CSE 490H Assignment 1

Colin Scott and Bill Cauchois

February 11, 2011

Our implementation essentially consists of two parts: modifications to the RIO message layer, and the addition of an RPC layer.[1]

1 Modifications to the Reliable In-Order Message Layer

1.1 Overview

We made several modifications to the RIO layer to ensure its semi-reliable operation in the presence of node failures. The basis of our fault-tolerance mechanism is the concept of an *instance ID*, a number associated with an `RIONode` that uniquely identifies an incarnation of that node. By this we mean that when a node crashes, its new instance ID is guaranteed to be distinct from any of its previous instance IDs. However, instance IDs do not need to be globally unique – it is perfectly plausible for two nodes to have the same instance ID. By including a *source instance ID* and *dest instance ID* with every packet, we are able to determine which incarnation of a client that packet was sent from, or which incarnation of a server that packet was sent to.

Whereas the previous RIO protocol had only two types of packets (DATA and ACK), we have five types: DATA, ACK, SYN, SYN-ACK, and RST. These packets roughly correspond to their TCP equivalents – SYN and SYN-ACK are used in a handshake, while RST is used to reset a connection.

Note that all connections in our system (and in the original RIO system) are unidirectional – that is, we only think in terms of one-way communica-

[1] I like toast. Vincent, do you like toast? If you haven't tried it, you should, because toast is really good. Especially with butter. Buttered toast, buttered toast.

tions. In order to establish a bidirectional channel, it is necessary to establish two unidirectional channels (which means two handshakes, etc.).

1.2 Initiating the Connection and Sending Packets

Each connection in our system begins with a two-part handshake. The handshake is necessary because although each node knows its own instance ID, it does not know the instance ID of the node it wishes to establish a communications channel with. This is the job of our handshake. Later we will explain how knowing the instance IDs on either end of the connection can help us determine if a failure (of either end) has occurred. The handshake begins when a node (that we'll call the client) tries to send a packet to another node (that we'll call the server). This data packet is buffered while the handshake occurs. The handshake consists of the client sending a SYN packet to the server containing its instance ID, and the server sending an ACK-SYN back to the client containing *its* instance ID. The SYN packet is sent using the same mechanism used to send RIO packets – that is, if no ACK-SYN is received after a time interval has elapsed, another SYN is sent to the server. Once the handshake has completed and both the client and the server know of each other's instance IDs, communication can begin in earnest.

Sending data packets after the handshake is largely unchanged from the way it was handled before. Packets are sent and, if no ACK is received after a time interval has elapsed, re-sent. The server, upon receiving a packet, immediately sends an ACK back to the client. The only difference is that each of these packets contains a *source instance ID* and a *dest instance ID*.

1.3 Fault Detection, Notification, and Recovery

After the handshake, the client stores (in its OutChannel) what *it thinks the instance ID of the server should be* while the server stores (in its InChannel) what *it thinks the instance ID of the client should be*. By comparing the instance IDs on the packets they receive to these stored values and their own instance IDs, the client and server should be able to tell when the other one has crashed.

For example, say the client performs the handshake and then sends n data packets to the server. However, the server crashes before it can receive and ACK the nth packet – and then restarts a short while later. Thus, the

client, upon receiving no ACK, attempts to send that packet again. In the old RIO system, the server would get that packet and then buffer it, since it has sequence number n and its expecting a packet with sequence number 0 (that will never arrive). In our new system, the server upon restarting will increment its instance ID. So say the client now has an instance ID of 0 and the server has an instance ID of 1. The packet that the client is resending has sourceID=0 (the current ID of the client) and destID=0 (what the client thinks the ID of the server is). The server, upon receiving this packet, sees that the destID does not match its current instance ID and knows that it was meant for a previous incarnation. Now the client must be brought up to speed (and informed that the server has crashed). This is the purpose of the RST packet. Upon receiving an RST packet, a client is forced to forget everything it knows about the corresponding connection and re-initiate a handshake if it wants to send more data. So, the server receives a data packet from the client and recognizes an incorrect destID, then sends an RST packet to the client. Finally, the client RIO layer notifies application code that a peer failure has occurred by invoking `Node.onPeerFailure()` (with a list of packets that remain unsent).

