**CHAPTER 3**

**INTER MACHINE COMMUNICATION**

This chapter discusses the protocols and the method used for communication. The method of communication used between the Application, Bank and the Subsidiary Servers are Socket Programming. As much detail is provided it will be required to understand the core of the implementation. The theory behind this communication is explained in this section.

**3.1 Socket Programming**

**General Principles**

Sockets allow communication between two different processes on the same or different machines.

In practice "socket" usually refers to a socket in an Internet Protocol (IP) network (where sockets may be called Internet sockets), in particular for the Transmission Control Protocol (TCP), which is a protocol for one-to-one connections. In this context, sockets are assumed to be associated with a specific socket address, namely the IP address and a port number for the local node, and there is a corresponding socket address at the foreign node (other node), which itself has an associated socket, used by the foreign process. Associating a socket with a socket address is called binding.

Note that while a local process can communicate with a foreign process by sending or receiving data to or from a foreign socket address, it does not have access to the foreign socket itself, nor can it use the foreign socket descriptor, as these are both internal to the foreign node. For example, in a connection between 10.20.30.40:4444 and 50.60.70.80:8888 (local IP address:local port, foreign IP address:foreign port), there will also be an associated socket at each end, corresponding to the internal representation of the connection by the protocol stack on that node, which are referred to locally by numerical socket descriptors, say 317 at one side and 922 at the other. A process on node 10.20.30.40 can request to communicate with node 50.60.70.80 on port 8888 (request that the protocol stack create a socket to communicate with that destination), and once it has created a socket and received a socket descriptor (317), it can communicate via this socket by using the descriptor (317): the protocol stack will then forward data to and from node 50.60.70.80 on port 8888. However, a process on node 10.20.30.40 cannot request to communicate with "socket 922" or "socket 922 on node 50.60.70.80": these are meaningless numbers to the protocol stack on node 10.20.30.40.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

**Socket Usages**

A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

**Socket Types**

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.

Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

* Stream Sockets − Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order − "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
* Datagram Sockets − Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets − you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).
* Raw Sockets − These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.
* Sequenced Packet Sockets − They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

**3.2 Network Address**

The IP host address, or more commonly just IP address, is used to identify hosts connected to the Internet. IP stands for Internet Protocol and refers to the Internet Layer of the overall network architecture of the Internet.

An IP address is a 32-bit quantity interpreted as 48-bit numbers or octets. Each IP address uniquely identifies the participating user network, the host on the network, and the class of the user network.

An IP address is usually written in a dotted-decimal notation of the form N1.N2.N3.N4, where each Ni is a decimal number between 0 and 255 decimal (00 through FF hexadecimal).

**3.3 Client-Server Model**

Most of the Net Applications use the Client-Server architecture, which refers to two processes or two applications that communicate with each other to exchange some information. One of the two processes acts as a client process, and another process acts as a server.

**3.2.1 Client Process**

This is the process, which typically makes a request for information. After getting the response, this process may terminate or may do some other processing.

Example, Internet Browser works as a client application, which sends a request to the Web Server to get one HTML webpage.

**3.2.2 Server Process**

This is the process which takes a request from the clients. After getting a request from the client, this process will perform the required processing, gather the requested information, and send it to the requestor client. Once done, it becomes ready to serve another client. Server processes are always alert and ready to serve incoming requests.

Example − Web Server keeps waiting for requests from Internet Browsers and as soon as it gets any request from a browser, it picks up a requested HTML page and sends it back to that Browser.

Note that the client needs to know the address of the server, but the server does not need to know the address or even the existence of the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

**3.2.3 2-tier and 3-tier architectures**

There are two types of client-server architectures –

* 2-tier architecture − In this architecture, the client directly interacts with the server. This type of architecture may have some security holes and performance problems. Internet Explorer and Web Server work on two-tier architecture. Here security problems are resolved using Secure Socket Layer (SSL).
* 3-tier architectures − In this architecture, one more software sits in between the client and the server. This middle software is called ‘middleware’. Middleware are used to perform all the security checks and load balancing in case of heavy load. A middleware takes all requests from the client and after performing the required authentication, it passes that request to the server. Then the server does the required processing and sends the response back to the middleware and finally the middleware passes this response back to the client. If you want to implement a 3-tier architecture, then you can keep any middleware like Web Logic or WebSphere software in between your Web Server and Web Browser.

