Universidade Federal do Sul e Sudeste do Pará
Faculdade de Computação e Engenharia Elétrica
Curso de Engenharia da Computação
Iago Costa das Flores - 201840601017
Juliana Batista da Silva - 201740601024

Sistemas Distribuídos
Trabalho avaliativo 01

Marabá
2021

Iago Costa das Flores - 201840601017
Juliana Batista da Silva - 201740601024

Sistemas Distribuídos
Trabalho Avaliativo 01

Relatório apresentado no curso de Engenharia da Computação, turma de 2018 como obtenção de nota parcial na disciplina de Sistemas Distribuídos, ministrada pelo Professor Dr. Warley Muricy Valente Junior.

Marabá
2021

UNIFESSPA

Sumário

1 - Introdução

Este relatório tem como objetivo apresentar a solução encontrada pela equipe para o trabalho final 01 da disciplina de sistemas distribuídos, que consiste em elaborar um código em qualquer linguagem de programação, que seja capaz de realizar a comunicação “multicast” entre 3 processos, e que eles possam trocar informações entre si, onde essas informações solicitadas serão criptografadas pelo processo detentor da mensagem/informação e descriptografadas pelo processo que fez a solicitação, utilizando o sistema de chaves públicas e privadas, atentando-se para uso de “sockets” e comunicação unicast na troca de informações diretas entre os processos.

2 - Atividades

Para a elaboração do código foi utilizada a linguagem de programação python. Para compor um processo, seguindo os exemplos de server UDP e client disponibilizados pelo professor da disciplina nas atividades anteriores, utilizou-se um modelo de um server com protocolo UDP e um client colocados juntos em uma pasta, assim a parte de comunicação multicast entre os processos se dá pelo uso do tipo do servidor, no caso com protocolo UDP, permitindo que todos os processos possam receber uma mesma mensagem ao mesmo tempo, no estilo broadcast.

A utilização de um cliente junto ao código do servidor se faz necessária pois sem ela não teria como enviar as mensagens de requisição das informações. Vale lembrar que a comunicação dos processos em questão ocorrem pela mesma porta e utilizando o mesmo ip, no caso o localhost, o código foi então replicado para os outros 2 processos, sendo assim necessário o uso de 6 terminais para fazer com que todo o processo funcione perfeitamente, sendo necessário um terminal para cada cliente e um terminal para cada servidor.

2.1 - Servidor com protocolo UDP

```
1   #importação de bibliotecas para configuração de socket udp multicast
2   import sys, struct, socket
3   import json
4
5   from datetime import datetime
6
7   # importação de bibliotecas para geração de chaves públicas e privadas
8   from cryptography.hazmat.primitives import serialization
9   from cryptography.hazmat.primitives.asymmetric import padding
10  from cryptography.hazmat.primitives import hashes
11
12  # define o nome do server do par
13  serverName = '225.0.0.1'
14  #define a porta para conexão
15  serverPort = 9000
16
```

Figura 01: Código servidor linhas 1 à 15.

Da linha 2 a linha 10 são feitas as importações para o funcionamento do código, na linha 13 e 15 são setados o ip nominal do par em questão e a porta respectivamente, vale ressaltar que apesar dos pares estarem recebendo ips diferentes para nomeação, a comunicação dentre eles ocorre no mesmo ip e porta.

```
17  # configurando socket udp
18  serverSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
19
20  # Allow multiple sockets to use the same PORT number
21  serverSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
22
23  # bind udp port
24  serverSocket.bind(('', serverPort))
25
26  # set mcast group
27  mreq = struct.pack('4sl', socket.inet_aton(serverName), socket.INADDR_ANY)
28  serverSocket.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
29
30  # imprimindo na tela o início do servidor do par
31  data_e_hora_atuais = datetime.now()
32  data_e_hora_em_texto = data_e_hora_atuais.strftime('%d/%m/%Y %H:%M')
33  print("running server: " + serverName + ":" + str(serverPort) + " - " +···
34
35  nomeArquivoKeyPublic = ""
36
```

Figura 02: Código servidor linhas 17 à 32.

As linhas 18, 21 e 24 são referentes a configuração do socket para a comunicação entre os processos.

Nas linhas 27 e 28 o servidor do par é adicionado no grupo de comunicação multicast.

As linhas 31 e 32 são referentes a data e hora que serão mostradas no momento da entrada do par no grupo multicast. Na linha 33 é feito o print na tela das informações do servidor e porta do par em questão.

