Project 3: RPC

Out: Tuesday April 17
Due: Tuesday May 1 (11:59 PM)
Note: There is a midterm on 4/25
Turnin: Online
Teams: Pairs

Assignment Overview

Communicating over raw sockets is difficult. Sockets transmit bytes, leaving data encoding and message format implementation up to each application programmer. Sockets provide minimal inherent synchronization and protocol, mechanisms, leaving to each application programmer to invent protocols and enforce them.

An enhancement to raw sockets is Remote Procedure Call (RPC). As the name implies, it provides an abstraction as similar as possible to procedure call. The caller simply invokes a procedure on a remote machine. Arguments are sent, with details of data encoding worked out by the RPC implementation. The calling thread is suspended (blocks) until a message from the remote procedure arrives indicating that it has completed, or encountered an error. That message can contain return values (i.e., it can be remote function call).

In this project you'll implement a simple RPC mechanism, and use it to build a simple application (ping).

RPC Protocol Overview

This section mostly talks about the RPC protocotol, but also a bit about its implementation. A more complete overview of the implementation is in next section.

RPCs take place over a TCP connection. There is an initial RPC handshake performed when the TCP connection is established. In theory, that can be followed by one or more remote calls ("persistent connection"); in practice, we allow only a single call and then close the TCP connection ("non-persistent connection").

All messages are JSON encoded `JSONObject`s, sent as `TCPMessageHandler` encoded strings. There is an RPC header, represented by those `JSONObject`fields that exist at the top level. The message payload contains the arguments the caller is sending to the remote callee, or the return value from the callee. The payload is required to be a `JSONObject`.

The Handshake

The initial handshake is communication between the calling RPC service and the remote RPC service. It is not part of application communication, i.e., not application to application.

The handshake begins with the calling RPC service opening a TCP connection to the remote RPC service, and sending a `connect` control message. Connect messages look like this. (This is JSON notation for what is basically a map. Ordering of the fields isn't significant.)

```
{"id":2,"host":"cse461","action":"connect","type":"control"}
```

All of this is what you can think of as RPC header. The `id` field is a unique ID for this message. (While RPC service implementations can simply count up by one to generate these IDs, they are not sequence numbers; the recipient won't necessarily see consecutive IDs.) The `type` field indicates this is a control message, i.e., the intended recipient is the remote RPC service, not some application on the remote machine. The `host` field identifies the sending RPC service.

It's a unique string name, useful in Project 4 but of no further significance in this project. The `action` field indicates what kind of control message this is. (There's only one, but we send this field to allow later extension of the protocol.)

If the remote RPC service is willing to connect, it sends a success response. For our example, the response looks like:

```
{"id":1,"host":"","callid":2,"type":"OK"}
```

The `id` is, again, a unique ID for this message. The name of this remote host is the null string. The `callid` field gives the `id` of the message for which this is a response. The `type` field indicates success.

An error response to the same call looks like this:

```
{"id":1,"host":"","callid":2,"type":"ERROR","msg":"Max connections
exceeded"}
```

The `msg` field is a free form error message, intended to help the caller understand what the problem is. If an error response is received, the caller should close the TCP connection, and not try to send any further RPC requests on it.

RPC Invocation

An invocation message looks like this:

```
{"id":4,"app":"echo","host":"cse461","args":{"msg":"test
message"},"method":"echo","type":"invoke"}
```

The `type` field indicates this is an application-to-application invocation message. The `app` field is the (unique) name of the remote application. The `method` field indicates which of the remote application's methods the caller wants to invoke. The `args` field provides the arguments for that invocation. These are always bundled up as a `JSONObject`. The arguments in this call are a single string, named `msg`, with value `test message`.

A success response looks like this:

```
{"id":3,"host":"","callid":4,"value":{"msg":"test
message"},"type":"OK"}
```

The `value` field is the return value of the invocation. It too must be a `JSONObject`. In this case, the application is `echo`, which just returns anything sent to it, so the return value matches the invocation arguments.