**3.2.4 Types of Server**

There are two types of servers you can have –

* Iterative Server − This is the simplest form of server where a server process serves one client and after completing the first request, it takes request from another client. Meanwhile, another client keeps waiting.
* Concurrent Servers − This type of server runs multiple concurrent processes to serve many requests at a time because one process may take longer and another client cannot wait for so long. The simplest way to write a concurrent server under UNIX is to fork a child process to handle each client separately.

**3.2.5 How to Make Client**

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. Both the processes establish their own sockets.

The steps involved in establishing a socket on the client side are as follows −

* Create a socket with the socket() system call.
* Connect the socket to the address of the server using the connect() system call.
* Send and receive data. There are a number of ways to do this, but the simplest way is to use the read() and write() system calls.

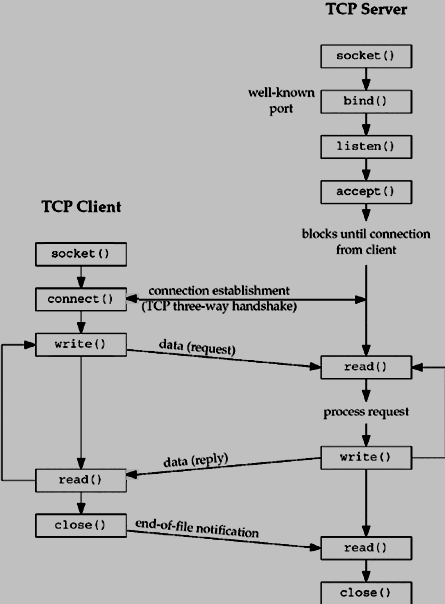
**3.2.6 How to make a Server**

The steps involved in establishing a socket on the server side are as follows −

* Create a socket with the socket() system call.
* Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
* Listen for connections with the listen() system call.
* Accept a connection with the accept() system call. This call typically blocks the connection until a client connects with the server.
* Send and receive data using the read() and write() system calls.

**3.2.7 Client and Server Interaction**

Following is the diagram showing the complete Client and Server interaction –



Socket Client Server

**CHAPTER 4**

**COMMUNICATION BETWEEN APP AND BANK SERVER**

This chapter discusses the protocols and the method used for communication between the Android Application and the Bank Server. As much detail is provided to understand the core of the implementation.

**4.1 Communication Diagram between the App and Bank Server**

There are 2 important stages that encompass the communication between the Android Application and the Bank Server. These are as follows:

* OTP Generation and Verification
* Passing OTP, Login Id and Password for Retrieval/Splitting

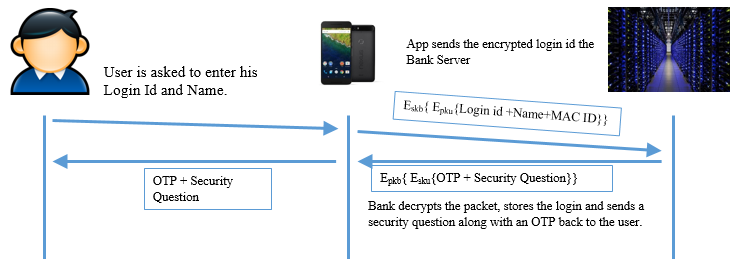
Each of these 2 stages are called (except for in the process of Signup) in each of the 2 processes of the Bank Cryptosystem we are trying to implement. The 2 basic processes are as follows:

* Signup/Registration (Splitting)
* Login (Retrieval)

**4.1.1 Signup/Registration (Password Splitting and Distribution Stage)**

In this process, the user enters his password for the first time into the App after following the registration process which involves entering his Login Id and Name. The initial split of the password takes place in this stage following Shamir’s Algorithm following which the pieces of the password are sent to the Subsidiary Servers based on the location pass where they are stored in a database consisting of the Login Id, Location Pass and Password Piece received from Bank Server.

**4.1.1.1 OTP Generation and Verification Overview**



The Communication between the App and the Bank Server for sending the OTP is one using Sockets. A TCP Socket is opened at the Application Side and the Bank Side. Packet consisting of Login Id and Name is correspondingly encrypted and sent over this line. Reply is got over the same line from the app.

