DISTRIBUTED SYSTEM FINAL REPORT

Remote shell with MPI

Group 5 - ICT
University of Science and Technology of Hanoi

Vũ Đinh Anh	BI9 - 037
Nguyễn Lan Hương	BI9 - 114
Nguyễn Hồng Quang	BI9 - 194

March 2021

Contents

Chapter 1

Introduction

1.1 Definitions

1.1.1 Remote shell

The Remote shell (rsh) [1] is a command line computer program that can execute shell commands on another computer across a computer network called the hostname machine as another user.

To do so, the remote shell must connect to a service called Remote shell daemon (rshd) on the hostname machine. Typically, this daemon uses the Transmission control protocol (TCP) port number 514.

1.1.2 Message passing interface

Message passing interface (MPI) [2] is a portable message-passing standard designed for academia and industry to function on a myriad of parallel computing architectures. It is meant to provide essential virtual topology, synchronization and communication functionality between a set of processes that has been mapped to nodes, servers or computer instances.

MPI consists of:

- a header file mpi.h
- a core library of routines and functions
- a runtime system

1.2 Objective and approach

In this project, we are going to create a remote shell to communicate between a client and a server through a Message Passing Interface.

To do this, we are going to implement a Python program using the following as main libraries:

- pycryptodome: to encode then encrypt and decode then decrypt data for security
- mpi4py: to set up a remote shell with MPI

Chapter 2

Method

Source code can be found in dinhanhx/ds2021

2.1 Environment setup

This project operates on GNU/LINUX. We need to setup the following:

- Python 3
- an MPI package
- Python packages: mpi4py and pycryptodome

Use these commands:

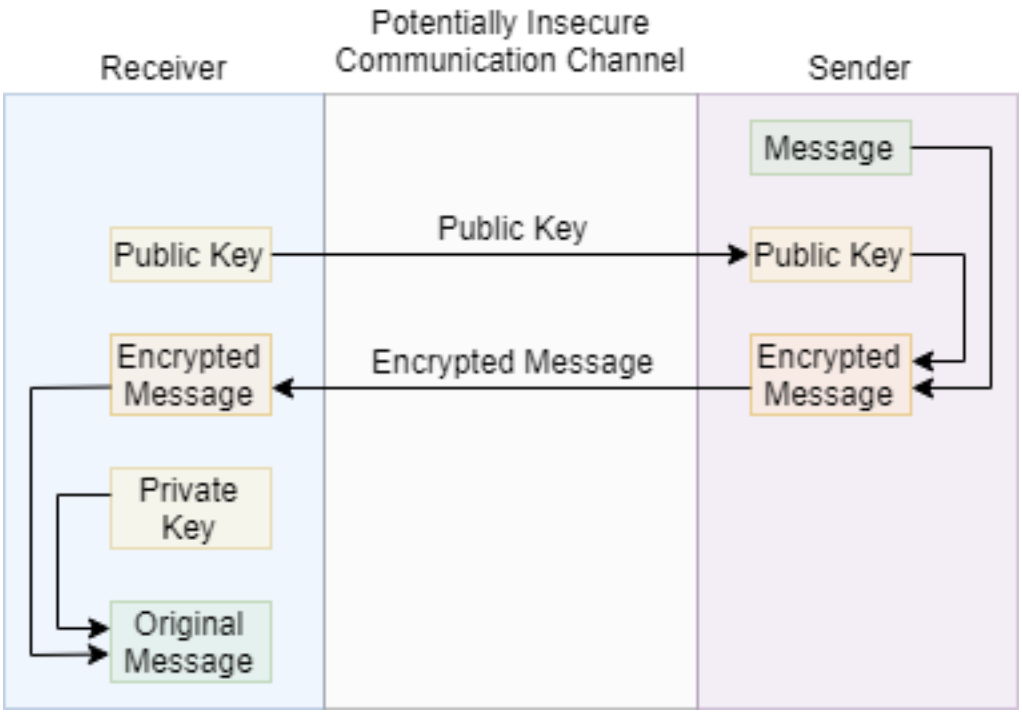
```
sudo apt install mpi
pip3 install mpi4py pycryptodome
```

2.2 Security

2.2.1 Protocol

A secure data transmission system always needs a cryptosystem. In this project, we use the RSA cryptosystem [3] as it is fairly simple to implement.

