Data Networks
Final Project: Simple RPC With Raw Sockets

Dr. Mohammad reza Pakravan

Notes:

- While you can implement this project in any programming language, you are **only** allowed to use Python 3. While it can be argued that implementations in other languages might be more suited to this project, other languages might provide you with unfair advantages in terms of performance, hence they are prohibited.
- **(IMPORTANT)** The only external library that you can use is PyYaml. All of your implementations must utilize raw TCP sockets, thus use of `asyncio` and `WebSocket` is prohibited and codes that utilize these frameworks will not be accepted.
- A separate report is required. You must detail how you implemented your code. Note that requirements and formats stated in the project description are absolutely necessary, anything outside of these formats is up to you, and should be detailed in your report.

  The report is usually in pdf format, yet if you are confident in your coding skills, you may document the code with docstrings as you progress.
- It is **HIGHLY** suggested that you read the entire project description before starting to implement it. Some bonus features may require an overhaul on your previous codes, which will be difficult and time-consuming if you have already implemented the previous parts without them.

Introduction

In this project, you will attempt to code a simple implementation of an RPC toolkit (based loosely on some widely used implementations, like Google's gRPC).

A Remote Procedure Call (RPC) is any protocol that allows for a machine to trigger a procedure on a remote machine, as if it was a local call. More specifically, RPCs abstract away transport layers between two machines, and allows two machines to execute code purely in the application layer.

The main purpose of RPC protocols is to allow for simple application development, without needing to worry about the intricacies of how two machines communicate in the transport layer with a protocol like TCP. Instead, software developers can focus on things that are purely related to the application layer.

An example would be when a machine attempts to run a command that requires a certain operating system (or hardware) that is not available on their local machine, and instead "asks" a remote machine to do it for them and return the result, while anything concerning how this connection is established remains hidden from the programmers.

A similar goal is pursued by REST APIs, which you might have heard of. The main difference between these protocol classes is that RPCs are completely action-oriented and can be tailored to the needs of the developers, whereas REST is fully resource-oriented, meaning that it keeps a tight control on what the source machine can run on the remote machine (consequently, making RPCs safe is much harder compared to REST APIs. Even now, many security breaches happen as a result of un-patched RPC interfaces, a recent example is this one in Microsoft Windows!)

In this project, you will attempt to implement an RPC using raw IPv4 sockets (i.e. the most basic implementation of TCP in most operating systems). We will have no concern about scalability and security at any point in this project.

Terminologies

Before we start, allow us to setup some terminologies so that we can be clear as we progress with explaining your tasks.

Imagine that you are running a private network, each node in the this network can reach any other node in the network with an arbitrary routing scheme. Each node can be reached by pinging a static and known IP address.

Allow us to define the following:

- **RPC Client (also called Client Stub):** A node that triggers remote calls locally on it's own address space and receives the result from another machine.
- **RPC Server (also called Server Stub):** A node that serves requests triggered by an RPC client.
- **Connection pattern:** A graph that shows all possible connections at any given time, by connecting appropriate nodes with an edge.
- **Service Pattern:** When a client and a server trigger an RPC, they will start communicating with

each other in various patterns. These patterns emerge as a result of the transport protocol used during this conversation (which is just TCP in your case).

There is multiple service patterns for TCP, the easiest (and the only one you will implement) is the request-response pattern. In this pattern, the client sends a request and the server responds with a message, then the connection is closed immediately. This is equivalent to a function call.

Project Scenario

Assume that you have a series of machines connected on a network. Your implementation in this project assumes the following (mostly just to simplify things), that **connection only occurs between a pair of client and server nodes**, so the clients and servers alone are not allowed to communicate.

In other words, the connection pattern in your project is a complete bipartite graph (see figure 1 for an example).

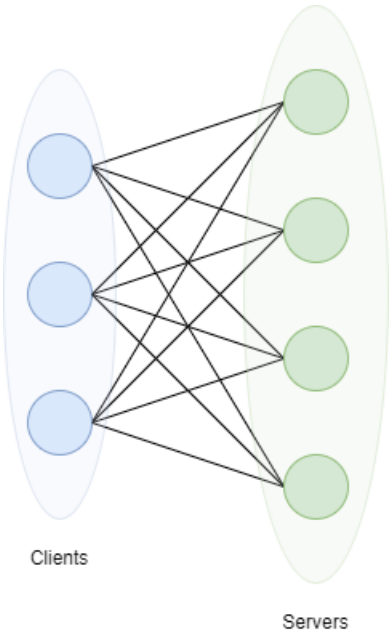

Figure 1: An example of a valid connection pattern. Note that no edge connects two clients or servers

To setup this connection pattern (and some other things) we will use an initializer file that we will call `init.yaml`. You will be provided an example of this file for reference.

The first part of your project will focus on the RPC servers. Appropriately, the second part will implement the RPC clients. The final part of your project will focus on reading the initializer file to setup the connection patterns and registered RPCs.