**4.1.1.1.1 Socket Creation in Android Application**

Sockets are created in Android by using the Socket class. The general syntax is as follows:

Socket ([String](https://developer.android.com/reference/java/lang/String.html) host, int port)

The implementation of Socket in the Application is as follows:

**public class** PasswordSignup **extends** AsyncTask<Void, Void, Void>

.

.

socket = **new** Socket(**dstAddress**, **dstPort**);  
**firstpacket** = **OTP**+ **loginid** + **password**;  
String firstpacketsend;

In order to send data from the Application to the Bank Server, an instance of the class PrintWriter is created. The following is its usage in our code:

OutputStream outStream = **socket**.getOutputStream();  
System.***out***.println(**"Packet to be sent "**+firstpacketsend);  
**out1** = **new** PrintWriter(outStream, **true**);  
**try** {  
 TimeUnit.***SECONDS***.sleep(2);  
 **out1**.println(**OTKey**);  
 TimeUnit.***SECONDS***.sleep(2);  
 **out1**.println(firstpacketsend);  
}

**4.1.1.1.2 Socket Creation in Bank Server**

The Bank Server has been written in python to facilitate querying and storage of the Password pieces which will be explained later on. Sockets are created and implemented in the following fashion:

# server.py

import socket

from random import randint

import pickle

from bitarray import bitarray

import hashlib

import csv

import time

import pyotp

import pickle

# create a socket object

import json

publkey = "001100010011000100110000001100110111"

serversocket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

serversocket2 = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

serversocket3 = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# get local machine name

# host = socket.gethostname()

port = 9999

port2 = 8888

port3 = 7777

# bind to the port

serversocket.bind((' ', port))

serversocket2.bind((' ', port2))

serversocket3.bind((' ', port3))

# queue up to 10 requests

serversocket.listen(10)

serversocket2.listen(5)

serversocket3.listen(5)

while True:

# establish a connection

clientsocket3,addr = serversocket.accept()

print("Got a connection from %s" % str(addr))

# privatekey = clientsocket.recv(1024)

option = clientsocket3.recv(1024)

print("Option is %s" %option.decode('ascii'));

elif int(option.decode('ascii'))==2:

totp = pyotp.TOTP('base32secret3232')

r = totp.now()

clientsocket3.send(r.encode())

print(r)

clientsocket3.close()

# print (macadd.decode('ascii'))

print("It is reaching here 0")

clientsocket5,addr = serversocket.accept()

one\_time\_key = clientsocket5.recv(1024)

firstpacket = clientsocket5.recv(1024)

print("Packet 1 : %s" %firstpacket.decode('ascii'))

# passw1 = clientsocket3.recv(1024)

rstring = int(firstpacket.decode('ascii')) ^ int(publkey,2)

password = str(rstring)

print("Packet 1 Decoded: %s" %password)

OTP = int(password[0:4])

login\_id = password[4:9]

password\_n = int(password[9:13])

# one\_time\_key = password[13:45]

print(one\_time\_key);

location\_pass = 12

csv\_storage\_loaction\_pass = int(one\_time\_key,16)^location\_pass

fieldnames = ['Account\_No','Location\_Pass']

with open("D:/Final Year Project/Test1/test.csv", "w", newline='') as fp:

writer = csv.DictWriter(fp,fieldnames=fieldnames)

writer.writerow({'Account\_No': login\_id, 'Location\_Pass': csv\_storage\_loaction\_pass})

clientsocket,addr = serversocket.accept()

print("Got a connection from %s" % str(addr))

clientsocket2,addr2 = serversocket.accept()

print("Got a connection from %s" % str(addr2))

chosen\_prime = 5915587277

hash\_object = hashlib.md5(login\_id.encode())

login = hash\_object.hexdigest()

# chosen\_prime = 48112959837082048697

random\_r = randint(0,chosen\_prime)

pass1\_n = (password\_n+random\_r)%chosen\_prime

pass1 = str(pass1\_n)

dict1 = {login:pass1}

resp1 = pickle.dumps(dict1)

pass2\_n = (password\_n+2\*random\_r)%chosen\_prime

pass2 = str(pass2\_n)

dict2 = {login:pass2 }

resp2 = pickle.dumps(dict2)

clientsocket.send(option)

clientsocket2.send(option)

