HadoopRpc

Hadoop uses a general-purpose RPC mechanism. The main idea is, define a single interface, shared by the server and the client. The client will use the `java.reflection` proxy class pattern, to generate an implementation to the RPC interface. See Java theory and practice: Decorating with dynamic proxies fro more details.

When the client calls a method, say, `String ping(String msg)`, in the proxy class, it would:

1. Serialize the arguments for the method (`msg` in our example).

2. Connect to the RPC server. 3. Tell the server to execute the method with its arguments (in our example, we'll tell the server to execute method `ping`, with a single argument `msg`

The server would:

1. Deserialize given arguments.

2. Execute requested method with those arguments (in our example it'll deserialize `msg` and run `ping(msg)`). 3. Serialize the return value. 4. Send the return value back to the client.

Here is an example of a very simple Hadoop RPC call.

Shared interface:

```
@ProtocolInfo(protocolName = "ping", protocolVersion = 1)
interface PingRPC {
    String ping();
}
```

Note that there's an extra detail I neglected to mention. For backwards compatibility purposes. The RPC interface needs to be versioned, to make sure it's backwards compatible.

For example we might update the `PingRPC` interface to be:

```
@ProtocolInfo(protocolName = "ping", protocolVersion = 2)
interface PingRPC {
    String ping();
    String enterprisePing(String companyName, String referee,
        int priority, long timeOut, boolean useTimeout,
        boolean useUseTimeout);
}
```

Old clients will still be able to call `ping`.

The server would do:

```
class PingRPCImpl implements PingRPC {
    public String ping(String msg) { return "pong=" + msg; }
}
public static void main(String[] args) {
    final RPC.Server server = new RPC.Builder(new Configuration()).
                setInstance(new PingRPCImpl()).
                setProtocol(PingProtocol.class).
                build();
    server.start();
}
```

At last the client would run

```
public static void main(String[] args) {
    PingRPC ping = RPC.getProxy(PingProtocol.class,
        RPC.getProtocolVersion(PingProtocol.class),
         server.getListenerAddress(), new Configuration());
    System.out.println("Server ping returned " + ping.ping("X"));
    // output: pong=X
}
```

Note that currently, a production Hadoop RPC interface, must extend and implement the `VersionedProtocol` interface. It's used by `ProtocolProxy.fetchServerMethods` to make sure client and server protocol version match. We neglected this detail for the sake of simplicity.

We understand now how to implement a basic Hadoop RPC interface, and even how to add methods to it, maintaining backwards compatibility. But what happens under the hood? What goes through the wire when the proxy issues `ping`?

The logic behind the RPC mechanism is not very easy to follow. There are two main tasks one needs to do when issuing an RPC call. Serializing and deserializing sent Java Objects, and sending them on the wire with some protocol.

The wire protocol is defined at the `org.apache.hadoop.ipc.Client` and `org.apache.hadoop.ipc.Server` classes. A client call starts its life with the `Client.call` method. There we would create a new connection (or pull one from the connection pool) and begin an RPC handshake.

The RPC handshake starts with a header:

```
+----------------------------------+
|  "hrpc" 4 bytes                  |
+----------------------------------+
|  Version (1 byte)                |
+----------------------------------+
|  Service Class (1 byte)          |
+----------------------------------+
|  AuthProtocol (1 byte)           |
+----------------------------------+
```

Version is probably the RPC protocol version, and is currently 9. ServiceClass defaults to 0. The AuthProtocol determines whether or not to use SASL authentication before starting actual RPC calls. SASL will be used if `hadoop.security.authentication` is set to non simple authentication, or that the client explicitly gave an authentication token when creating the proxy .

Note that this header is different between Hadoop 2.0.x-alpha and 2.1.0-beta. Hence, as you can expect, clients with Hadoop 2.0.x jars cannot connect to Hadoop 2.1.0 servers:
hadoop-2.0.1-code, https://github.com/apache/hadoop-common/commit/fbbe30fcce227ea79365a003ef97684166cd9d24#hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/ipc/Client.javHADOOP-9630.