Protocol Implementation

The protocol that you will implement is fully message-oriented, message types are designated by their *header* field, which will contain a string that will show the message type. Each of these messages will also have a *value* field.

RPC Request Message (RPC-REQ)

This message signifies an attempt by the client to call a remote function on a server. The value for this message must contain:

- **service:** Each session will have a unique service name associated with it. Only clients that are registered to that service may have access to remote calls in that service (this will become more apparent when we discuss the initializer file).
- **rpc:** The name of the rpc script to call on the server.
- **arguments:** A call may have zero or more arguments that are required by the remote function. You may bundle them into a list in this particular field.
- **argument-types:** The receiver side will attempt to decode the argument values (which can be just strings at this point) into their original values by using this field. You don't have to implement all types, the least of them should be `int`, `str`, `float`, `List[int]`, `List[str]` and `List[float]`. There is no need to implement complex types.
- **return-types:** Similar to *argument-types*, but for return values.

RPC Response Message (RPC-RES)

This message is returned as a response to some RPC request message, and must include:

- **return values:** A call may have zero or more return values, similar to the arguments field in RPC request messages. No *return-types* field is needed here.

RPC Exception Message (RPC-EX)

This message is returned by a server upon an exception during the evaluation of some RPC request message. Each of these messages must have the following values:

- **exception-type:** These will be discussed below.
- **exception-message:** Depending on the exception type, this message may include the details and some things about how it should be treated.

The exception types and messages that must be implemented are the following:

- **Execution Exception:** If any exception occurs while executing the RPC in the server, then this exception will be raised. The exception message field may now return the message returned by Python during the function call.
- **Invalid Arguments:** When the called RPC and the types requested do not match (both for return values and arguments), this exception will be raised. The message must inform what return types and argument types may be valid for a call to the desired RPC.
- **Service Not Found:** If the server receiving this request is not part of the service that was declared in the request message, this exception will be raised. The exception message is essentially just an optional field here and can be left empty.
- **RPC Not Found:** If the requested RPC does not exist in the declared service, this message will be returned. The exception message field can be left empty.
- **Client Not Registered:** If a client attempts to call a valid RPC in a valid service, but is not registered as a client of that particular service, this exception will be raised. The exception message field must inform the client that they are not allowed to access that particular service.

You may use JSON to create these messages from Python objects. This can be done easily with the `json` module in Python.

You may use a code similar to the following for this matter (if you don't use type-checking then just ignore them):

```
import json

class Message:
  def __init__(self, type: str, value: str) -> None:
    self.type = type
    self.value = value

  @classmethod
  def from_json(cls, message_as_bytes) -> Object:
    return Message(**json.loads(message_as_bytes.decode()))

  def to_json(self) -> bytes:
    return json.dumps(self.__dict__).encode()
```

Part I: Implementing RPC Servers

Your RPC server must implement the above protocol and respond to `RPC-REQ` messages.

Considering that the server may experience some exceptions while handling some of the clients requests, your server should not crash upon such exceptions and must keep working even after such events, being able to serve other clients.

Each RPC is essentially a Python script that can be called from the server locally. An RPC is uniquely located by using 2 parameters, `service-name` and `rpc-name`.

As a suggestion (and not an enforcement on your coding style and implementation), you can store these functions in a hierarchy of files.

This means that one way of doing this, would be to create and maintain the following file structure:

```
root
├── <your rpc-server codes>
└── services
    ├── service1
    │   ├── rpc1.py
    │   └── rpc2.py
    ├── service2
    │   ├── rpc1.py
    │   └── rpc2.py
    └── service3
```

Your server may only implement request-response service patterns.

To run a Python script from another script, you may leverage the `importlib` module.

```python
import importlib.util
import sys

"""
Example of running a function called 'rpc' in a file named 'test.py'
with a list of arguments in another python script.
"""

arguments = [1, "string"]

spec = importlib.util.spec_from_file_location("test", <path-to-test.py>)
module = importlib.util.module_from_spec(spec)
sys.modules["test"] = module
spec.loader.exec_module(module)

try:
  outputs = getattr(module, 'rpc')(*arguments)
except Exception as e:
  # If execution fails with an exception
  print(e)
```

(BONUS) Overloaded RPCs

An RPC can be overloaded, this means that RPCs can be addressed not only with their service name and rpc name, but also their argument types (similar to how it works in programming languages).

If you attempt to implement this part, be sure to explain how you treat these RPCs, as you effectively need 3 parameters instead of 2 in order to identify these RPCs.

(BONUS) Logging

A server must always log any connection attempts. To this end, you may use the `logging` module in python and create a log. This log must contain the following:

- Record of all attempts to connect to the server.
- The address of any client that managed to successfully connect to the server.

- Records of unauthorized access by clients, this means that if at any moment, a client attempts to use a service that is not registered for them, it should be logged.
- Records of successful RPC executions, or exceptions.