Na linha 36 mostra a variável onde será armazenada a chave pública do par para compartilhamento entre os processos.

```
38  cont = {"escolhido": 0, "falhas": 0, "sucesso": 0}
39
40  # try e except para salvamento ou leitura do arquivo rel.txt que contém o histórico de cada par
41  try:
42      with open('rel1.txt', 'r') as key_file:
43          result = key_file.read()
44  except:
45      with open('rel1.txt', 'w') as key_file:
46          key_file.write(str(cont))
47
```

Figura 03: Código servidor linhas 38 à 47.

Na linha 38 uma variável é utilizada para guardar as informações dos processos de requisição dos arquivos associados ao par, onde servirá para decidir o critério de escolha quando o arquivo solicitado estiver em mais de 1 processo.

Da linha 41 a 46 é utilizado um try e except para realizar a leitura ou escrita do arquivo que terá o valor armazenado da variável já citada na linha 38.

```
49  try:
50      while 1:
51
52          data, addr = serverSocket.recvfrom(4096)
53
54          # opção de recebimento da chave pública
55          if (data.decode() == "pub_key"):
56              # recebe nome para chave publica
57              client, addr = serverSocket.recvfrom(4096)
58              # guarda nome para chave em uma variável
59              nomeArquivoKeyPublic = client.decode()
60
61              # try para salvar a chave pública
62              try:
63
64                  # abrindo arquivo
65                  arq = open(nomeArquivoKeyPublic + '.pem', 'wb')
66                  data, addr = serverSocket.recvfrom(4096) # recebendo chave pública em bytes
67
68                  arq.write(data) # salvando chave pública em arquivo
69                  arq.close() # fechando arquivo
                                        ⋮
182
```

Figura 04: Código servidor linhas 49 à 69..

A partir da linha 49 até a linha 182, um looping infinito é criado para que o par permaneça sempre online, dentro desse looping o servidor analisa a mensagem de requisição e dependendo do conteúdo é definida algumas das opções, dentre elas:
-Receber a chave pública do par que fará uma requisição a ele, após o recebimento a chave é salva ou sobrescrita em um arquivo na pasta do par em questão.
-Fazer um teste de ping para testar a conectividade entre ele e o par cliente.
-Realizar a requisição de um arquivo que será criptografado utilizando a chave pública do par cliente.

```
192    except KeyboardInterrupt:
193        # caso clique control + c ele interrompe a execução do bind e finaliza o script
194        print('done')
195        sys.exit(0)
196
```

Figura 05: Código servidor linhas 192 à 195.

As linhas 192 e 195, servem para finalizar o looping infinito de conexão do par.

2.2 - Cliente

```
18   #configurado socket udp para o cliente do par
19   clientSocket = socket(AF_INET, SOCK_DGRAM)
```

Figura 06: Código cliente linhas 18 à 19.

Da parte do cliente, as linhas e códigos iniciais são praticamente iguais às do servidor, a não ser pela configuração do socket, que no servidor é feita para receber as mensagens e no cliente é feita para enviar.

```
31   # Gerando a chave privada para esse par específico
32 > private_key = rsa.generate_private_key( ···
35   )
36
37   # Gerando chave pública a partir da chave privada desse par
38   public_key = private_key.public_key()
39
40   # Convertendo a chave privada em bytes para salvar como arquivo .pem
41 > private_bytes_key = private_key.private_bytes( ···
45
46   # Convertendo a chave pública em bytes para envio
47 > public_bytes_key = public_key.public_bytes( ···
50
51   # Salvando chave privada em arquivo
52   arq = open('par1_pri_key.pem', 'wb')
53   arq.write(private_bytes_key)
54   arq.close()
```

Figura 07: Código cliente linhas 32 à 54.

Nas linhas 32 e 38 são geradas as chaves privada e pública respectivamente, nas linhas 41 e 47 elas são convertidas em bytes para que possam ser salvas em um arquivo .pem.

Da linha 52 a 54 a chave privada do par é salva em um arquivo e armazenada na pasta do mesmo.

```
56  # enviando e avisando sobre chave pública para os outros processos
57  data_e_hora_atuais = datetime.now()
58  data_e_hora_em_texto = data_e_hora_atuais.strftime('%d/%m/%Y %H:%M')
59  print("sending public key - " + data_e_hora_em_texto)
60  clientSocket.sendto("pub_key".encode(), (serverName, serverPort)) # avisando sobre o envio da chave pública
61  clientSocket.sendto("par1_pub_key".encode(), (serverName, serverPort)) # enviando o nome do par
62  clientSocket.sendto(public_bytes_key, (serverName, serverPort)) # enviando a chave em bytes
63  data_e_hora_atuais = datetime.now()
64  data_e_hora_em_texto = data_e_hora_atuais.strftime('%d/%m/%Y %H:%M')
65  print("------------ connection closed public key sent - " +···
```

Figura 08: Código cliente linhas 56 à 65.