Basically, the RSA algorithm generates a pair of distinct sequences called the public key and the private key. The receiver keeps the private key in secret whil publishing the public key to any sender. The sender encodes their message using the public key, which then can only be decrypted by anyone with the private key.



2.2.2 Implementation

We create a Python file named security.py and put all the functions for data encryption and decryption inside. These functions will later be imported to be uses in the MPI.

Key generation

Basically, this function generates a public and a private key based on the RSA cryptosystem and then store them in two separate txt files.

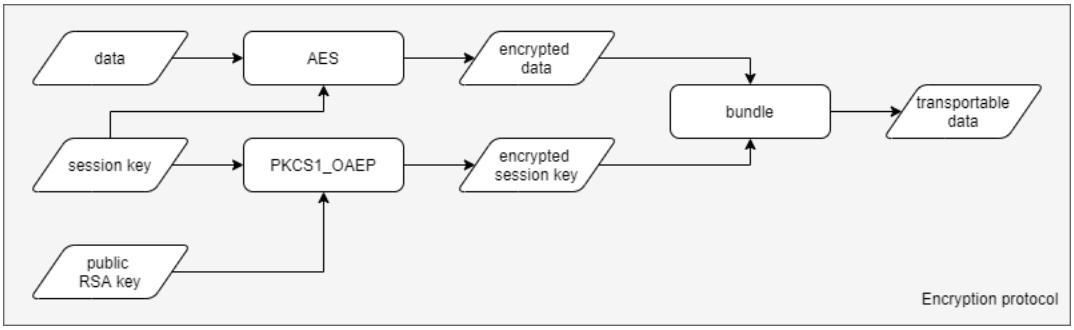
```
def gen_pair(save_dir: str='.'):
    key = RSA.generate(2048) # Generate 2048-bits key
    Path(save_dir).mkdir(exist_ok=True)
    private = key.export_key()
    private_path = Path(save_dir).joinpath('private.key.txt')
    with open(private_path, 'wb') as f:
        f.write(private)

    # Generate public key from a RSA key
    public = key.publickey().export_key()
    public_path = Path(save_dir).joinpath('public.key.txt')
    with open(public_path, 'wb') as f:
        f.write(public)
```

This function imports the keys from txt files to be used in other functions.

```
def import_key(private_key_file: str, public_key_file):
    private_key_file = Path(private_key_file)
    public_key_file = Path(public_key_file)
```

```
    private_key = RSA.import_key(open(private_key_file, 'r').read())
    public_key = RSA.import_key(open(public_key_file, 'r').read())
    return private_key, public_key
```

Encryption

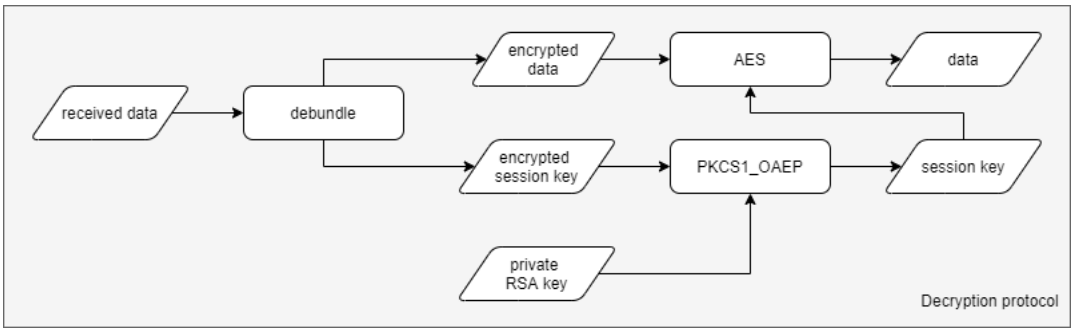


This function encodes a string of data into a bundle of bytes which is transportable. It takes three arguments:

- A string of data that needs to be encrypted
- previously generated RSA public key
- a randomly generated session key

```
def encode_encrypt(data: str, public_key: RSA.RsaKey):
    encoded_data = data.encode()
    session_key = get_random_bytes(16)

    cipher_RSA = PKCS1_OAEP.new(key=public_key)
    encrypted_session_key = cipher_RSA.encrypt(session_key)

    cipher_AES = AES.new(session_key, AES.MODE_EAX)
    encrypted_data, tag = cipher_AES.encrypt_and_digest(encoded_data)

    bundle = (encrypted_session_key, cipher_AES.nonce, tag, encrypted_data)
    return dumps(bundle)
```

Decryption



This function turns the received data into a bundle of bytes that contain the encrypted data and the encrypted session key. Using these two and the previously generated RSA private key, it returns the decrypted data.

```
def decrypt_decode(bundle: bytes, private_key: RSA.RsaKey):
    session_key, nonce, tag, encrypted_data = loads(bundle)

    cipher_RSA = PKCS1_OAEP.new(key=private_key)
    session_key = cipher_RSA.decrypt(session_key)

    cipher_AES = AES.new(session_key, AES.MODE_EAX, nonce)
    encoded_data = cipher_AES.decrypt_and_verify(encrypted_data, tag)

    return encoded_data.decode()
```