An error response looks like tis:

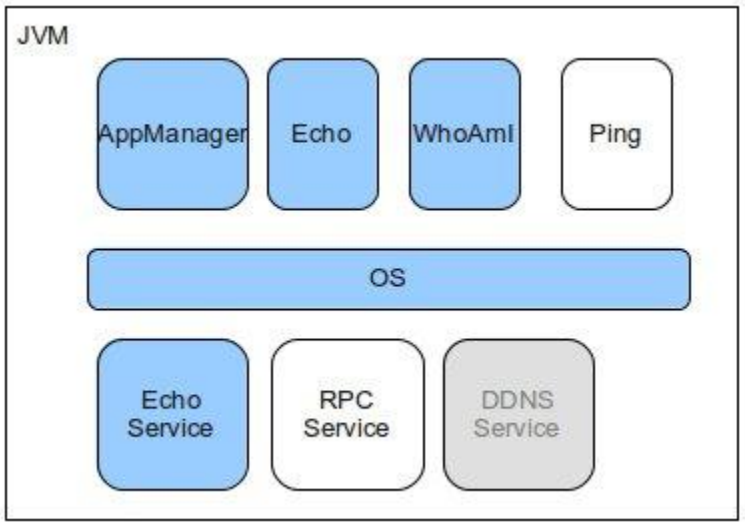
```
{"message":"some error
message","id":3,"host":"","callid":4,"type":"ERROR",
callargs":{"id":6,"app":"echo","host":"cse461","args":{"msg":"test
message"},"method":"echo","type":"invoke"}}
```

The new field here is `callargs`, which simply sends back the invocation arguments. It can be useful in debugging the caller.

System Architecture

The RPC service you'll build fits into a larger architecture. That serves two purposes. First, it's useful for debugging. Some of the code can be viewed as simple test programs. Second, our architecture mimics in many ways that of real systems. That is a natural side-effect of trying to build a flexible system - we'll be using the RPC system in succeeding projects.

The architecture looks like this:



The blue components are completely implemented in the code provided to you. The white components are things you will implement. (Additionally, you'll implement an Android edition of the Ping client.) The grey component, DDNS (Dynamic DNS (Domain Name System)) is part of Project 4, shown here so you'll know where we're headed.

The upper layer components, which we call applications, implement user interfaces. This picture shows the "console applications," those whose interface is textual, and so can run on arbitrary workstations. (The only Android application is Ping.) The `OS` and components below it are "services." They have no UI.

There is no fundamental difference between applications and services, though. Although some implementation details distinguish them, both start at system boot and run until system shutdown. There can be at most one instance of any component type. This is the usual notion of a system service, but not of an application. It is similar in some ways to the Android notion of an application, however.

Both applications and services can use the `RPC` system to make calls, or to expose methods they implement to remote callers. If they have methods callable by `RPC`, they must provide a unique, "well known name" that can be used to identify them.

Component Roles

OS

The `OS` starts services. It does this by dynamically (at run time) loading the required Java class into the JVM and creating an instance of the class. It maintains a data structure mapping the unique name of the service (e.g., "rpc" for

the RPC service) to the Java service object. It provides an interface that takes a service name as an argument and returns a reference to the Java service object.

Additionally, the `OS` maintains a Java `Properties` object that contains parameter values set in a `.ini` configuration file (discussed later), and provides mechanisms for components to access those values.

RPC Service

The `RPC Service` implements the RPC protocol. It has two main components, one supporting calls from the local system to remote systems and the other supporting calls from remote systems to the local one.

RPC is implemented over TCP. To receive remote calls, the `RPC Service` creates a Java `ServerSocket`, binds it to a local port, and executes `accept()` on it.`accept()` returns a new Java `Socket` object when a remote TCP connection is established. The RPC code then engages in its handshake using that socket, and waits for invocation messages. When those arrive, it makes a call to the application (or service) and method named in the invocation message, collects the return value (or error response), and sends that back to caller. That process more or less requires threads. One thread is required to sit in a loop performing `accept()` calls on the`ServerSocket`, since that is a blocking call. Additionally, a new thread should be created to handle each new connection. That makes it easier to keep track of what state the protocol is in than if you try to serve multiple connections from a single thread.