Da linha 57 65 o par envia aos demais processos do grupo sua chave pública, printando nos terminais a data e hora do envio juntamente da chave pública em si, as informações são enviadas por meio do socket configurado anteriormente para o envio das mensagens.

```
68  try:
69      # Loop infinito para realizar ping nos pares e solicitar arquivos para baixar
70      while 1:
71
72          # espera entrada para saber qual função vai ativar se é o ping ou file (para baixar arquivos)
73          sentence = input('Digite o nome do comando:')
74
75          # if que realiza o ping no multcast os pares conectados irão retornar pong
76          if (sentence == "ping"):
77              #envio da mensagem ping
78              clientSocket.sendto(sentence.encode(), (serverName, serverPort))
79
80              # while fica recebendo todos os envios de pong dos pares
81              while 1:
82                  try:
83                      dados, addr = clientSocket.recvfrom(2048) #recebe os valores
84                      print(dados, addr) #printa na tela o pong recebido
85                  except:
86                      break # se tiver alguma exceção ele finaliza o while
87                  if not dados:
88                      break # se não tiver recebido mais dados ele também finaliza
89              data_e_hora_atuais = datetime.now()
90              data_e_hora_em_texto = data_e_hora_atuais.strftime('%d/%m/%Y %H:%M')
91              print("------------ connection closed pong - " + data_e_hora_em_texto) #avisando a finalização do pong
                                                    :
                                                    :
179                                                 :
```

Figura 09: Código cliente linhas 68 à 91.

Das linhas 68 a 179 um looping infinito é criado assim como no código do servidor, com a diferença de que no código do cliente será enviada as requisições referentes a cada opção do looping, enquanto que do lado do servidor o looping é feito para tratar essas requisições feitas por parte do cliente.

```
181 except KeyboardInterrupt:
182     # caso clique control + c ele interrompe a execução do bind e finaliza o script
183     print('done')
184     sys.exit(0)
```

Figura 11: Código cliente linhas 181 à 184.

Da linha 182 a 184 ao final do looping, a conexão do cliente poderá ser finalizada caso o usuário pressione “ctrl + c”.

3 - Conclusão

Ao longo do desenvolvimento do código, algumas dificuldades foram encontradas, dentre elas a escolha da linguagem para desenvolvimento do código, apesar de python ser uma linguagem de programação de fácil entendimento e diminuir consideravelmente o uso de linhas de códigos na elaboração dos scripts, o código de exemplo disponibilizado pelo professor está escrito em Java, que é uma linguagem orientada a objeto e fortemente tipada, com uma escrita bem mais pesada, que torna a elaboração do código mais trabalhoso, por outro lado é possível observar de forma mais clara o que é feito em cada linha de código devido a orientação a objetos que é marca registrada da linguagem.

Devido a diferença entre a linguagem dos scripts de apoio e a linguagem utilizada para o desenvolvimento do trabalho, foi necessário escrever novos códigos de apoio, utilizando somente a lógica de implementação dos códigos já prontos.

Na execução dos códigos, alguns “bugs” ocorreram, ao utilizar arquivos maiores para serem criptografados/descriptografados, ocorre um erro relacionado ao processo em si, que talvez pode ser causado por algum caractere especial no meio das palavras ao qual não é possível ser identificado no processo de criptografia. Em alguns momentos foram mostrados “buffs” indesejados nos terminais e um erro que não dá prosseguimento ao processo de requisição de arquivo do cliente, onde é preciso parar a execução do código dos servidores e começar o processo da execução novamente.

Mesmo com alguns erros e dificuldades na implementação do trabalho, os pontos presentes no roteiro foram atendidos, sendo o código capaz de executar o que foi inicialmente proposto no documento.

4 - Referências

CRYPTOGRAPHY.IO. **RSA - cryptography documentation**. Disponível em: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/?highlight=rsa. Acesso em: 14 jun. 2021.

PYMOTW.COM. **Multicast - python**. Disponível em: https://pymotw.com/2/socket/multicast.html. Acesso em: 13 jun. 2021.

PYTHON.ORG.BR. **SocketBasico**. Disponível em: https://wiki.python.org.br/SocketBasico. Acesso em: 14 jun. 2021.

PYTHONTIC.COM. **Sendto()**. Disponível em: https://pythontic.com/modules/socket/sendto. Acesso em: 13 jun. 2021.