2.3 Remote shell with MPI

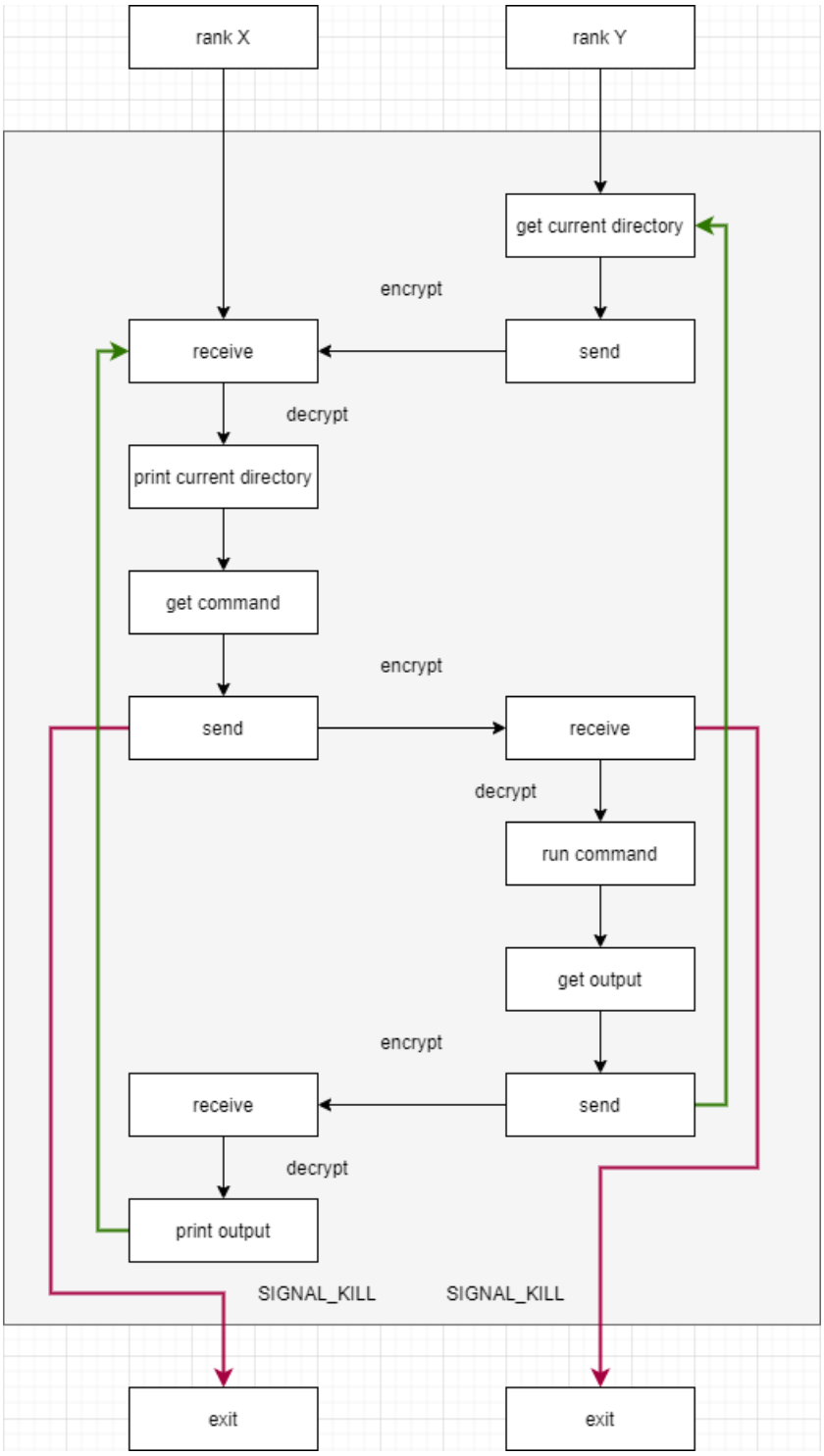
2.3.1 Protocol

The graph below demonstrates the basic protocol for the client-server interaction that we implemented. The left column (rank X) represents the client while the right (rank Y) is the server. After a connection has been set up between two nodes, the program uses their rank X and Y to identify which side is the server and which side is the client.

Firstly, the server side sends the encrypted current directory to the client. After decryption, the user is able know which directory they are working on and create a command accordingly.

That command is then encrypted and sent to the server. After decryption, the server runs that command and sends the output back to the client.

The whole process above can be repeated as many times as desired until a kill signal is raised by the user. The connection between two nodes is cut afterwards.



2.3.2 Implementation

Required libraries

Beside the mpi4py library to setup MPI connection, we need the subprocess and os library to execute shell commands and get their outputs.

We also import the written functions for data encryption and decryption in the security.py file from the above section.

```
import mpi4py.MPI as MPI
import subprocess
import os

from security import import_key, encode_encrypt, decrypt_decode
```

MPI shell

This function sets up a connection between two side. Inside of it, we create nested functions that define the behaviors of the client and the server according to the protocol. Rank 0 denotes the client and rank 1 denotes the server.

```
def mpi_shell():
    """Setup a connection between two proccesses/two cores/two nodes/two clusters"""

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()

    if rank == 0:
        # client-like side
        # write mpi function to send and receive here

    elif rank == 1:
        # server-like side
        # write mpi function to send and receive here


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

Client side

```
    if rank == 0:
        # client-like side

        # setup keys
        client_private_key = 'rank0/private.key.txt'
        server_public_key = 'rank1/public.key.txt'
        client_private_key, server_public_key = import_key(client_private_key, server_public_key)

        # enter client-server-like loop
        while True:
            cwd = decrypt_decode(comm.recv(source=1, tag=42), client_private_key)

            cmd = input(f'{cwd}#>')

            comm.send(encode_encrypt(cmd, server_public_key), dest=1, tag=42)

            output = decrypt_decode(comm.recv(source=1, tag=42), client_private_key)
            print(output)
```

Server side

```
elif rank == 1:
    # server-like side

    # setup keys
    server_private_key = 'rank1/private.key.txt'
    client_public_key = 'rank0/public.key.txt'
    server_private_key, client_public_key = import_key(server_private_key, client_public_key)

    # enter server-client-like loop
    while True:
        cwd = os.getcwd()
        comm.send(encode_encrypt(cwd, client_public_key), dest=0, tag=42)

        cmd = decrypt_decode(comm.recv(source=0, tag=42), server_private_key)

        output = subprocess.getoutput(cmd)
        comm.send(encode_encrypt(output, client_public_key), dest=0, tag=42)
```

2.4 Conclusion

Although the program that we built is extremely simple, it is still able to illustrate the basic essence of a remote shell MPI between a client and a server. Another advantage of it is that it also incorporated a simple secure system, which is the RSA scheme.

However, we still have not developed to evaluate the performance of the system or taken into consideration problems such as data loss during transmission or incapacity to deal with a large number of requests and so on.

Bibliography

[1] https://www.unix.com/man-page/mojave/1/rsh.

[2] https://www.open-mpi.org/.

[3] https://medium.com/@jinkyulim96/algorithms-explained-rsa-encryption-9a37083aaa62.