On the outgoing call side, the `RPC Service` exposes a method for making remote invocations. That method opens a TCP connection to a remote RPC`ServerSocket`, engages in the RPC handshake, sends the invocation message, collects the response, and delivers it back to the local caller. Additionally, it must block the calling thread until the response can be returned. The easiest way to do that is to simply use the calling thread to execute all of the code involved in making a remote call.

Echo Service

The `Echo Service` makes a single method available via RPC, `echo()`. It simply returns any arguments it is sent.

AppManager

The `AppManager` performs dynamic loading of the Java classes that implement applications, and creates an instance of each. It then enters a loop asking the user to select an application to run, and invokes the `run()` method of the named application.

Echo

The `Echo` application asks the user to identify a remote system by giving its IP address and the port of the RPC service running there, then to input a line of text. It contacts the remote `Echo Service`, handing it the line of text, and prints whatever value is returned.

WhoAmI

The `WhoAmI` service prints the identity of the system on which it's running. That identity is the IP address of the host and the port to which the RPC `ServerSocket` is bound.

Ping

`Ping` measures (and then reports) how long it takes to make an minimal RPC call to some remote system. By "minimal" we mean a call to the remote `Echo Service`passing an empty argument object. `Ping` should repeat the call five times, reporting the time taken by each test.

(`Ping` presents a natural motivation for implementing persistent connections: rather than closing the TCP connection after an RPC call has completed, instead leave it open in case further calls to the same remote system are issued. With persistent connections, we expect `ping` will report a longer time for the first call than the subsequent ones, since the first includes the RPC handshake but the subsequent ones would not. Persistent connections are by no means

required. Their implementation requires a bit more programming, involving timers and threads, but if you're familiar with those things it shouldn't be too bad.)

RPC Implementation

The RPC implementation consists of two main classes: `RPCService`, which implements the receiving side (accepting calls from remote systems), and`RPCCallerSocket`, which implements sending remote calls. Minimal skeleton code exists, mainly to define the public Java methods they must implement.

The `RPCService` implementation is by far the more complicated piece. Part of what it does is create a Java `ServerSocket` for incoming connections, and implement the RPC protocol over those connections. (There are some details in the RPC Protocol section above.)

Another part of what it does is to accept registration of RPC-callable methods from services (and potentially applications). A service registers a method, and when an RPC comes in for it, a callback to that method occurs. It turns out that implementing this process is a little tricky in Java, so the skeleton code includes a helper class,`RPCCallable.RPCCallableMethod`, that does the hard part. Have a look at the comments in its file for an explanation of what it does, and in`EchoService.java` for an example of how it's used.

Other Implementation Issues

You're writing two applications, `ping` for the console and `ping` for Android. The console-based `ping` should display five ping results. The Android-based one need display only a single ping result (but you can display five as well if you'd like).

Console-based `ping` should be very easy. You can model it on the `Echo` console application. Android-based `ping` is straightforward as well, except that you have to do some potentially new things: create a new Android project in Eclipse, then deal with Android. You can use the Android app from Projects 1/2 as example code, and there are lots more examples at developer.android.com. Your Android `ping` will also want to borrow code from `OS.main()` to load and initialize the various system components.

One issue that would be easy to overlook, and whose symtoms end up being a little confusing, is that an Android application must declare any system permissions it requires in its `AndroidManifest.xml` file. Your application will need at least `INTERNET` permission. If you omit a permission you require, some call will fail, but probably won't hint that the problem is you don't have the required permission in the manifest file.

Note that the usual Java code structuring scheme of creating a `jar` file and using it as a library may not work as you expect. Source compiled as an Eclipse Java project may not run on Android, and vice versa. You therefore need to either create two, system-specific `jars`, or else take the approach already incorporated in the distributed Eclipse project. That approach is to structure the code into natural, library-like Eclipse projects, but to have each project using a "library" include the library's source using Eclipse folder links, so that it's compiled for the appropriate target system.

You're also implementing RPC, which is a core service in a somewhat large body of code. It makes calls into other components (the `OS`), and other components make calls into it. This inevitably brings up the issue of what code you can modify.

