Dwayne Charp
CS 550
10-29-2013

File Consistency

Introduction:

This programming assignment will be an extension of program 1 and 2. The new features added will be file consistency across all peers. The system will be implemented in a push and pull manner. In the Push method, the server notifies other Peers that a file is no longer valid. In the pull method the Peers are responsible for asking the origin server if a file is still valid.

High-Level Overview:
The basic operation of the system is as follows:

1. The peers will register with the naming Server
2. Each client will then act as a remote object and a Fileserver
3. Once all of peers are up, they each will attempt to get proxies for each neighbor
4. All of the objects will communicate through Pyro4 proxy objects, but the file downloads will be executed from a Threaded TCP server

The main data structures that I used where Dictionaries and List. Dictionaries simplified a lot of the bookkeeping necessary in the system.

PULL Mode:

For the pull mode I implemented Python threading timers. When a peer downloads a file, it gets the ttr information for that file and sets a timer that will be triggered in another thread. When the timer goes off, the peers queries the origin of the file to see if the file is still valid. If it is it resets the ttr, otherwise the file is marked as invalid. From the servers perspective it responds to these request by sending -1 if the file is invalid or the new ttr if it is valid. The user can simulate toggle the state of a local file from the command prompt.

PUSH Mode:

In Push mode the server is responsible for sending out invalid messages. The mechanics of this message are the same as the query messages, except different functions handle them. The file state can also be toggled from the command line.

Not much effort is needed to toggle between modes. Just switch the mode state in the PeerDriver.py file, and the logic will handle the rest.

Python:

As I mentioned I decided to use Python as my programming language. I will also be using a library similar to Java RMI called Pyro4. Pyro4 is a library that allows for remote method invocations. I was initially going to use my own proxy object and create sockets, but this is much cleaner and allows for my attention to be focused on the actual operation of the system. Python also provides a SocketServer framework that allows the simplification of making a networked server. This time around I made user of Python timers to allow execution of helper threads that maintain the files state.

The Peer has been modified to allow for the extra functionality needed. Mainly I added functionality for both the push and pull methods.

1. The Peer gets the file name and generates a message id.
2. The query(messageid,TTL,filename,portnumber) is sent to all neighbors, and the messageID and filename are stored in a dictionary.
3. Each neighbor then gets the message and then searches for the file and forwards it also.
4. If a hit is made, then I hit query is sent on the reverse path. But this time the Peer who hit, sends its port number in the message.
5. On each node back the Peer checks to see if they sent for that hit query, if it did then they store the port number in a list that is the in message sent dictionary
6. Now the Peer can look at the messages sent list and pick a Peer to download from
7. Before a hit query is sent out though, the files state is check, if it is valid then the message can be sent, otherwise it is ignored.

This is some of the new logic added:

QueryHelper.py

```
####  These methods are used for push mode ####
   def invalidate_message(self, imessage):
      #if ive seen this message return
      #invalidate file if i have it

      messageId = imessage.msg_id
      if self.client.messages_received.has_key(messageId) or
self.client.messages_sent.has_key(messageId):
         return
      sender_info = imessage.origin_server
      if self.client.ip_address == sender_info:
         self.client.messages_sent[messageId] = [imessage.file_name]
      else:
         self.client.messages_received[messageId] = sender_info

      self.client.meta_data.invalidate_file(imessage.file_name)
```

```
        self.send_invalid_message(imessage)

    def send_invalid_message(self, imessage):
        for peer in self.client.peers.values():
            peer.invalidate_message(imessage)

    def generate_invalid_message(self, f):
        im = InvalidMessage.InvalidMessage(self.client.generate_next_message_id(),
self.client.ip_address,f.name, f.version_id)
        self.invalidate_message(im)


#### These methods are for pull mode ###

    def is_file_valid_on_origin(self, file_name):
        print "Asking server if file is valid"
        f = self.client.meta_data.get_file(file_name)
        peer_id = int(f.owner) + 9000
        peer = self.client.peers[peer_id]
        return peer.is_file_valid(file_name)



    def set_timer_for_refresh(self, file_name, TTR):
        print "setting timer for refresh"
        t = Timer(TTR, self.refresh_file,[file_name])
        t.start()

    def refresh_file(self, file_name):
        print "refreshing file"
        response = self.is_file_valid_on_origin(file_name)
        if response == -1:
            print "file was invalid"
            self.client.meta_data.invalidate_file(file_name)
        else:
            print "file was valid"
            self.client.meta_data.set_ttr(file_name,response)
            self.set_timer_for_refresh(file_name,response)


    def is_file_valid(self, file_name):
        print "someone calling for file state"
```

```
f = self.client.meta_data.get_file(file_name)
if f.state == "valid":
    return f.get_ttr();
else:
    return -1
```

I also added a new invalid message type.

Further Improvements:
In the future I could refactor some of the code to make the logic cleaner and make some of the classes less cluttered. It would also be nice to allow the peers to operate in separate modes, so that some peers can query servers and others could be passive.

Vertification:

I used print statements to verify that the peers and servers were responding to the invalidate messages and file state request. The rest of the operation is similar to program 2.

Manual :
The mode needs to be set in the PeerDriver.py file at the top

The only dependency that Python needs is PYRO4. This is include in the source folder. Navigate to the Pyro4 folder and run Python install(install like any other python module). This will install Pyro4.

For every directory that you are running the server and Peer in, include all of the .**py** files. I have made 2 peer folders with all of the necessary files. Just run python PeerDriver.py in each.

In the PeerDriver.py, the topology can be changed. This is towards the top of the file.

The PeerDriver also needs to be set to push or pull prior to operation.

The topology files are formatted as follows:

Mesh:
Peer1,n2,....
Peer2,n1….

Each Peer listing in the MESH topology needs to be symmetric for the clients. I.E -> If peer1 knows about peer2, then the reverse needs to be true.

Before starting the Peers we will need to start the Pyro4 naming server. To do this enter the following 2 commands:

```
export PYRO_SERIALIZERS_ACCEPTED=pickle
python -m Pyro4.naming
```

The first command uses pickling as the format for transferring the files. The second one starts the the naming server.

Now run the all of the Peers
All of the Peers need to be up before any system operation begins. Right now I have the configuration file to setup up **2** peers. Once all Peers are up, run get Peer proxies from the command line for each Peer. Once this is done, normal system operation can take place.

Final Thoughts:
The overall implementation of this was straightforward. I did not need to modified too much of the existing code from program 2. I was able to add the file consistency functionality in the peer QueryHelper and MetaData files, and I added a few extra checks where the query messages are being sent out. I was kind of busy when this was due, so I did not clean it up as much as I would like to have.