permissions to a file.

The cache coherency layer takes a DistNode (the client) and creates a RIOLayer which it forwards the packets to. It also creates a cache which is a HashMap mapping filenames to actual Files. When it receives a RPCPacket it determines if the recipient is a server or a client by checking if the address of the recipient is 0 (address for server). If it is a server it forwards the packet to masterRecieve() which executes the RPCPacket. Otherwise it forwards that packet to slaveReceive() which executes the RPCPacket. We do this since server functionality is very different then client functionality. Each of the methods masterReceive () and slaveReceive() uses the passed packet's protocol to determine what kind of action to take. The cache coherency layer is the layer that contains all the methods for each action type (create, get, put, etc.) which are called by masterReceive() and slaveReceive(). It also has a getFileFromCache() that takes a filename and returns that File.

We also added a File class and MasterFile which extends File. Clients use the File class and servers use the MasterFile class. A File has 3 types: INV (invalid), RO (read only), and RW (read write). The file type specifies the permissions that the client has on that file. If a client takes an action that would require new permissions (such as delete), he sends a request to the server and when the server responds the client sets the new permissions and can complete the action. A file also maintains a queue of commands that are waiting to be executed on the file. This way commands don't get lost on files that are already in the process of executing a command. MasterFile is used by the server. It keeps a HashMap for that file which takes and address as a key and uses permission types as a value. This way the server can easily check what permissions a client has on a file. It also has a getUpdates method that returns a HashMap that takes a permission type as a key and has a list of addresses as the values showing which clients need to update their permissions.

Assumptions:

Our two biggest assumptions are that we pretend servers and clients don't crash so that we can focus solely on consistency between users. Also, we are assuming that the server starts with no files. Every file has to be created on server since the server does not save state between stopping and starting.

Abhishek Choudhary
Ryan Oman
Brad Bicknell

How to Use Implementation:

Running this implementation is the same as last time, except we got rid of the 'server' parameter in all the commands.

Outstanding Issues:

Nothing except for the assumptions stated earlier.

Synoptic

The two Synoptic outputs are actually pretty similar even though there were dozens of bugs we fixed between the two states of the implementation. When we created the first graph we couldn't really find any bugs in it (or didn't notice the bugs in the graph) so we didn't use it as a tool for bug finding. Instead we tested for bugs on a case by case basis with the Eclipse debugger which was very handy. Looking back though, we did find a bug in the graph. If you follow the append path in the final graph you can see that it goes from append → send → write. In the old graph the append → send, but dies instead of passing it onto write. This was the clearest difference, but there might be more that we didn't see.