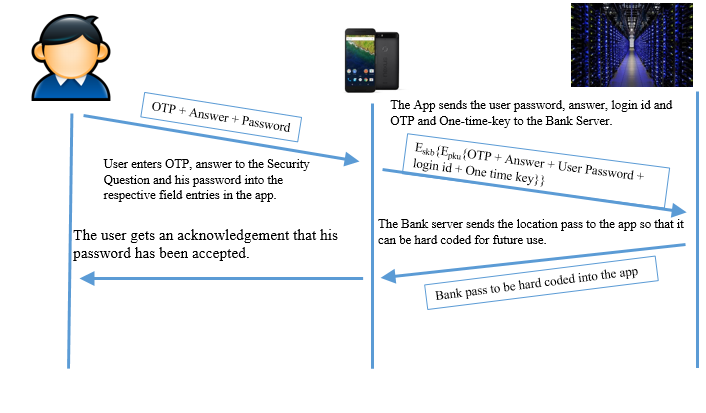
clientsocket.send(resp1)

clientsocket2.send(resp2)

else :

print("Option is not present and is wrong");

**4.1.1.2 Passing OTP, Login Id and Password for Retrieval/Splitting**



The Communication between the App and the Bank Server for passing OTP, Login Id and Password for Splitting is one using Sockets. A TCP Socket is opened at the Application Side and the Bank Side. Packet consisting of OTP, Answer, Login Id and One-Time-Key is correspondingly encrypted and sent over this line. Reply is got over the same line from the app with the bank pass/location pass to be hardcoded in the application.

**4.1.1.1.1 Socket Creation in Android Application**

The implementation of Socket in the Application is as follows:

**public class** Client **extends** AsyncTask<Void, Void, Void>

.

.

socket = **new** Socket(**dstAddress**, **dstPort**);

In order to send data from the Application to the Bank Server, an instance of the class PrintWriter is created. The following is its usage in our code:

ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream(  
 1024);  
**byte**[] buffer = **new byte**[1024];  
**int** bytesRead;  
InputStream inputStream = socket.getInputStream();  
OutputStream outStream = socket.getOutputStream();  
PrintWriter out =  
 **new** PrintWriter(outStream, **true**);

**4.1.1.1.2 Socket Creation in Bank Server**

The Bank Server has been written in python to facilitate querying and storage of the Password pieces which will be explained later on. Sockets are created and implemented in the following fashion:

# server.py

import socket

from random import randint

import pickle

from bitarray import bitarray

import hashlib

import csv

import time

import pyotp

import pickle

# create a socket object

import json

publkey = "001100010011000100110000001100110111"

serversocket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

serversocket2 = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

serversocket3 = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# get local machine name

# host = socket.gethostname()

port = 9999

port2 = 8888

port3 = 7777

# bind to the port

serversocket.bind((' ', port))

serversocket2.bind((' ', port2))

serversocket3.bind((' ', port3))

# queue up to 10 requests

serversocket.listen(10)

serversocket2.listen(5)

serversocket3.listen(5)

while True:

# establish a connection

clientsocket3,addr = serversocket.accept()

print("Got a connection from %s" % str(addr))

# privatekey = clientsocket.recv(1024)

option = clientsocket3.recv(1024)

print("Option is %s" %option.decode('ascii'));

if int(option.decode('ascii'))==1:

firstpacket = clientsocket3.recv(1024)

print("Packet 1 : %s" %firstpacket.decode('ascii'))

secondpacket = clientsocket3.recv(1024)

# passw1 = clientsocket3.recv(1024)

rstring = int(firstpacket.decode('ascii')) ^ int(publkey,2)

password = str(rstring)

print("Packet 1 Decoded: %s" %password)

clientsocket3.close()

clientsocket,addr = serversocket.accept()

print("Got a connection from %s" % str(addr))

clientsocket2,addr2 = serversocket.accept()

print("Got a connection from %s" % str(addr2))

# print("Make sure that the password that you are about to enter does not exceed 9 digits")

# password = input('Enter the password to be stored ')

.

.

.

.

.

