CSE 490h – Project 1: Client-Server File Store Writeup

Wayne Gerard - wayneg
Zachary Stein - steinz

January 13, 2011

Description:

We decided to pass commands by extending the packets existing protocol field. This involved adding additional protocol identifiers to the Protocol class. We also considered packing the command into the payload as a string, but decided this which would be unnecessarily fragile. Or approach makes parsing simple without complicating packet payloads. In fact, onCommand sends requests by mapping the given command string to the correct protocol identifier using the Protocol classs static stringToProtocol method. This means that additional commands can be added by editing only the Protocol class (unless custom parsing is needed).

The file operations were implemented in the obvious ways, using the Utility class to check for file existence, printing the proper error codes on failure, and sending responses to the requesting client after completion. Put was implemented as suggested in Vincents email to deal with Javas FileWriter quirk.

RIO Fault Tolerance:

To handle faults, we decided to implement a handshaking mechanism. When any node comes alive, it generates a UUID (which is universally unique) and uses that for its entire lifetime. If we have two nodes A and B that have never communicated previously and A wishes to send a message to B, A will first send a junk command to B (Ideally Protocol.NOOP, but any command will work). B will recognize that A does not have its current UUID, and initiate a handshake. The contents of this handshake packet are Bs UUID. A will recognize this as a handshake attempt (via the protocol field), and store Bs UUID in a hashmap mapping addresses to UUIDs. For reasons specified in the next section, A will also immediately send B a handshake packet containing its own UUID.

When A sends a message to B, part of the packet header for packet P is a UUID which represents the UUID that node A thinks node B has. So, our modified RIO Packet looks like this:

```
---HEADER----
Senders stored UUID for destination address
Protocol
---PAYLOAD---
Contents
-------------
```

If B receives a UUID that is not its current UUID the following occurs:

B drops any packets it currently has from A.

The reason we drop all packets is to keep at-most-once semantics. Even if some of the packets in the RIO queue have the correct UUID, from Bs point-of-view its unclear whether P arrived late, or whether a node (either A or B) crashed and a new handshake needs to occur. In the event that A crashed, storing old packets may result in B waiting

for a packet (in order to keep the packets in-order) that may never come. Rather than attempt to determine which situation occurred, which may result in some actions being executed twice, we decided to liberally apply at-most-once semantics in this case and just drop all packets in queue.

B initiates a new handshake.

B will immediately send A a handshake packet, containing its actual UUID in the contents of the packet. If P was simply arriving late, A will have Bs correct UUID and so A will proceed as normal. If A or B crashed, then a handshake will initiate and both nodes will have each others correct UUID afterward.

A will drop any packets its attempting to resend currently.

Knowing that a fault may have occurred, A will drop all packets its attempting to resend (packets that have not been ACKed). It does this because it knows that B is dropping all packets in its queue, meaning that the sequence number on the resent packet will not be correct (i.e. B will be waiting for packet number 0 immediately after this occurs, whereas the resent packet may have a sequence number of 10, resulting in commands being executed out-of-order).

Assumptions for testing our implementation:

Due to our handshaking mechanism, we expect that before any commands are sent from node A to node B, A will initiate a handshake like so:

```
A noop B
```

If this step is not done, then at the very least the first command from A to B will be ignored - possibly more depending on whether or not A waits for Bs response.

We assume that after initiating a handshake via a NOOP, whoever initiated the handshake (likely the client) will wait for the response (the reciprocal handshake) to arrive before sending any other commands. If the sender does not, then the recipient will likely ignore incoming packets until the client begins sending packets with the correct UUID attached.

How To:

Our included test scripts can be run with:

```
./execute.pl -s -f 0 -n={Client,PerfectClient}
             -c scripts/{HandshakeOneWay, HandshakeTwoWays, FSTestOps, FSTestErrors}
```

The PerfectClient is identical to the Client except its failure, recovery, drop, and delay rates are all 0. It is useful mostly to verify correct behavior in the simplest cases.

In the Handshake scripts, nodes communicate their UUIDs and then try to create a file. If packets are delayed or dropped, the create command(s) may be ignored.

The FSTest scripts test some simple operations and errors. Timeouts are best tested manually by raising the failure level of the simulator via -f.

Logging:

We added considerably more output through the use of the Clients printVerbose method. All commands print a message after executing. Messages are also printed after command responses are sent and when packets are presented to the Client from the RIOLayer. Message are printed to stderr when errors occur. Error codes are described in the ErrorCode class.

We also added a command DEBUG that can be entered in the simulator. It currently outputs the given nodes sequence number data (used for debugging handshakes).

Messages are also printed when packets are resent by the OutChannel.

The ReliableInOrderMsgLayer prints a message when it ACKs a packet, receives a packet from the simulator, and when it maps a node address to a UUID after recieving a Handshake.

Outstanding issues:

It would be nice if a delayed packet with a bad UUID did not cause a total reset of both client and server, but were hesitant to call this an outstanding issue since it does implement at-most-once semantics, though admittedly at the cost of performance and reliability (i.e. the client can not rely that the server has actually done what the client sent it in a previous lifetime). Whether this is an outstanding issue is something we leave up to you, but we thought we should mention it anyway.

We did try resending packets using timeouts with corrected UUIDs, but the sequence numbers got out of sync in some tricky edge cases, and simply resetting the connections ended up seeming cleaner.

For this assignment, we havent really completely implemented RPC on top of the onCommand function, which will likely be useful in future assignments. It turned out to be easier to test the command sending functionality by just using the onCommand function since we were testing directly from the simulator - not calling file system operations from code. In the future, we will probably migrate these commands to separate methods, although were unsure how to make these methods block until a response is received without adding a thread and some thread-safe data structures.

Several TODOs remain in the codebase at this point - most are notes and ideas about generalizing our logging mechanism and do not affect functionality.