COMP 360 — Programming Task 1

Jeff Epstein

Due 29 Sep 2016

1 Introduction

We're going to build a (simple) distributed file system. This project will be broken into several smaller tasks, each of which will be an assignment. Because each task builds on the preceding tasks, it's important that each task be satisfactorily completed before moving on. In case of any doubt or in need of help, please contact me.

In the first part of the project, we're going to build an remote procedure call (RPC) system. There are two important roles in an RPC system: a *client*, which is responsible for invoking calls; and a *server*, which is responsible for responding to them.

To prepare for this project, please read section 4.2 in the textbook.

RPC An RPC is an invocation of a function across a network. Like a traditional function call, an RPC is identified by the name of the function to be called; unlike a traditional functional call, the endpoint (host and port) of the RPC server must also be taken into account. An RPC may accept parameters, and may return a value.

Typically, the sequence of events in an RPC looks like this:

1. The client establishes a network connection with the server.
2. The client send (in serialized form) a request, consisting of:
   - The name of the remote function to invoke.
   - Any parameters needed by that function.
   - A unique request ID.
3. The server receives the request and deserializes it.
4. The server executes the requested function.
5. The server sends (in serialized form) a response to the client, consisting of:
   - A status code, indicating success or failure of the operation.

- The return value of the function, if any.
- The same unique request ID.

6. The client receives the response from the server, deserializes it, and closes the connection.

You will implement the roles of client and server as separate programs.

2 Rules

Operating system I will evaluate your program submission by running it on a computer running the Debian Linux operating system. Your grade will therefore reflect the behavior of your work in a Linux environment.

While you are welcome to develop your projects under any operating system you like (such as Windows or OS X), I urge you to test it in an environment similar to mine, in order to assure yourself of getting the highest possible grade. Even if Linux is not your primary operating system, you can run it in a virtual machine under your primary operating system.

Network A distributed system, by definition, runs on multiple computers. Although we are building distributed systems, I assume that most students do not, in fact, have easy access to multiple computers. Therefore, I propose two possible approaches for developing a distributed system without distributed hardware:

1. **Virtual network** To simulate multiple computers, you may use multiple virtual machines (running under a virtual machine manager such as VirtualBox), configured to use a virtual network. Instructions for setting this up at available on Moodle. Using virtual machines to simulate computers in a distributed system is the preferred option, as it results in the most "authentic" network environment. However, virtual machines requires a lot of RAM, and if your personal computer doesn't have a lot of RAM, this option may not be feasible.

2. **Loopback interface** Another approach to developing networked applications on a single computer is make use of the Linux *loopback interface*, also known as *localhost*. This mechanism lets programs (such as the client and server in this assignment) communicate over a network interface, even though they are running on the same computer. (The term *loopback* should conjure the image of a network cable connected to a computer, looping back on itself, and then plugged in again with the other end into the same computer.) Opening a connection to the special address `localhost` will create a connection such that the destination address is on the same computer as the source address, although the endpoints may have different ports.

Whatever solution you choose to test your program (real network, virtual network, or loopback interface), the design and implementation of your program should be the same. That is, your program should be agnostic as to the network device that it is using to communicate: your program simply accepts an address as a parameter, and opens a connection to that address, leaving the question of how to resolve that address to the network stack.

Language You should use whichever programming language you feel most comfortable with. However, if you want to use a language other than Python or Java, please talk with me first.

If you choose to use Python, I recommend Python 3.

Academic honesty You should write this assignment entirely on your own, without consulting code from any other source, including from other students.

Code quality You should adhere to the conventions of quality code:

- Indentation and spacing should be both consistent and appropriate, in order to enhance readability.
- Names of variables and functions should be descriptive.
- All functions should have a comment in the appropriate style describing its purpose, behavior, inputs, and outputs. In addition, where appropriate, code should be commented to describe its purpose and method of operation.

If I think that the quality of your code (independent of the correctness of your program) is not of professional quality, I may ask you to make corrections and resubmit your work before I will give you a grade.

