CSE 490h Assignment 2

Colin Scott and Bill Cauchois

February 11, 2011

In this assignment we extended our RPC layer to support several different types of operations in order to implement a write-back cache coherent distributed storage algorithm coordinated by a centralized manager.

1 Cache Coherence

Our cache coherence algorithm was based on the IVY system described in the paper we read for class. It is *write-back*, meaning that writes are not immediately propagated to the rest of the system – instead, the propagation is delayed until another peer explicitly requests access to the file. It is *linearizable*, meaning that every operation is visible to the rest of the system in a consistent total ordering. By *coherent cache* we mean a store of files in which repeated reads to the same files will yield the same result, unless there is a write.

We achieve these properties through the use of a *centralized manager* – a node whose job it is to process operations from *clients* while ensuring that the proper invariants are preserved and keeping track of some global state.

Note that each client maintains a *file table* which contains, for each file, a lock (to ensure that the client only performs one operation on a file at a time) as well as an access modifier. File access is one of Invalid, ReadOnly, or ReadWrite.

1.1 Centralized Manager

The main invariant that the centralized manager preserves is as follows. Either a file has not yet been created, in which case all clients have Invalid access to that file; or, the file has been created and one client has READWRITE access to that file, in which case every other client has INVALID access; or, the file has been created and one or more clients have READONLY access to that file, in which case every other node has INVALID access and no one can have READWRITE access.

The global state that the centralized manager maintains is as follows. In the manager's own *file table*, for each file, we have an *owner* and a *copy set*. A file always has exactly one owner, who has either READWRITE or READONLY access to that file. The copy set for a file consists of those clients who have READONLY access to the file.

1.2 Message Types

Several different message types are used to synchronize state between clients and the centralized manager, as well as to transmit data. In our system, communication between nodes may never occur – all communication goes through the centralized manager.

- An INVALIDATEREQUEST may be sent from the manager to a client to tell that client to mark its access for the specified file to INVALID.
- An INVALIDATECONFIRM is sent from a client to the manager to indicate that it received the INVALIDATEREQUEST and has performed the invalidate successfully.
- A READQUERY is sent from a client to the manager when a client wants to read a file but does not have access.
- A READFORWARD is sent by the manager to the owner of the file (a client) when it receives a READQUERY from another node.
- A READDATA may be sent in two cases: when a client receives a READFORWARD for a file to which it has access, in which case that client sends back a READDATA with the contents of the file; or when the server receives a READDATA from a client after sending that client a READFORWARD (in response to a READQUERY), it sends a READDATA back to the client that originated the query.
- A READCONFIRM is sent by the client after it has successfully sent a READQUERY and received a READDATA in return. The purpose of

this message is to indicate to the manager that the read has completed and it can release its lock on the file.

- A WriteQuery is analagous to a ReadQuery, except in this case we are requesting write access to the file.

- A WriteForward is sent by the manager to the owner of the file when it receives a WriteQuery from another node. Since when we grant ReadWrite access to the originator of the request it is an invariant that every other client must have Invalid access, a WriteQuery is also an implicit invalidate.

- A WriteData may be sent in two cases, analagous to ReadData: by the manager in response to a WriteQuery, or by the owner in response to a WriteForward.

- A WriteConfirm is sent by the client after it has successfully sent a WriteQuery and received a WriteData in return. The purpose of this message is to indicate to the manager that the write has completed and it can release its lock on the file.

Message types we used not from the IVY paper include:

- A CreateRequest is sent by the client to create a file.

- A CreateConfirm is sent by the manager to indicate to a client that its create request has succeeded.

- A DeleteRequest is sent by the client to delete a file.

- A DeleteConfirm is sent by the manager to indicate to a client that its delete request has succeeded.

1.3 Local Operations

It bears mentioning that after a client has received a copy of a file from the manager, it writes that file to disk. Subsequent reads and writes will use the local copy of the file instead of invoking remote procedures, for efficiency's sake (as long as the client has the proper access).

1.4 Manager as a Client

Note also that the manager can act as a client. That is, it can perform every read/write operation that a client can perform. However, the manager does not keep a "master copy" of the cache. When it needs to read or write a file, it must send a WriteForward or a ReadForward. An optimization is that the manager does not first send a WriteQuery or a ReadQuery to itself – in every case where a client would have to communicate with the manager, the manager (acting as a client) can eliminate the middleman.

1.5 Example Scenarios

We present several examples to illustrate the functioning of our system. Assume for each example that 3 nodes have been started – node 0 is the manager, and nodes 1 and 2 are clients.

