**Cryptography and Network Security Lab**

**Name: Rushikesh R. Ware**

**PRN: 2020BTECS00012**

**Batch: B2**

**Diffie-Hellman Key Exchange**

**Aim:**

Implementation of Diffie-Hellman Key Exchange.

**Theory:**

It is a practical method for public exchange of a secret key. This is used in a number of commercial products.

Steps:

* all users agree on global parameters:
  + large prime integer or polynomial q
  + α a primitive root mod q
* each user (eg. A) generates their key
  + chooses a secret key (number): xA < q
  + compute their **public key**: yA = αxA mod q
* each user makes public that key yA
* shared session key for users A & B is K:

K = yAxB mod q (which **B** can compute)

K = yBxA mod q (which **A** can compute)

Example:

* K is used as session key in private-key encryption scheme between Alice and Bob
* if Alice and Bob subsequently communicate, they will have the **same** key as before, unless they choose new public-keys
* attacker needs an x, must solve discrete log
* users Alice & Bob who wish to swap keys:
* agree on prime q=353 and α=3
* select random secret keys:

A chooses xA=97, B chooses xB=233

* compute public keys:

yA=397 mod 353 = 40 (Alice)

yB=3233 mod 353 = 248 (Bob)

* compute shared session key as:

KAB= yBxA mod 353 = 24897 = 160 (Alice)

KAB= yAxB mod 353 = 40233 = 160 (Bob)

**Code:**

**Server:**

from generate\_prime import is\_prime, generate\_prime\_no

import socket

# Function to find mod: a^m mod n

def findExpoMod(a, m, n):

    # Decimal to binary conversion

    m\_bin = bin(m).replace("0b", "")

    # Convert it into list (individual characters)

    m\_bin\_lst = [int(i) for i in m\_bin]

    # Initialize the list

    a\_lst = [a]

    # Functions to perform operations

    # If next value = 0

    def oneOperation(num):

        return (num\*num) % n

    # If next value = 1

    def twoOperation(num):

        return (a \* oneOperation(num)) % n

    for j in range(len(m\_bin\_lst)):

        if j+1 == len(m\_bin\_lst):

            break

        if(m\_bin\_lst[j+1] == 0):

            a\_lst.append(oneOperation(a\_lst[j]))

        else:

            a\_lst.append(twoOperation(a\_lst[j]))

    return a\_lst[-1]

def is\_primitive\_root(alpha, q):

    L = []

    for i in range(1, q):

        L.append(findExpoMod(alpha, i, q))

    for i in range(1, q):

        if L.count(i) > 1:

            L.clear()

            return False

        return True

# Initialize Socket

HOST = 'localhost'

PORT = 12345

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_address = (HOST, PORT)  # Server address and port

server\_socket.bind(server\_address)

server\_socket.listen(1)

print(f"Server started at: {HOST}:{PORT}")

print("Waiting for a client to connect...")

client\_socket, client\_address = server\_socket.accept()

print("Client connected: ", client\_address)

# DH Key-exchange

# Choose prime no. 'q'

print("Choose a large integer prime number(q):")

gen\_r = input("Do you want to generate the prime number automatically ? [y/n]\n")

if gen\_r == 'y':

    dig\_p = int(input("Enter the number of digits in prime number: "))

    q = generate\_prime\_no(dig\_p)

    print(f"q = {q}")

elif gen\_r == 'n':

    q = int(input("Enter a large prime number:\n"))

    if not is\_prime(q):

        print(f"Entered number is not prime!")

        exit()

else:

    print("Invaild choice!")

    exit()

# Choose primitive root 'alpha'

print("Choose primitive root (alpha):")

gen\_pr = input("Do you want to find the primitive root automatically ? [y/n]\n")

if gen\_pr == 'y':

    for a in range(2, q):

        if is\_primitive\_root(a, q):

            alpha = a

            break

    print(f"Alpha = {alpha}")

elif gen\_pr == 'n':

    alpha = int(input(f"Enter the primiitive root of {q}:\n"))

    if not is\_primitive\_root(alpha, q):

        print(f"This is not the primitive root!")

        exit()

else:

    print("Invaild choice!")

    exit()

# Server's Private key

Xa = int(input(f"Enter the private key for A (Xa) [less than {q}]:\n"))

if Xa >= q:

    print("Private key must be less than choosen prime!")

    exit()

# Server's Public Key

Ya = findExpoMod(alpha, Xa, q)

# Send this data to client

print(f"Server's Public Key is: {Ya}")

print("Sending Public Key to client...")

send\_Ya = str(Ya).encode()

client\_socket.sendall(send\_Ya)

print("Sending choosen large prime to client...")

send\_q = str(q).encode()

client\_socket.sendall(send\_q)

print("Sending primitive root to client...")

send\_alpha = str(alpha).encode()

client\_socket.sendall(send\_alpha)

print("Waiting for Client's Public Key...")