Next, we'll set the connection context. The connection context is a protocol buffer message `IpcConnectionContextProto`, generated by `makeIpcConnectionContext`. The context object defines the destination RPC protocol name (we defined it as "ping", using Java annotations in the example above), and the user who calls the protocol. We'll tell more about this object when we'll speak about RPC security. Note that unlike other protocol buffers objects, the context is delimited by 4 bytes ints, and not by varint.

```
+-----------------+
|  4 bytes length |
+-----------------+
| IpcConnection   |
| ContextProto    |
+-----------------+
```

After the header was sent, the client will start a SASL authentication if necessary. I'll cover authentication in a dedicated post.

OK then. The initial handshake is done, and we're ready to send the first request to the server.

The wire format of a single RPC request is defined in org.apache.hadoop.ipc.Client:
Format of a call on the wire:

1. Length of rest below (1 + 2) 2. RpcRequestHeader - is serialized Delimited hence contains length 3. RpcRequest 4. The payload header, `RpcRequestHeaderProto`, is a google Protocol Buffers object, hence serializable by Protocol Buffers. The main content of this header are `callId`, which is an identifier of the RPC request in the connection, and `clientId`, which is a UUID generated for the client. Negative {{callId}}s, are reserved for meta-RPC purposes, e.g., to handshake SASL method.

The payload itself is a serialized representation of a method call. Serializing and deserializing the objects is done by an `RpcEngine`, the default one is `WritableRpcEngine`, which uses the native Hadoop serialization to serialize Java objects `Writable`. Protocol Buffers based engine can be selected by setting `rpc.engine.<protocol_name>=ProtobufRpcEngine`, but we'll concentrate on the default implementation.

Naturally, `WritableRpcEngine` writes the payload as a `Writable` object, whose serialization is defined by the `Invocation` object write method, WritableRpcEngine.Invocation.write:
The weird mix between Protocol Buffers and Writables, is probably due to on going migration to Protocol Buffers based RPC.

The RPC server sends a similar response to send the return value:

```
int - Message Length
Varint - length of RpcResponseHeaderProto
protocol buffers object - RpcResponseHeaderProto
Writable response value (length known by intial length)
```

The `RpcResponseHeaderProto`, contains the `callId`, used to identify the client, an indicator whether or not the call was successful, and in case it wasn't, information about the error.

The last detail of the RPC protocol is the ping message. If the RPC client times out waiting for input, the client will send it 0xFF_FF_FF_FF, probably to keep the connection alive. The server knows to ignore such headers. This can be disabled by setting `ipc.client.ping=false`, and is controlled by the `PingInputStream` class.

Let's have a look at a real example. Let's take the simple ping RPC server in our example git repository hadoop_rpc_walktrhough.

First, we'll set a fake server that records client traffic, and then we'll clone and run the simple RPC client.

```
$ nc -l 5121 >clientRpcCall &
[1] 25269
$ git clone https://github.com/elazarl/hadoop_rpc_walktrhough.git
Cloning into 'hadoop_rpc_walktrhough'...
remote: Counting objects: 41, done.
remote: Compressing objects: 100% (27/27), done.
remote: Total 41 (delta 4), reused 36 (delta 4)
Unpacking objects: 100% (41/41), done.
$ mvn -q exec:java -Dexec.mainClass=com.github.elazar.hadoop.examples.HadoopRPC -Dexec.args="client"
2013-08-18 06:32:06 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using
builtin-java classes where applicable
```

