Distributed Locking For A File Server Using The Spread Toolkit

CS6210A: Advanced Operating Systems
Fall 2007 - Project 4

James Richard Bailey (richard@magichappyplace.com), Chris Kelly (chris.kelly@oit.gatech.edu)

Overview

The goal of project 4 was to implement Calypso-like token management and arbitration using spread.org as middleware. We feel that we meet all requirements of this assignment in our implementation, but certainly do not cover all aspects of the Calypso system (such as automatic discovery, crash recovery, etc). Our system can handle multiple clients accessing multiple files with exclusive write locks and shared read locks that are downgraded or revoked as necessary to facilitate a new lock request. C was used for the entire project, Spread is used as the middleware for handing locks, and the actual file server interaction occurs using a basic file server and client api using TCP sockets.

Organization, How To Build, How To Run

The spread daemon must be installed and running on the local machine (we assume the port is 4803). If you'd like to change this, change the information: GROUP_NAME, SPREAD_NAME in proj4.h. Depending on how the spread daemon is configureg on your machine, you may need to change SPREAD_NAME in proj4.h as well to something appropriate for your installation (check proj4.h for the port number that our project is expecting the Spread daemon to be running on). To build:

```
$ tar -zxf bailey_kelly_p4.tar.gz
$ cd bailey_kelly_p4/
$ ./make.sh # This is a bash script that performs the following actions

   cd fserv
   make # build the fileserver
   cp fserv-api.o .. # copy the api to interact with the remote fileserver down
   cd ..
   make # build the lockserver and the clients
```

Then to run, do each of these in separate terminals:

```
$ ./fserv/fserv 88888        (this starts the file server, ctrl-c or SIGINT to exit)
$ ./lockserver               (this starts the lock server, ctrl-c or SIGINT to exit)
```

To run the clients, you may run one instance of client1, client2, and client3. For each one, pass in parameters for some file or combination of files, ex:

replace fileserver-host and fileserver-port with the host and port that fserv is running on

```
$ ./client1 filename.txt R fileserver-host fileserver-port
$ ./client2 filename.txt W fileserver-host fileserver-port
$ ./client3 filename.txt R fileserver-host fileserver-port
```

View filename.txt to see the contents that was written (random string of characters).

After a client quits due to a revoked lock, you may use that client in a new connection.

Functionality and Caveats

We feel that our implementation fully meets the spirit of this project. Firstly, token arbitration works:

- read requests result in any write locks being downgraded
- write requests result in any write or read locks being revoked

Files (currently just garbage data) are written to and read from the file server only when the proper lock is acquired, and files can be written in seek or append mode (though this is not explicitly exposed in the testing client that supports distributed locking).

If multiple clients are attempting to lock on the same files at the same time, there is a possibility that messages may get "out of sync" and some clients will fail to properly acquire their locks because the in-transition flag was not implemented. This will result in failing to acquire the requested locks, and the client that times out can simply re-try to acquire it's lock. Retry capability was not implemented in our client, but due to the "demo" nature of the client, someone using it can see that it fails to acquire the lock, send it a SIGINT, and try again. However, we were unable to actually see this issue in testing so it may require a larger scale system than is reasonable for us to easily test.

Additionally, buffer sizes, file name lengths, etc, are all specified in proj4.h. It's likely that they many or all of them may be raised but testing was done with small values for everything. Certain combinations of very long group names, file names, host names, etc, may lead to unexpected results.

Performance

As expected, everything performs fairly quickly. Locking transactions are near instant and the entire process of requesting a write, receiving a list of clients, sending them revoke messages, receiving the acks for each, sending a revoke complete to the lockserver, and receiving the ack is not measurably different from a simple request write followed by an ack. Further testing on many more machines would likely show some differences, but our tests show that the speed of this system will be more than enough for most situations.

Due to the construction of our system, there are only several points where scalability may be an issue:

- Spread limits the size of messages. They are working on increasing this, but a read lock shared by many thousands of clients may lead to the lockserver attempting to send a client list to a client requesting a write that is too large to fit in a single message.
- Dynamically allocated linked list are used to manage lists of files and locks in both the lock server and the clients. As load increases, this will not scale and should be replaced with something using trees or hashtables that are allocated in large chunks instead of one record at a time (and keep memory for some time after an item is removed instead of immediately freeing it)

Otherwise, performance is truly limited by the message bandwidth that Spread can provide. Any limitations of Spread will be visible in our system under load, and any future improvements to spread will directly increase the capacity of our system.

Conclusions

This was a neat project to work on. Spread is a very useful toolkit (we actually looked at using it for the previous project before realizing that it would not be allowed), and it's fun to see it in action. However, it was likely not the best tool to use for this specific situation:

- We ended up implementing unicast messages over Spread's multicast system

- Addressing support was implemented on top of Spread to allow this
- All clients likely receive all messages (Spread may do some intelligent server side filtering of this, but we did not try to measure this)

We do have some thoughts on better ways to use spread here, but they were beyond the scope of what we had time to experiment with:

- Clients listen on the wire for messages relevant to them based on tokens and act appropriately (different protocol pattern than in the Calypso paper):
    - a write client sees a read request and automatically downgrades (and the client requesting the read sees the downgrade)
    - a client sees a write request and automatically revokes it's own lock (and the client requesting the write sees the revoke)
    - etc
- Multiple lock servers are used for redundancy, with each storing it's own information and clients ignoring duplicate messages. This likely would need to be done in conjunction with the previous option.
- Some other groups seem to have used a different spread group for each file but we feel that this may not scale very well in the long term as there is likely more overhead for joining and leaving groups than for simply exchanging one message.

However, even considering all of this, we feel that our implementation is still surprisingly useful for a 2-week long project, and should be able to scale to larger numbers of clients than we are able to test.