# Receive Client's Public Key

received\_Yb = client\_socket.recv(1024)

Yb = int(received\_Yb.decode())

print(f"Received Public Key of Client: {Yb}")

# Compute shared key and send to client

Ks = findExpoMod(Yb, Xa, q)

print(f"Server's Shared Key is: {Ks}")

print("Sending it to client...")

send\_Ks = str(Ks).encode()

client\_socket.sendall(send\_Ks)

# Receive Client's Shared Key

print("Waiting for Client's Shared Key...")

received\_Kc = client\_socket.recv(1024)

Kc = int(received\_Kc.decode())

print(f"Received Client's Shared Key as: {Kc}")

if Ks == Kc:

    print("Both shared keys are equal\nKeys exchanged successfully!")

else:

    print("Both shared keys aren't equal.\nKey exchange failed!")

client\_socket.close()

server\_socket.close()

**Client:**

import socket

# Function to find mod: a^m mod n

def findExpoMod(a, m, n):

    # Decimal to binary conversion

    m\_bin = bin(m).replace("0b", "")

    # Convert it into list (individual characters)

    m\_bin\_lst = [int(i) for i in m\_bin]

    # Initialize the list

    a\_lst = [a]

    # Functions to perform operations

    # If next value = 0

    def oneOperation(num):

        return (num\*num) % n

    # If next value = 1

    def twoOperation(num):

        return (a \* oneOperation(num)) % n

    for j in range(len(m\_bin\_lst)):

        if j+1 == len(m\_bin\_lst):

            break

        if(m\_bin\_lst[j+1] == 0):

            a\_lst.append(oneOperation(a\_lst[j]))

        else:

            a\_lst.append(twoOperation(a\_lst[j]))

    return a\_lst[-1]

HOST = 'localhost'

PORT = 12345

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_address = (HOST, PORT)  # Server address and port

client\_socket.connect(server\_address)

print(f"Connected to server at: {HOST}:{PORT}")

# Receive the server's public key, q, alpha

received\_Ya = client\_socket.recv(1024)

Ya = int(received\_Ya.decode())

print(f"Received server's public key as: {Ya}")

received\_q = client\_socket.recv(1024)

q = int(received\_q.decode())

print(f"Received large prime as: {q}")

received\_alpha = client\_socket.recv(1024)

alpha = int(received\_alpha.decode())

print(f"Received alpha as: {alpha}")

# Client's Private Key

Xb = int(input(f"Enter the private key for B (Xb) [less than {q}]:\n"))

if Xb >= q:

    print("Private key must be less than choosen prime!")

    exit()

# Client's Public Key

Yb = findExpoMod(alpha, Xb, q)

print(f"Client's Public key is: {Yb}")

# Send this to Server

print("Sending Client's Public Key to Server...")

send\_Yb = str(Yb).encode()

client\_socket.sendall(send\_Yb)

# Receive Server's Shared key

received\_Ks = client\_socket.recv(1024)

Ks = int(received\_Ks.decode())

print(f"Received Server's Shared key as: {Ks}")

# Compute shared key and send to server

Kc = findExpoMod(Ya, Xb, q)

print(f"Client's Shared key is: {Kc}")

print("Sending it to server...")

send\_Kc = str(Kc).encode()

client\_socket.sendall(send\_Kc)

if Kc == Ks:

    print("Both shared keys are equal\nKeys exchanged successfully!")

else:

    print("Both shared keys aren't equal.\nKey exchange failed!")

client\_socket.close()

**Generate\_prime:**

import random

# Miller-Rabin primality test

def is\_prime(n, k=5):

    if n <= 1:

        return False

    if n <= 3:

        return True

    # Write n as d\*2^r + 1

    r, d = 0, n - 1

    while d % 2 == 0:

        r += 1

        d //= 2

    # Witness loop

    for \_ in range(k):

        a = random.randint(2, n - 2)

        x = pow(a, d, n)

        if x == 1 or x == n - 1:

            continue

        for \_ in range(r - 1):

            x = pow(x, 2, n)

            if x == n - 1:

                break

        else:

            return False

    return True

# Function to generate a random n-digit number

def generate\_large\_number(digits):

    lower\_bound = 10 \*\* (digits - 1)

    upper\_bound = 10 \*\* digits - 1

    return random.randint(lower\_bound, upper\_bound)

def generate\_prime\_no(digits):

    while True:

        number = generate\_large\_number(digits)

        if is\_prime(number):

            return number

**Output:**