Example 1

Action	*Explanation*
`1 create foo.txt`	Client 1 sends a CreateRequest to the manager, who responds with a CreateConfirm. In the case of a create, the client that initiated the create is set to the owner and receives read-write access.
`1 get foo.txt`	Client 1 tries to read from foo.txt, and finds that it already has read-write access. Therefore it reads (the empty string) from its local cache on the filesystem and returns immediately.
`2 get foo.txt`	Client 2 tries to read from foo.txt. Since it does not have a local copy (its access is set to Invalid), it must send a ReadQuery to the manager, which sends a ReadForward to client 1 (the owner of foo.txt). When client 1 receives this ReadForward, it forwards the contents of foo.txt using a ReadData message. The manager receives this message and sends it back to client 2.

Example 2

Action	*Explanation*
`1 create foo.txt`	Client 1 creates foo.txt, as before.
`1 put foo.txt hello`	Client 1 tries to write to foo.txt, and finds that it has write access. Therefore, the operation completes locally with no external communication.
`2 get foo.txt`	Client 2 tries to read foo.txt, and finds that it has no access – so it sends a ReadQuery to the manager, who sends a ReadForward to client 1 (the owner). The ReadData messages are propagated back and two other things happen: client 1 changes its access from ReadWrite to ReadOnly (since it is no longer the sole owner), and the manager adds client 2 to the copy set for foo.txt. Now client 2 has ReadOnly access.
`2 put foo.txt bye`	When client 2 tries to write to foo.txt, it must send a WriteQuery to the manager since it only has ReadOnly access. The manager, upon receiving the WriteQuery, invalidates foo.txt for every member of the copy set. Then it sends a WriteForward to the owner of the file (client 1) who responds with a WriteData that is then forwarded to client 2, who now performs the write locally.

Example 3

Action	*Explanation*
`1 create foo.txt`	Client 1 creates foo.txt, as before.
`2 put foo.txt hello`	Client 2 tries to write to foo.txt and finds that it has no access. Therefore it sends a WriteQuery to the manager which results in a WriteForward being sent to client 1 (the owner). The manager invalidates 1's copy of foo.txt and forwards the WriteData to 2, who then performs a local write.
`0 get foo.txt`	This case is slightly different because the "client" is the manager. Where a normal client would have to send a ReadQuery, the manager can directly send a ReadForward to the owner of foo.txt (which is now client 2).
`0 put foo.txt bye`	Again, the manager can omit sending a WriteQuery since it knows the owner. So the manager first invalidates the copy set of foo.txt, then sends a WriteForward to the owner of foo.txt (still 2), and, after receiving a WriteData, performs the local write.

1.6 Major Differences from IVY

There are several differences between our implementation and the cache coherence algorithm described in the IVY paper.

Perhaps the biggest difference is the addition of several new message types to support create and delete operations. The IVY protocol was for virtual memory systems, so it could assume that pages were allocated across all machines. However, since we are dealing with a file cache we must handle the creation and deletion of files. Creating a file is simple – the client sends a CreateRequest to the manager, who then sets the owner of the file to that client and responds with CreateConfirm. Deletions are slightly more complicated, as the manager must send an InvalidateRequest to every member of the copy set to ensure that no one can read the file after it has been deleted.

We also simplified the "improved centralized manager" algorithm in the paper so that all communication happens through the server. In IVY, peers may communicate with each other for increased efficiency. In our system, peers only communicate by sending messages to the server, which may then be forwarded on to other peers.

2 Changes to the RPC Layer

We made several changes to the RPC layer for this assignment. We used the RPC layer exclusively to transmit messages used by the cache coherence protocol.

The biggest change was our switch to Protocol Buffers. We decided that instead of rolling our own encoding for remote procedure calls, we would use Google's Protocol Buffer library to serialize and deserialize packets. Our message types are all defined in `proj/RPCProtos.proto`.

The basis of our new RPC encoding is the `RPCEnvelope` message type, which encapsulates the type of the call, the call ID (a new field), the error code (if an error occurred), and a variable-length sequence of bytes (the "payload", another message encoded with protobuffers).

Another addition is the call ID field. The RPC client assigns every outgoing call an integer identifier (by simply incrementing it each time) and when the RPC server sends its result, it attaches that same call ID. This is used to associate calls with their results (using a HashMap in `RPCClient`), and is integral to the new handle mechanism, which is covered in the next section.

3 New Abstractions

We implemented several novel abstractions which make it easier (we hope) to deal with the asynchronous, callback-oriented nature of our code within the simulator framework.

3.1 Handles

A handle represents a "promise" for a computation that has yet to be completed. Handles are parameterized on their result type.

For the calee, using a handle is a simple as creating one inside your function and returning it. Then, call `completedSuccess` with the result value when your computation has completed – or call `completedError` with an error code.

For the caller, using a handle involves registering a listener on returned handle. When the computation completes, your listener will be activated.

Handles allow us to effectively utilize delayed (as by the network) computations.