I'm hoping you don't find a reason to want to modify existing code, beyond RPC itself. I have an existence proof that an implementation exists that does not require modifying it, but the main problem with working with existing code is that it can be harder than just writing your own. All the code from this project will be used in the next one(s), so the more you change interfaces the harder it will be to integrate new code distributions. I've worried about this issue since beginning work on the project, so I hope the existing code doesn't get in your way.

There's one implementation issue I'm not confident my code can be totally resilient to: exceptions. In general, existing code expects RPC methods to throw some kind of exception when anything goes wrong. Depending on what libraries you use, though, your RPC implementation may throw exception classes I didn't encounter. To try to deal with this, all existing code that calls RPC methods anticipates that an `Exception` might be thrown. I don't know Java well enough to know if this handles everything possible, though. In the "tricky bit" of the RPC callbacks I implemented, something came back that wasn't an `Exception`, so I know there were things I didn't know, and I don't know there aren't more.

Finally, you can't connect to machines running behind NATs from machines not also behind the NAT. If you can't connect to your home machine from the UW, you won't be able to connect to your Project 3 application running on your home machine from the UW. The other direction should be fine, though: an instance on your home machine should be able to connect to an instance running on a CSE machine.

Additional Android Information

There is a separate page elaborating on Android implementation issues.

Config Files

There are a number of system settings that you might like to set on a per-invocation basis. These are placed in a configuration file. The infrastructure reads the configuration on startup. It's values are available at runtime through various `OS` methods. Among other things, the config file lets you specify the port that the RPC's`ServerSocket` should be bound to. (Remember that only one instance of a `ServerSocket` can be bound to any given port, system wide.)

Here's an example config file. It's contains some information relevant to Project 4, but almost superfluous in Project 3. In particular, it has the notion of a "host name." For now, host names are just strings. In Project 4, they will have structure like DNS names (.e.g., `www.cs.washington.edu`). This file is for the host named`foo.bar`. (Peculiarly, the name '`foo.bar`' is actually a short version of the full name, '`foo.bar.`'.)

```
# This file contains device-specific configuration information
# that is read by the infrastructure at launch.

#-----------------------------------------------
# host config values
#   host.name must be a full name (ends with a .)
#-----------------------------------------------
host.name=foo.bar.

#-----------------------------------------------
# ddns config
#-----------------------------------------------
ddns.rootserver=localhost
ddns.rootport=46120

ddns.cachettl=60

#-----------------------------------------------
# rpc config
#-----------------------------------------------
rpc.timeout=10000
rpc.serverport=
```

```
#-----------------------------------------------
# debug config
#   Levels: v:2  d:3  i:4  w:5  e:6
#-----------------------------------------------

debug.enable=1
debug.level=4
```

The `ddns` section is totally irrelevant in this project. The `rpc.timeout` is intended to let you specify how long your code should wait for something to happen on a socket before giving up. (You have to implement code to use this value.) The `rpc.serverport` value lets you tell the code to what port number you'd like to bind the RPC `ServerSocket`. The null value shown here means "any port". (Again, you implement the use of these settings.)

The `debug` settings are used to control the behavior of `android.util.Log`, which prints debug logging messages. An implementation is distributed with the skeleton code so you can write debug log statements that work both when running in console mode, on any system, and on Android. You can turn debug printing off entirely by setting `debug.enable` to 0. You can affect the amount of logging that is printed by (a) choosing a log level at which to print a specific message (e..g, calling Log.v() vs. Log.i()), and then setting `debug.level` to the minimum level for which you want messages printed. See the Android documentation for more information on log levels.

The Source and Running It

Javadoc for the source is here, as well as in the `doc` folder of the distribution. Additional non-Javadoc comments are in the source.

The source is at `/cse/courses/cse461/12sp/461Project3.jar`. Unjar'ing it produces a `461Project3` directory that is an Eclipse workspace. Open Eclipse, and if it opens an old workspace, choose Switch Workspace from the File menu and navigate to the new workspace. Once Eclipse re-opens, if the Package Explorer pane is empty, right-click on it, choose Import..., then General, then Existing Projects into Workspace, Next, browse to the new workspace folder, and click Finish.