All of these entries must have a timestamp. For this purpose, you can use UNIX timestamps or the `datetime` module. You may also append thread names for debugging, though they are not needed as part of the implementation.

Part II: Implementing RPC Clients

Clients will attempt to configure the servers and may change some existing configuration on them, but mostly, clients only request RPCs and responses.

The end goal is that a client can run a code locally on it's own machine, but may call for an RPC during this process, the client will then send appropriate `RPC-REQ` messages.

Please note that request-reponse messages are assumed to be **blocking**, this means that your client code will halt it's execution while it waits for the server to respond.

Part III: Initializer File

Before sessions can start, the clients must subscribe to appropriate RPC services and the servers must receive the code for each of their RPCs.

To this end, and initializer file will act as the initial configuration of servers and clients. This file will be called `init.yaml`. Here, YAML is used as the appropriate markup language, as unlike JSON, it supports additional metadata (like comments) that make it appropriate for such purposes.

An example file will be provided to you. But in brief, the file contains the following fields:

- **network:** This field will contain information on the clients and servers. This includes names, ip addresses and open ports.
- **services:** This field will register the available services that clients can subscribe to. Each service has a *provider* and some *tenants*. Providers are servers that will resolve the RPC calls for this service, and tenants are authorized clients that can access these services.

  Most importantly, services will have an *rpcs* field, which will declare valid RPCs for that service.
- **rpc:** Here, the name, argument types and return types of an rpc are declared. Here, you may provide the path to the source code of this RPC, which is assumed to be a single Python file with a function of the same name (just for simplicity).

You must create a program that reads this file, and sends each of the RPC codes to the appropriate server. If you implemented the aforementioned file hierarchy in the server, you must instruct the server to modify them as needed.

By default, any previous RPCs will be deleted and the servers will restart when a new file is initialized. This process can be triggered from any node.

No strict protocol like the one used for the RPC executions is needed here and you are free to introduce one as you see fit, although nothing complicated is required here, as you may only contact each server once.

(BONUS) Support For Runtime Update

As we said, your previous implementation requires you to restart the servers once the initialization file is changed.

This is not a very clean implementation and may cause some problems for active sessions. Modify the server code such that a change in the initialization file, does not necessarily require you to reboot the servers. Please don't think complicated, this has a very easy solution!

(BONUS) Make It More Efficient!

Your project does not implement any efficient encoding for it's messages. This may cause the initialization phase to be very sluggish and take unnecessarily long, as it attempts to send long strings of codes to the servers.

This is especially annoying if you only make small changes to the initializer file, as most of the RPCs that you will redistribute are already on the servers!

A simple solution is to actually compare codes before attempting to send them, one efficient way is the following.

Check for the existence of the RPC in the destination server. If the RPC does not exist, send it. If it does, then ask the server to take the SHA256 hash of the RPC script and send it to you while you do the same with your own script.

If the hashes are the same, then these are the same codes with very high probability, so just move on to the next RPC. If not, then these RPCs are different and the one in the server needs to be replaced with yours.

To take the SHA256 hash of an RPC code, you may do the following:

```
import hashlib

rpc_code_file = open("./path/to/rpc.py", "r")
rpc_code = rpc_code_file.read()
hash_of_rpc_code = hashlib.sha256(rpc_code.encode()).hexdigest()
```

Also note that you can easily check for the existence of an RPC, just by leveraging RPC-EX messages!

Final Notes:

- The code snippets are only a suggestion, you may simply ignore them. You also have complete freedom (outside of using external libraries) when it comes to design decisions (especially for bonus sections)
- If your are using older Python versions (like 3.5), you may need to supplement appropriate encoding names when using `str.encode()` and `bytes.decode()` methods.

  Most of the time, this encoding is passed as just a string, like `'utf-8'`.
- Be sure to provide a use case for your implementation.
- When writing your code, you may just test it on your local machine for debugging, but for the final evaluation, you may be asked to run your code over a virtual machine that is bridged to your local machine.

  The details will be explained in a separate session.
- Use PyYaml to parse the initializer file into a nested dictionary object.
- The mapping between server names and (ip, port) pairs is assumed to be always known by the clients.

What Should I Do?

You must write the following Python codes:

- `rpc-server.py`: This code will implement the RPC server on each node. Note that this *exact, same* code should be executed on *every* server node. So make sure that you allow inputs for things like IP addresses and ports.
- `rpc-client.py`: Similar to `rpc-server.py`, this code must be able to reach any of the servers given the server name, and call for an RPC, then return the result to the code that called it.
- `initializer.py`: Reads the `init.yaml` file and distributes the RPCs among the servers.

And of course, if you don't document your code thoroughly, be sure to provide a report in pdf format. Compress all of these files into a **.zip** file named **PROFECT_<STUDENT-ID>.zip** and upload it to Quera before the deadline.