3 Assignment

You're going you to write two programs, `client` and `server`.

Server The server, once started, will sit in an endless loop, servicing RPC requests from the client.

We can forcibly terminate the server by pressing Ctrl-C. It will try to bind TCP port 38000; if, for some reason, it cannot bind to that port (for example, because another program is already listening on that port), it will try port 38001, and so on, until it finds an available port, or until it fails to bind port 38010, at which point it will give up and emit an error message. If it successfully binds a port, the program will then, in an endless cycle, accept RPC requests (see the detailed description of supported functions). For each request, it will determine if the request is valid, that is, if the requested function exists and if the correct parameters are provided in the request. If so, it will perform the

designated operation, and send the result back to the client. If the request is invalid, the program will send an error message to the client.

I recommend that your server print to the terminal helpful status messages indicating what it is doing, such as "Accepting connection from host 1.2.3.4" or "Rejecting RPC request because function is unknown."

Client The client is used to connect to a running server and transmit RPC requests.

The client is intended to be run from a command line. It must support an optional command-line argument, `--server`, for specifying the server host to connect to. It will also support a mandatory command-line argument specifying the RPC to be executed (see the detailed description of supported functions). If no server is specified, it should by default use `localhost` (that is, the machine it is being run on). It should attempt to establish a connection to the given server on port 38000; if, for some reason, it cannot connect to that port, it will try subsequent ports, up to 38010, at which point it will give up. It will transmit an RPC request, and will wait for a response. Upon receiving a response, it will display the status of the RPC to standard out.

RPCs Your programs should support the following RPCs:

- `print [text]` – The `print` RPC takes an arbitrary string as a parameter. Upon its execution, the server will simply display the string on the terminal.
- `set [key] [value]` – The `set` RPC takes two strings: a key and a value. Upon its execution, the server will set a locally-stored value named *key* to the value *value*. If such a named value already exists, it will be replaced.
- `get [key]` – The `get` RPC takes as a parameter a string identifying a key. It will find the value associated with the given key by a previous call to `set`, and return it to the client. If there is no such value, the RPC will return an error.
- `query_all_keys` – This RPC returns a list of all keys (but not values) that have been set by the `set` RPC.

Your program should support command-line syntax so that these RPCs can be specified as parameters to your client program. Here's an example run. First we start the server, and it waits for incoming connections from a client:

```
eppie@tenger:~/Projects/wesleyan/distsys/phase1$ ./server.py
Trying to listen on 38000...
```

Then we run the client, specifying (a) the address of the server and (b) the RPC to invoke, along with any RPC-specific parameters (in this case we want to print the string `hello`):

```
eppie@tenger:~/Projects/wesleyan/distsys/phase1$ ./client.py --server localhost print hello
Trying to connect to localhost:38000...
{u'status': u'ok'}
eppie@tenger:~/Projects/wesleyan/distsys/phase1$
```

The server receives the client request and performs the action, then goes on waiting for the next client:

```
eppie@tenger:~/Projects/wesleyan/distsys/phase1$ ./server.py
Trying to listen on 38000...
Printing hello
```

4 Hints

- A brief reminder on the magic incantations necessary for creating network socket connections. Here we use Python. (If you are using Java, you will want to read about the `java.net.Socket` and `java.net.ServerSocket` classes, which work similarly to the Python code below.)

