BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)

CS ZG501 Introduction to Parallel and Distributed Programming

Lab#1

Note: Please use programs under *Code* directory supplied with this sheet. Do not copy from this sheet.

The lab has the following objectives:

1. UDP socket programming
2. TCP socket programming
3. Remote procedure calls (RPC) - gRPC

Socket Programming

Both forms of communication (UDP and TCP) use the socket abstraction, which provides an endpoint for communication between processes. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process. For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs. Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Each computer has a large number (2^{16}) of possible port numbers for use by local processes for receiving messages. Any process may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer. However, any number of processes may send messages to the same port. Each socket is associated with a particular protocol – either UDP or TCP.

UDP datagram communication:

A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries. If a failure occurs, the message may not arrive. A datagram is transmitted between processes when one process sends it and another receives it. To send or receive messages a process must first create a socket bound to an Internet address of the local host and a local port. A server will bind its socket to a server port – one that it makes known to clients so that they can send messages to it. A client binds its socket to any free local port. The receive method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.

Consider the client code given in UDPClient.py:

```
1.  from socket import *
```

```
2· serverName = 'hostname'
3· serverPort = 12000
4· clientSocket = socket(AF_INET, SOCK_DGRAM)
5· message = input('Input lowercase sentence:')
6· clientSocket·sendto(message·encode(),(serverName, serverPort))
7· modifiedMessage, serverAddress = clientSocket·recvfrom(2048)
8· print(modifiedMessage·decode())
9· clientSocket·close()
```

Consider the following code given in UDPServer.py:

```
1· from socket import *
2· serverPort = 12000
3· serverSocket = socket(AF_INET, SOCK_DGRAM)
4· serverSocket·bind(('', serverPort))
5· print("The server is ready to receive")
6· while True:
7·     message, clientAddress = serverSocket·recvfrom(2048)
8·     modifiedMessage = message·decode()·upper()
9·     serverSocket·sendto(modifiedMessage·encode(), clientAddress)
```

Q?

1. Use Visual Code or any other editor to open the files in "Code" directory. Run UDPS-erver.py using the following command

   python UDPServer.py

   In another terminal run the UDP Client:

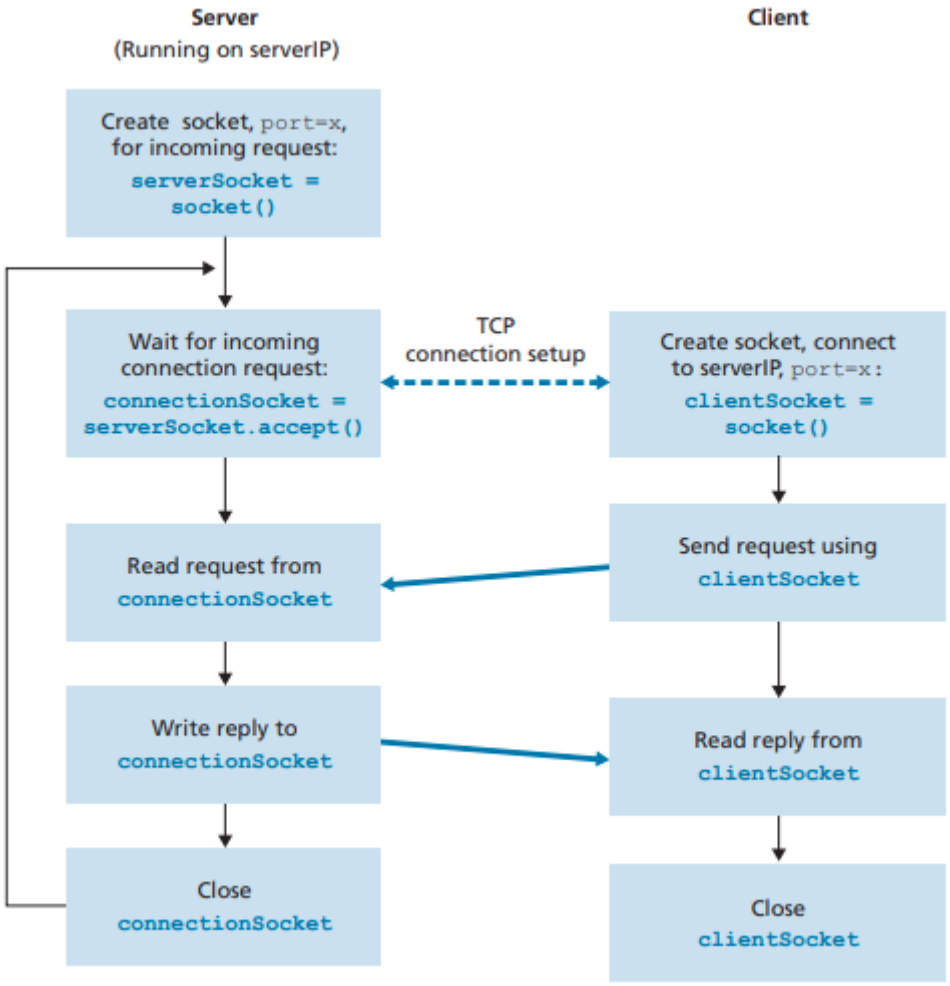
   python UDPClient.py

2. Change the client code such that client keeps asking user input in a loop and dis-plays the output.
3. Change the server code such that after reading the message, it reverses the string and sends.
4. Change the server code such that it can take an expression such as "23+24" and evaluates it and gives the result to the client.

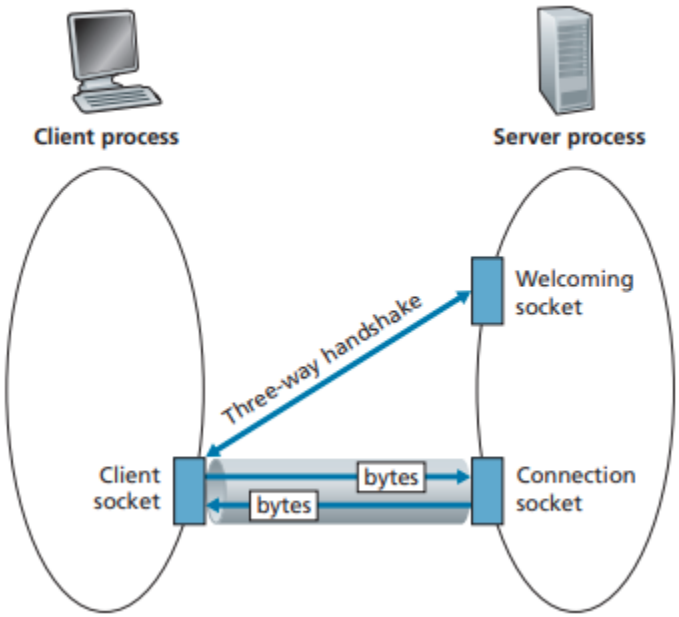
   <u>Socket Programming with TCP</u>

Unlike UDP, TCP is a connection-oriented protocol. This means that before the client and server

can start to send data to each other, they first need to handshake and establish a TCP connection. One end of the TCP connection is attached to the client socket and the other end is attached to a server socket. When creating the TCP connection, we associate with it the client socket address (IP address and port number) and the server socket address (IP address and port number). With the TCP connection established, when one side wants to send data to the other side, it just drops data into the TCP connection via its socket. This is different from UDP, for which the server must attach a destination address to the packet before dropping it into the socket. The following diagram illustrates interactions between TCP server and client.



TCP server has two types of sockets as shown in the diagram below. First one is listening socket or a welcoming socket. This socket is always there to welcome new client connections. When a client is accepted, a new socket is created that is called connection socket.



First, as in the case of UDP, the TCP server must be running as a process before the client attempts to initiate contact. Second, the server program must have a special door—more precisely, a special socket—that welcomes some initial contact from a client process running on an arbitrary host. With the server process running, the client process can initiate a TCP connection to the server. This is done in the client program by creating a TCP socket. When the client creates its TCP socket, it specifies the address of the welcoming socket in the server, namely, the IP address of the server host and the port number of the socket. After creating its socket, the client initiates a three-way handshake and establishes a TCP connection with the server. The three-way handshake, which takes place within the transport layer, is completely invisible to the client and server programs. During the three-way handshake, the client process knocks on the welcoming door of the server process. When the server "hears" the knocking, it creates a new door—more precisely, a new socket that is dedicated to that particular client. In our example below, the welcoming door is a TCP socket object that we call serverSocket; the newly created socket dedicated to the client making the connection is called connectionSocket.