Now, let's look at the annotated hexdump of the output: hadoop_rpc_walktrhough

```
//h   r     p     c
0x68, 0x72, 0x70, 0x63,
// version, service class, AuthProtocol
0x09, 0x00, 0x00,
// size of next two size delimited protobuf objets:
// RpcRequestHeader and IpcConnectionContext
0x00, 0x00, 0x00, 0x32, // = 50
// varint encoding of RpcRequestHeader length
0x1e, // = 30
0x08, 0x02, 0x10, 0x00, 0x18, 0xfd, 0xff, 0xff, 0xff, 0x0f,
0x22, 0x10, 0x87, 0xeb, 0x86, 0xd4, 0x9c, 0x95, 0x4c, 0x15,
0x8a, 0xb0, 0xd7, 0xbc, 0x2e, 0xca, 0xca, 0x37, 0x28, 0x01,
// varint encoding of IpcConnectionContext length
0x12, // = 18
0x12, 0x0a, 0x0a, 0x08, 0x65, 0x6c, 0x65, 0x69, 0x62, 0x6f,
0x76, 0x69, 0x1a, 0x04, 0x70, 0x69, 0x6e, 0x67,
```

This is the standard header sent before any RPC call is made. Now starts a stream of RPC messages

```
// Size of size delimited
// RpcRequestHeader + RpcRequest protobuf objects
0x00, 0x00, 0x00, 0x3f, // = 63
// varint size of RpcRequest Header
0x1a, // = 26
0x08, 0x01, 0x10, 0x00, 0x18, 0x00, 0x22, 0x10, 0x87, 0xeb,
0x86, 0xd4, 0x9c, 0x95, 0x4c, 0x15, 0x8a, 0xb0, 0xd7, 0xbc,
0x2e, 0xca, 0xca, 0x37, 0x28, 0x00,
// RPC Request writable. It's not size delimited
// long - RPC version
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, // = 2
// utf8 string - protocol name
0x00, 0x04, // string legnth = 4
// p     i     n     g
0x70, 0x69, 0x6e, 0x67,
// utf8 string - method name
0x00, 0x04, // string legnth = 4
// p     i     n     g
0x70, 0x69, 0x6e, 0x67,
// long - client version
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, // = 1
// int - client method hash
0xa0, 0xbd, 0x17, 0xcc,
// int - parameter class length
0x00, 0x00, 0x00, 0x00,
```

Since the `nc -l 5121` server does not respond, the client will eventually send a ping message.

```
// ping request
0xff, 0xff, 0xff, 0xff,
```

How would we get the server message? Let's use the client request file we've recorded earlier, and feed it to `nc` as a Hadoop RPC client demiculu. Note we need to wait for input, hence sleep before closing `nc`'s input stream.

```
$ (cat client.dat;sleep 5) | nc localhost 5121 |hexdump -C
00000000  00 00 00 33 1a 08 00 10  00 18 09 3a 10 9b 19 9b  |...3.......:....|
00000010  41 4d 86 42 d7 94 79 3f  4b 16 a0 22 7c 40 00 00  |AM.B..y?K.."|@..|
00000020  10 6a 61 76 61 2e 6c 61  6e 67 2e 53 74 72 69 6e  |.java.lang.Strin|
00000030  67 00 04 70 6f 6e 67                              |g..pong|
00000037
```

adding some annotation will make that more clear:

```
// size of entire request
0x00, 0x00, 0x00, 0x33,
// varint size of RpcResponseHeader
0x1a, // 16 + 10 = 26
0x08, 0x00, 0x10, 0x00, 0x18, 0x09, 0x3a, 0x10, 0x9b, 0x19,
0x9b, 0x41, 0x4d, 0x86, 0x42, 0xd7, 0x94, 0x79, 0x3f, 0x4b,
0x16, 0xa0, 0x22, 0x7c, 0x40, 0x00,
// Writable response
// short - length of declared class
0x00, 0x10,
// j     a     v     a     .     l     a     n     g     .
0x6a, 0x61, 0x76, 0x61, 0x2e, 0x6c, 0x61, 0x6e, 0x67, 0x2e,
// S     t     r     i     n     g
0x53, 0x74, 0x72, 0x69, 0x6e, 0x67,
// short - length of value
0x00, 0x04,
// p     o     n     g
0x70, 0x6f, 0x6e, 0x67,
```

This are the essential details of a Hadoop RPC server. You can see example go code that parses Hadoop RPC on the main_test.go test that walks through an example static binary output, or a very simple ping rpc client.