You're very likely to have to fix some broken absolute paths Eclipse keeps to external jars, following the procedure described in the Project 1 setup page. You'll also have to fix the output destinations given in the `xxx.jardesc` files: double-click on them and the output path is specified in the first dialog. The output should go to directory `lib/` of your project main project directory.

To run the code as a console application, you should invoke the `main` method of class `AppManager`. You can run a single instance through Eclipse. (Make sure you set the invocation arguments to something like `-f ../OS/foo.bar.config.ini`). The `echo` (or your `ping`) application in that instance can talk to the `echo service` in the same system instance, and will use RPC to do so.

If you want to run more than one instance, there is an updated `run.sh` in the lib directory that helps. To run an instance of the system that uses config file`OS/foo.bar.config.ini`, you say:

```
./run.sh console foo.bar
```

For this to work, you must first update `lib/OSConsoleApps.jar` by right-clicking on `OSConsoleApps.jardesc` in Eclipse and selecting `Create JAR`whenever you change your code and want to run. (If it complains, you probably didn't update the destination of the the `.jardesc`. See the instruction above.)

We're running a "fully operational" system 23.5/7 at `cse461.cs.washington.edu:46120`. You can use it for debugging. You can, and probably should, also try to interoperate with implementations of other teams. (Note that the code lets you use a DNS host name string like `cse461.cs.washington.edu` whenever an IP address is required.)

What to Turn In

We'd like to automate running your code, in console mode. Ideally, we should be able to take your `RPCService.java`, `RPCCallerSocket.java`, and `Ping.java`, insert them into our unmodified infrastructure code, and run. Use your informed judgement about which source files that means you need to provide.

In addition, hand in a `report.pdf` file containing:

- the names of your team
- a brief description of your RPC implementation, with elaboration as appropriate if you think you've done anything we might find surprising
- a description of what works and what doesn't, if anything doesn't
- the cut-and-paste output of your console-based `ping` to our server on `cse461.cs.washington.edu`
- the cut-and-paste output of native `ping` (the one your machine's OS provides) to the same server, performed at about the same time

Notes

- Our RPC implementation is fairly typical of remote invocation mechanisms, but leaves out one important facility that is often associated with the term "RPC" (say, in an interview): type checking. A purer RPC system will at least try to verify that the arguments provided by the caller match what the callee requires. Our system does almost the opposite: it requires that a `JSONObject` be used to pass arguments, making pretty much any old thing pass Java's type checking, and therefore making it uncommonly easy to create type errors on invocations. I'm bringing this up just in case you're someday talking with someone about your RPC implementation, say in an interview.
- This is a very complicated assignment to try to write instructions for. There is somewhat sophisticated Java. There is Eclipse, and all of its settings. There are the Android plug-ins, and Android development. There's the fact that we're using real networks, which have real configurations, and so real potential problems. All of that means I can't possibly anticipate, much less address, everything that could go wrong, or tell you everything that might be new to you individually (which will vary from person to person). Nonetheless, while I think you'll find the assignment challenging, I'm sure it's well within the scope of what you can do. (And, as a bonus, it's a real development experience, not just an instruction-following one. You either already know you can do that, or you'll be glad to discover that you can.)
- The hardest part of this assignment is that your code has to operate correctly despite all kinds of potential errors. And, you'll encounter those errors when debugging. The errors could be network issues, although those are the least likely. More importantly, your code will be talking to other RPC implementations. Some of those will have outright bugs. Some will have implemented something incompatible with yours, because the specifications weren't specific enough.
- Among the common errors are remote services that simply fail to respond. You're waiting for a response, but the remote service neither tells you that it will never come (by closing the connection) nor actually gives you a response. You can't hang forever. You must therefore arrange for operations involving remote agents to time out, and be prepared for that to happen. The skeleton code includes the few lines required to set up time outs. Your code needs to realize they can happen.
- I have in mind that we'll all bring phones loaded with our implementations to the class following the due date, and try to ping each other. In theory, all our implementations should interoperate just fine. The protocol spec is designed to minimize the opportunity for incompatible implementations. But, we'll see.