  A socket will be created either for incoming connections or for outgoing. In either case, the port number of the listening host must be specified.

```
import socket

# Create an outgoing connection to somehost.com on port 40000
# Give up after five seconds
host = "somehost.com"
port = 40000
timeout = 5

# Try to connect, or throws an exception
sock = socket.create_connection((host, port), timeout)
if sock is None:
    print("Can't connect")
```

  Creating a listening socket is a little more complicated:

```
import socket

# Listen on port 40000 on all interfaces
# Give up after five seconds
port = 40000
timeout = 5
bound_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Note these calls may throw an exception if it fails
bound_socket.bind(('', port))
bound_socket.listen(1)
```

```
if timeout:
    bound_socket.settimeout(timeout)

# Accept connections forever
# Returns the socket, address and port of the connection
while True:
    sock, (addr, accepted_port) = bound_socket.accept()
```

Once a socket is created, whether as an outgoing or incoming connection, the following operations can be used on it.

```
# Send a sequence of bytes
# Returns number of bytes sent
msg = "some arbitrary data as a string".encode()
sock.sendall(msg)

# Receive at most msg_length bytes
# Returns value received
msg_length = 4
msg = sock.recv(msg_length, socket.MSG_WAITALL)
if len(msg) == 0:
    # recv gives 0 result if the connection has been closed
    print("Connected terminated")
elif len(msg) != msg_length:
   print("Incomplete message")
else:
    print(msg.decode())

# Close the connection
sock.close()
```

- When sending a data structure as a message over the network, it must first be serialized (i.e. converted to a string). There are several ways you can do this, but I think the easiest will be to use one of the available serialization packages. Any of the following are able to convert (mostly) any arbitrary Python data structure into a string.
  - The `json` and `pickle` modules are in Python's standard library.
  - The `PyYAML` package can be installed.

  Similar tools are available for Java. Here's an example of using the `json` module to convert an arbitrary data structure (in this case, a `dict` containing a variety of values) to a `str`:

  ```
  import json
  some_data = {
      "stuff": 56,
  ```

```
    "more_stuff": [1,6,"number",1,-45],
    "extra_stuff": 3.2,
    "further_stuff": "a value",
   }
encoded = json.dumps(some_data)
```

- When sending a message (for example, the initial RPC request or its response) over a socket, the receiving side has to know how many bytes to read (given as a parameter to the `recv` function). The `recv` function, if it is called with the `MSG_WAITALL` parameter, will wait until it receives the expected number of bytes, so if it expects more than are sent, it will potentially wait indefinitely. (Without `MSG_WAITALL`, `recv` may return fewer bytes than requested, and you will need to call the function again to retrieve the rest.) The usual solution to this is to first send the length of the message as a fixed-size integer, then send the actual data. The receiving side would then first read the length, then read the requisite number of bytes of the actual message. You can use Python's `struct` module to convert a Python `int` to a suitable encoding. For example, in this case we use `struct.pack` to encode an `int` as a 32-bit binary value, which we can then send:

  ```
  # sending side

  import struct
  sock = ... # open the socket here
  msg = "this is the message"
  msg_length = len(msg)
  msg_length_encoded = struct.pack("!i", msg_length)
  sock.sendall(msg_length_encoded) # send the length
  sock.sendall(msg) # send the data
  ```

  And on the other side:

  ```
  # receiving side

  import struct
  sock = ... # open the socket here
  msg_length_encoded = sock.recv(4, socket.MSG_WAITALL) # read the leng
  msg_length, = struct.unpack("!i", msg_length_encoded)
  msg = sock.recv(msg_length, socket.MSG_WAITALL) # read the data
  ```

  Note that we use `sendall` instead of `send` to ensure that the whole message is sent. This approach can have negative consequences if we are sending lots of data (sending can block), but for the purposes of this assignment, it should be fine.

- For parsing the command-line parameters to the client program, I suggest using the Python `argparse` module.

5 Submission

You are obligated to write a `README` file and submit it with your assignment. The `README` should be a plain text file containing the following information:

- Your name
- Instructions for running and testing your program.
- The state of your work. Did you manage to complete the assignment? If not, what is missing? If your assignment is incomplete or has known bugs, I prefer that students let me know, rather than let me discover these deficiencies on my own.
- Any additional thoughts or questions about the assignment. Were the instructions clear? Were you given enough support in developing your solution? Was the assignment interesting?

Please save your client program as `client.py` and your server program as `server.py`. Submit these files, as well as your `README` file any other source code necessary to run your project, on Moodle.