Let's consider another illustrative example. Say the client performs the handshake and then sends a data packet to the server – but this data packet is delayed. Say the client then crashes and performs another handshake, perhaps sends some data to the server, and *then* the data packet from the previous incarnation of the client arrives at the server. In the old RIO system, this packet would greatly confuse the server – it might take the place of a legitimate packet, or it might cause the server to send back an erroneous ACK that further corrupts state on the client. The correct behavior, of course, is to ignore it. In our new system the client will have incremented its instance ID upon restarting. Therefore the delayed packet has an sourceID of, say, 0. When the client performs a second handshake with the server after restarting, its new instance ID will be propagated to the server. So the server, upon receiving the delayed packet with sourceID=0, knows that the current instance ID of the client is 1 and the packet should be ignored.

These are only a few of the scenarios we have thought through in-depth when designing our protocol. We believe it to be fairly robust in the face of faults – ensuring that no packet is delivered more than once, and no packets are allowed to corrupt the state of either the client or the server.

It is worthwhile to consider the cases in which a node may be notified of a peer failure:

1. After the server restarts, when the client sends a data packet with an incorrect destID, the server will send an RST packet back to the client – who will then notify its Node of a failure.

2. After the client restarts, it may attempt to initiate a new connection with a server with whom it had previously communicated. This server will already have a corresponding InChannel object, however the sourceID on the InChannel won't match the sourceID on the SYN packet. In this way, the server knows that the client has crashed and will notify its Node of a failure.

3. If the client does not get an ACK for a data packet after several tries, it assumes the server has gone down. It forgets everything it knows about the server and notifies its Node of a failure (this is the timeout case).

2 Description of the RPC Protocol

To make the implementation of RPCs modular, we have a single class RPCNode delegate calls to RPCClient and RPCServer objects. When issuing an RPC, RPCClients first encode the RPC function and arguments and use the RIOMessageLayer to reliably transfer the call to the specified server. Upon receipt of an RPC, RPCServers decode the function, perform the RPC and again use the RIOMessageLayer to send a result packet and error code back to the client. At-most-once semantics are guaranteed by the RIOMessageLayer; in the case of a server failure or RPC timeout, the RIOMessageLayer calls a special function `RPCNode.onPeerFailure(unsuccesfulCalls)` which notifies the client and server that the outstanding `unsuccesfulCalls` may or may not have been executed by the server.

The underlying RPCNode is notified of the failure or success of an RPC call when `onRPCSuccess()` or `onRPCFailure()` is invoked by RPCClient.

We encapsulate all RPCs into RPCPackets. This allows us to allows us to easily distinguish call packets from return packets, as well as indicate error codes for return packets. Our encoding scheme for the remainder of the RPC data is pretty kludgy: we simply insert the function call, filename and any additional parameters delimited by spaces into the payload. In the future, we would like to use Protocol Buffers to cleanly encode our RPC packets.

3 Assumptions

One major assumption that we make is that all nodes are cooperating – that is, none of them are malicious. We noticed at many points during the design process that a malicious node could easily corrupt the state of our system. Hopefully, we won't have to deal with security issues within the scope of this project.

4 Implementation Quirks, Other Discussion

We modified Protocol.java and RIOPacket.java fairly heavily so that the format of the packet headers is similar to TCP/IP in that ACK, SYN, etc. are encoded as flags while the protocol field is reserved for use by the application layer.

We think that bidirectional connections in the RIO layer would be preferable to unidirectional connections. This is because our protocol can only detect failures across a single (unidirectional) connection, which leads to some pretty odd cases. For example, suppose a client performs a handshake with a server, sends an RPC call across the wire, and then crashes immediately after the server receives the message. After the server executes the RPC, it will need to perform a handshake with the client to send the results back. If the client reboots in time, the handshake will complete successfully even thought the current incarnation of the client never issued the RPC... this sort of case could be eliminated if the connections were bi-directional.

We tested our code fairly extensively using the interactive simulator. Ideally we would have automated tests, but we found it too difficult to automate the testing process with the provided framework.

5 How to Use

To use our implementation, simply run `./execute.pl -s -n RPCNode`. Then, start as many nodes as you'd like and issue commands between them using the syntax specified in the assignment description. Feel free to fail some nodes as you see fit, our system can take it!