3.2 Locks

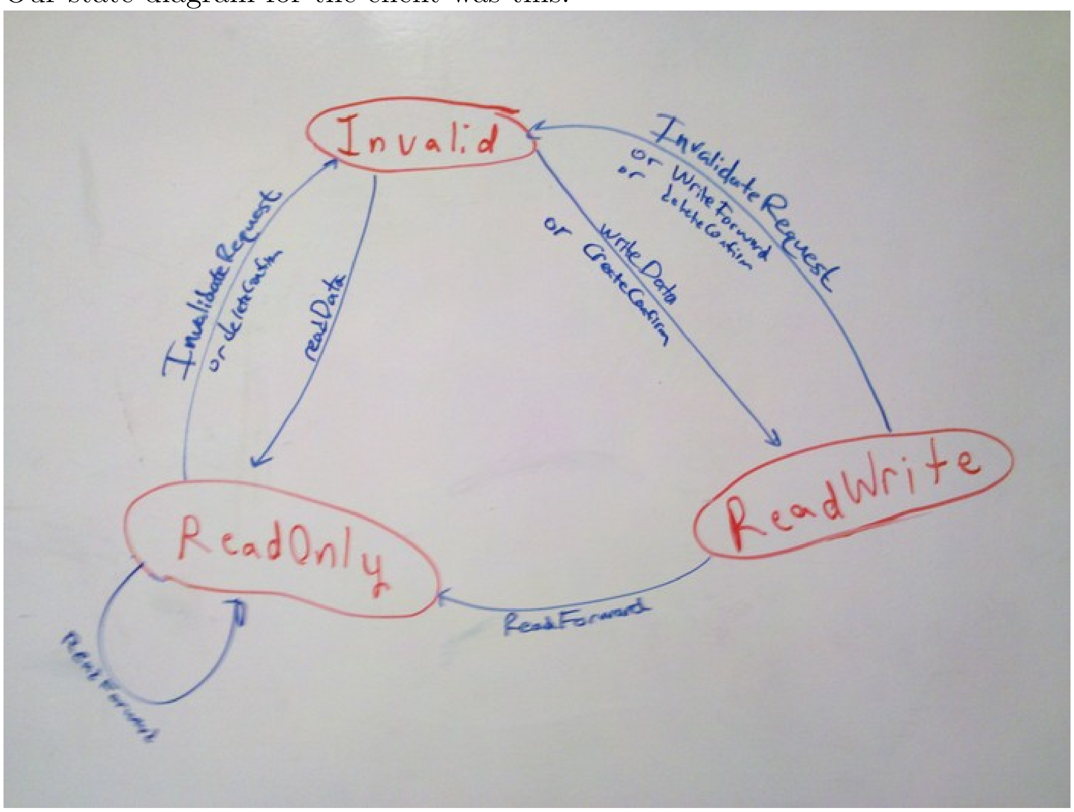
Locks are used to provide exclusive access to a resource. When acquiring a lock, you provide a callback function to be executed as soon as the lock is free. If the lock is already free, your callback is executed immediately. But if the lock has been acquired by someone else, your callback is queued until the lock is available. After you've finished using the resource, you must explicitly call `release()` inside your callback to release the lock.

4 States and Synoptic Analysis

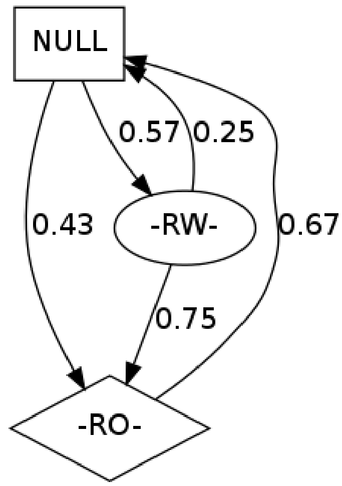
We drew a state diagram to model the behavior of our system, and we also analyzed its behavior using Synoptic.

Modeling the behavior at the client involves a simple set of three states, representing the different types of access that a client can have to a file, with arrows between them corresponding to different message types. Unfortunately, we weren't sure how to model the behavior of the server using a state diagram since (except where it acts like a client) the state that it has (owner and copyset) is not finite and discrete.

Our state diagram for the client was this:



Our state diagram as a result of running synoptic on our logs was this:



As you can see, the two diagrams roughly correspond. Although the labels in the second diagram are not labeled, we can see the correspondence by considering the number of incoming and outgoing edges for each "vertex". The read only state has two incoming edges and one outgoing edge (to invalid) in each diagram. The read write state has two outgoing edges and one incoming edge (from invalid) in each diagram. The invalid state has outgoing and incoming edges to both read write and read only.

Although we were supposed to use Synoptic before we debugged, we didn't get to that part of the project until after we had fixed most of the major bugs. Therefore we cannot provide a "before" Synoptic diagram to complement our "after". However, we can confidently say that Synoptic helped confirm the correct behavior of our program, and certainly made us feel better about our code.[1]

[1] Thanks Ivan!