TCPClient.py

```
1· from socket import *
2· serverName = 'servername'
3· serverPort = 12000
4· clientSocket = socket(AF_INET, SOCK_STREAM)
5· clientSocket·connect((serverName,serverPort))
6· sentence = input('Input lowercase sentence:')
7· clientSocket·send(sentence·encode())
```

```
8· modifiedSentence = clientSocket·recv(1024)
9· print('From Server: ', modifiedSentence·decode())
10· clientSocket·close()
```

TCPServer.py

```
1· from socket import *
2· serverPort = 12000
3· serverSocket = socket(AF_INET,SOCK_STREAM)
4· serverSocket·bind(('',serverPort))
5· serverSocket·listen(1)
6· print('The server is ready to receive')
7· while True:
8·  connectionSocket, addr = serverSocket·accept()
9·  sentence = connectionSocket·recv(1024)·decode()
10· capitalizedSentence = sentence·upper()
11· connectionSocket·send(capitalizedSentence·encode())
12· connectionSocket·close()
```

Q?

1. Use Visual Code or any other editor to open the files in "Code" directory. Run TCPS-erver.py using the following command.

   python TCPServer.py

   Run TCPClient.py in another terminal.

   Give input in the client and do you see any response from server?

   Change server code to print client ip and port on the console.

2. The code given in TCPServer.py is for an Iterative Server i.e. only one connection is handled at a time. Look at TCPServerConcurrent.py. This file has code that creates a new thread for every new client connection.

   Run TCPServerConcurrent.py

   Run TCPClient.py

   Run more instances of TCPClient to observe concrrency.

3. Introduce a global vaiable in TCPServerConcurrent.py. Let each client increment it. Use the following lock construct to protect the shared variable.

   lock=threading.Lock()

   lock.acquire()

   lock.release()

Group communication:

This section will be updated after covering Multicast communication in the class.

Remote Procedure Calls (RPC)

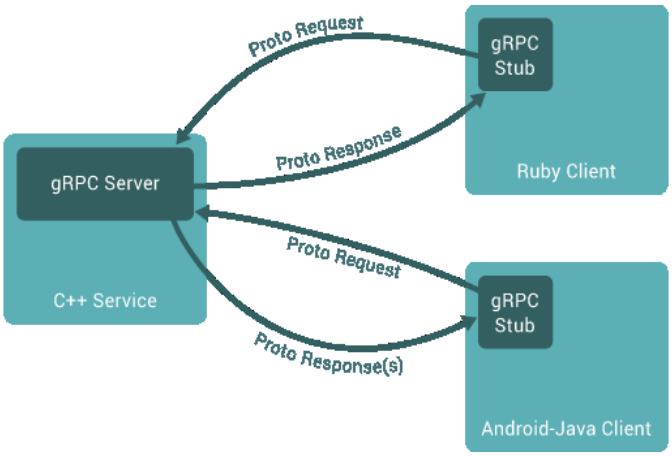
The remote procedure calls (RPC) approach extends the common programming abstraction of the procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it is local. The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call.

gRPC

gRPC, which stands for “Google Remote Procedure Call," is an open-source, high-performance framework developed by Google that facilitates efficient communication between distributed systems. It's designed to enable communication between various services, regardless of the programming languages they are implemented in or the platforms they run on. gRPC is based on HTTP/2, a modern protocol that offers improved efficiency over its predecessor, HTTP/1.

At its core, gRPC allows developers to define the structure of the data and operations that their services provide using Protocol Buffers (protobufs). Protocol Buffers are a language-agnostic mechanism for serializing structured data, making it easy to define the structure of messages and services in a concise and readable format.

We can then compile the protocol buffer to create types and functions to interact with our



API upon which we can build our client and server programs in any language.

Installation

Make sure you have python3 installed (It's installed by default on Linux and macOS). Then, use pip install grpc via the command:

```
pip install grpcio-tools
```

Also, install numpy if it's not already installed (for exercise 2).

Addition as RPC call

The goal of this tutorial is to get started by implementing a simple addition gRPC in Python.

1. Creating the protocol buffer file

   Create a folder named "protobufs" in the initially empty directory. In the folder, create a file named "addition.proto". This file will contain the description of our RPC. Write the following code into the file:

```
syntax = "proto3";

message AdditionRequest{
   int32 a = 1;
   int32 b = 2;
}

message AdditionResponse{
```

```
    int32 result = 1;
}

service Addition{
    rpc Addn(AdditionRequest) returns (AdditionResponse);
}
```

We start by defining the proto syntax version we'll use.