.

with open("D:/Final Year Project/Test1/test2.csv", "w", newline='') as fp:

writer = csv.DictWriter(fp,fieldnames=fieldnames)

# writer.writeheader()

writer.writerow({'Login\_Id': csv\_storage\_loaction\_pass})

def trial(existing\_bit,one\_time\_key):

hash\_object = hashlib.md5(one\_time\_key.encode())

one\_time\_key\_hash = hash\_object.hexdigest()

loc\_pass = int(one\_time\_key\_hash,16) ^ existing\_bit

existing\_bit = int(one\_time\_key,16) ^ loc\_pass

return (existing\_bit);

.

.

.

.

.

.

clientsocket.send(option)

clientsocket2.send(option)

clientsocket.send(resp1)

clientsocket2.send(resp2)

print("Packet 2 : %s" %secondpacket.decode('ascii'))

retreive\_query = str(secondpacket.decode('ascii'))

#print(retreive\_query)

.

.

.

.

with open('D:/Final Year Project/Test1/test2.csv', 'rt') as f:

reader = csv.reader(f, delimiter=',') # good point by @paco

for row in reader:

content = trial(int(row[0]),one\_time\_key)

print (content)

fieldnames = ['Login\_Id']

with open("D:/Final Year Project/Test1/test2.csv", "w", newline='') as fp:

writer = csv.DictWriter(fp,fieldnames=fieldnames)

# writer.writeheader()

writer.writerow({'Login\_Id': content})

hash\_object\_retreive = hashlib.md5(login\_id\_retreive.encode())

login\_query = hash\_object\_retreive.hexdigest()

clientsocket.send(login\_query.encode('ascii'))

clientsocket2.send(login\_query.encode('ascii'))

passw1 = clientsocket.recv(1024)

print("The Piece from Client 1 is %s" %passw1.decode('ascii'))

retreived\_p1 = int(passw1.decode('ascii'))

passw2 = clientsocket2.recv(1024)

passw3 = passw2.decode('ascii')

# StringUtils.substring(passw2.decode('ascii'), 0, 10)

print("The Piece from Client 2 is %s" %passw3[:10])

retreived\_p2 = int(passw3)

retreived\_pass = (2\*retreived\_p1-retreived\_p2)%chosen\_prime

print("The retreived password is %s" %retreived\_pass)

packet = retreived\_pass\*10000+OTP

#print(packet)

packet\_str = str(packet)

hash\_object = hashlib.md5(packet\_str.encode())

regenerated\_packet = hash\_object.hexdigest()

print("hashed pass with OTP is %s"%str(regenerated\_packet))

password\_ascii\_retreive = ''.join(str(ord(c)) for c in password\_char\_retreive)

regenerated\_ascii\_packet = ''.join(str(ord(c)) for c in regenerated\_packet)

print(password\_ascii\_retreive)

print(regenerated\_ascii\_packet)

print(int(password\_ascii\_retreive) - int(regenerated\_ascii\_packet))

if ((int(password\_ascii\_retreive) - int(regenerated\_ascii\_packet))==0) :

print("access granted");

else :

print("access granted");

clientsocket.close()

clientsocket2.close()

else :

print("Option is not present and is wrong");

**CHAPTER 5**

**COMMUNICATION BETWEEN BANK SERVER AND SUBSIDIARY SERVER**

This chapter discusses the protocols and the method used for communication between the Subsidiary Server and the Bank Server. As much detail is provided to understand the core of the implementation.

**4.1 Communication Diagram between the App and Bank Server**

There is 1 important stage that encompass the communication between the Subsidiary and the Bank Server. These are as follows:

* Passing OTP, Login Id and Password for Retrieval/Splitting

This stage is called in each of the 2 processes of the Bank Cryptosystem we are trying to implement. The 2 basic processes are as follows:

* Signup/Registration (Splitting)
* Login (Retrieval)

**4.1.1 Signup/Registration (Password Splitting and Distribution Stage)**

In this process, the user enters his password for the first time into the App after following the registration process which involves entering his Login Id and Name. The initial split of the password takes place in this stage following Shamir’s Algorithm following which the pieces of the password are sent to the Subsidiary Servers based on the location pass where they are stored in a database consisting of the Login Id, Location Pass and Password Piece received from Bank Server.