Next, we specify the request and response formats.

Lastly, we define the service and add the RPC Addn that takes the AdditionRequest as input and returns an AdditionResponse.

We can now use this file to create the python classes we'll use for the RPC.

2. Once the proto file is created, add another folder to the root directory of the project named addition. This folder will contain the .py files we'll generate using the .proto file. Now, to generate the files, open the terminal and navigate to the addition folder. Then, enter the following command (this must be done every time you change your proto file):

   python -m grpc_tools.protoc -I ../protobufs --python_out=. --grpc_python_out=. ../protobufs/addition.proto

   This can be broken down as:

   - python -m grpc_tools.protoc runs the protobuf compiler, which will generate Python code from the protobuf code.
   - -I ../protobufs tells the compiler where to find files that your protobuf code imports. You don't actually use the import feature, but the -I flag is required nonetheless.
   - --python_out=. --grpc_python_out=. tells the compiler where to output the Python files. As you'll see shortly, it will generate two files, and you could put each in a separate directory with these options if you wanted to.
   - ../protobufs/addition.proto is the path to the protobuf file, which will be used to generate the Python code.

   This adds the files addition_pb2_grpc.py and addition_pb2.py to the addition folder. These files include Python types and functions to interact with your API. The compiler will generate client code to call an RPC and server code to implement the RPC.

3. Now, we'll generate the client and server python file. First, we start with the *server* file. In the addition folder, add a file named addition.py and enter the following code into it:

```
import grpc
from addition_pb2 import AdditionResponse
import addition_pb2_grpc

class AdditionService(addition_pb2_grpc.AdditionServicer):
    def Addn(self, request, context):
        return AdditionResponse(result=(request.a + request.b))

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    addition_pb2_grpc.add_AdditionServicer_to_server(AdditionService(), server)
    server.add_insecure_port("[::]:50059")
    server.start()
    server.wait_for_termination()

if __name__ == '__main__':
    serve()
```

4. Next, we can generate the client side python code. Create another file called addition_client.py. Add the following code into it:

```
import grpc
from addition_pb2_grpc import AdditionStub
from addition_pb2 import AdditionRequest
```

`from concurrent import futures`
`import grpc`
`from addition_pb2 import AdditionResponse`
`import addition_pb2_grpc`
	We import grpc. We also import concurrent (provides a high-level interface for asynchronously executing callables).

`class AdditionService(addition_pb2_grpc.AdditionServicer):`
`  def Addn(self, request, context):`
`  return AdditionResponse(result=(request.a + request.b))`
	We define the AdditionService class. This is the implementation of out Addn function. Here, it just returns a response with the addition of the two numbers.

`def serve():`
`  server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))`
`  addition_pb2_grpc.add_AdditionServicer_to_server(AdditionService(), server)`
`  server.add_insecure_port("[::]:50059")`
`  server.start()`
`  server.wait_for_termination()`
	We then create the serve function. First, we create the server. We add the port we want to communicate over.

`if __name__ == '__main__':` `  serve()`	As long as it's not imported as a module, execute the serve function.

```
channel = grpc.insecure_channel("localhost:50059")
stub = AdditionStub(channel)
request = AdditionRequest(a=1, b=2)
print(stub.Addn(request))
```

We create the channel followed by a stub to call our function. We encode our request and finally remotely call our Addn function.

5. To execute, run the addition.py file in one terminal (we're starting the server first). Next, run the addition_client.py file and observe the result.

That was a simple gRPC example that called an addition function. Now think about how this can be extended. We can pass not only two integers, but thousands and perform computations on a remote machine. Moreover, we can also use gRPCs to create microservices that can abstract away the implementational details, leaving us with more modular and a more maintainable codebase.

Q?

A. Try to extend the addition RPC such that the RPC takes in three arguments: an operator in string format and two integers. Depending on what the operator is, perform the computation and return the result.
B. Now, create another RPC that takes in an array of integers and returns them in sorted format (Hint: bytes is a valid datatype in proto and numpy arrays can be encoded and decoded to base64 format)
C. Create a microservice for an online shopping cart using gRPC. The microservice API should provide calls for adding to cart, removing from cart and fetch current cart contents. Create a simple frontend using flask. Use a products list containing the product objects (each product has an id, name, price).

End of lab1