Relatório - Projeto: Programação Utilizando Sockets

Grupo: Eduardo Vinnicius(evdcn) e Rodrigo Rossiter(rrgf)

Como deveria ser executado

A ideia para executar nosso projeto era:

1. Realizar o pip install da biblioteca Crypto (pip install Crypto)
2. Executar o arquivo rede.py para inicialização da rede.
3. Executar o arquivo certificadora.py para inicialização da autoridade certificadora.
4. Executar o arquivo cliente.py seis vezes (uma para cada cliente), assim inicializando os nós da rede.

Organização do projeto

Primeiramente, optamos pelo desenvolvimento do projeto com base na primeira abordagem: a convencional sem o uso de docker. Agora vamos partir para como organizamos ele.

Para a criação do projeto, criamos 3 arquivos, sendo 2 deles responsáveis pela criação de objetos e inicialização da rede e autoridade certificadora, e 1 deles para a criação do objeto Cliente e execução de cada nó da rede. Segue os arquivos e uma breve explicação sobre:

- **rede.py**
  Este arquivo descreve a classe Rede, que representa uma rede de comunicação entre clientes através de sockets UDP. Ela inclui métodos para inserção de novos clientes na rede, roteamento de mensagens entre clientes e métodos auxiliares para comunicação e gerenciamento da rede.:

  - **Atributos**
    - **inicio:** Referência ao primeiro nó da rede.
    - **ultimo:** Referência ao último nó da rede.
    - **cont:** Contador para controle de clientes.
    - **clientes:** Lista de tuplas contendo os endereços IP e portas dos clientes na rede.
    - **endereco:** Tupla contendo o endereço IP e porta do socket.
    - **socket:** Objeto de socket para comunicação.
    - **roteamento:** Matriz de roteamento para determinar o próximo nó para encaminhamento de mensagens.

- **Métodos**
  - **insert(self, id, endereco):** Insere um novo cliente na rede e envia as informações de conexão para os clientes adjacentes.
  - **routing(self, emissor, remetente):** Roteia uma mensagem do emissor para o remetente, encaminhando-a através dos nós intermediários até o destinatário final.
  - **getClient(self, id_cliente):** Obtém o endereço de um cliente específico na rede a partir do id.
  - **listen(self):** Inicia a escuta do servidor para receber mensagens e realizar as operações correspondentes, como inserção de novos clientes, solicitações de conexão e roteamento de mensagens.

➢ **certificadora.py**

Responsável pela definição da classe Certificadora, que é responsável por gerenciar as chaves públicas dos clientes na rede e fornecer certificados de chave pública quando solicitados. Ela opera através de sockets UDP para comunicação e utiliza a biblioteca RSA para lidar com criptografia assimétrica.

- **Atributos:**
  - **chaves_pub:** Dicionário que mapeia o ID do nó (calculado a partir do endereço IP e porta) para sua chave pública correspondente.
  - **endereco:** Tupla contendo o endereço IP e porta do servidor.
  - **socket:** Socket UDP para comunicação de rede.

- **Métodos:**
  - **listen(self):** Inicia a escuta do servidor para receber mensagens e realizar as operações correspondentes, como o recebimento de novas chaves públicas e o envio de certificados de chave pública quando solicitado.

➢ **cliente.py**

Responsável pela definição do objeto Cliente, que é uma classe que representa um cliente em uma rede. Ele é utilizado para estabelecer conexões, trocar chaves de criptografia, enviar e receber mensagens criptografadas em uma rede de comunicação. Abaixo estão os principais métodos e atributos da classe:

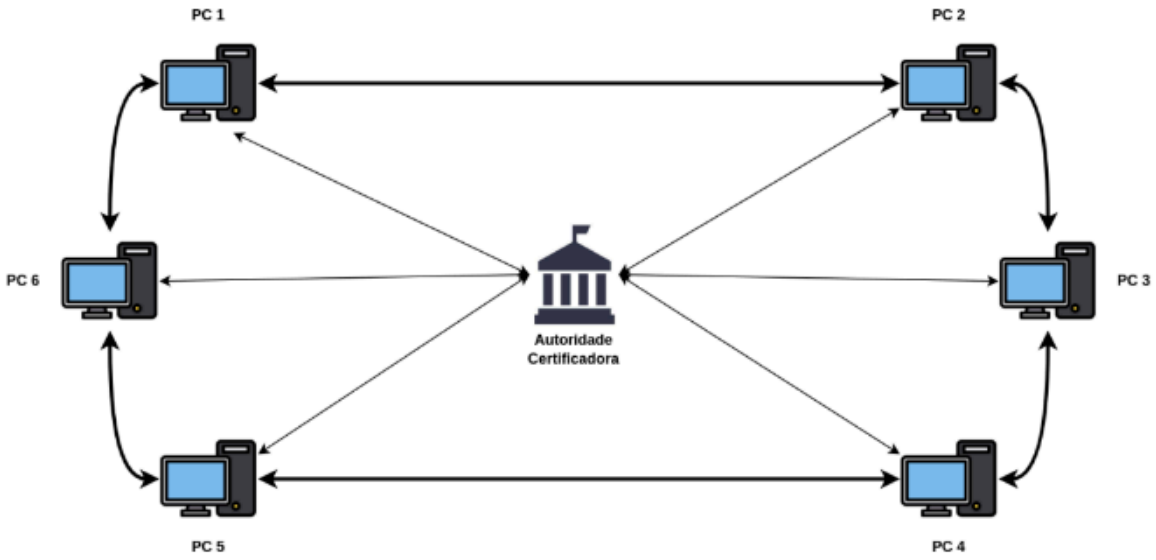
- **Atributos:**
  - **rede_endereco:** O endereço do socket da rede ao qual o cliente se conecta.

        - **certificadora_endereco:** O endereço do socket da certificadora ao qual o cliente envia suas solicitações de certificação.
        - **socket:** Um objeto de socket para comunicação com a rede, certificadora e outros clientes.
        - **socket_id:** Um objeto de socket de comunicação exclusiva para obter o ID do cliente.
        - **id:** O ID atribuído ao cliente.
        - **endereco:** O endereço IP e a porta do cliente para comunicação.
        - **prox e ant:** As referências para o próximo e o anterior cliente na rede.
        - **chave_priv:** A chave privada RSA do cliente.
        - **chave_sim:** Um dicionário para armazenar as chaves simétricas AES compartilhadas com outros clientes.
        - **conexao:** Um dicionário para rastrear as conexões estabelecidas com outros clientes.

    - **Métodos**
        - **create_socket(self):** Cria um socket para recebimento do id e o fecha após isso, e cria outro socket associado a um endereço IP e porta definitivo.
        - **insert(self):** Solicita e recebe informações sobre os clientes adjacentes na rede.
        - **certificate(self):** Gera chaves assimétricas para o cliente e envia sua chave pública para uma autoridade certificadora.
        - **connect(id):** Solicita o endereço de um cliente com o qual deseja se conectar.
        - **routing(endereco, msg):** trabalha em conjunto com o routing() da rede, enviando a ela o endereço do remetente e a mensagem.
        - **send(self, dupla):** Inicia a troca de mensagens com outro cliente na rede, incluindo solicitação de chaves, troca de chaves e envio de mensagens criptografadas.
        - **rcv(self, dupla):** Recebe mensagens de outro cliente na rede, incluindo a troca de chaves e mensagens criptografadas.
        - **key_exchange(self, certificadora, dupla):** Estabelece e troca chaves simétricas com outro cliente na rede.
        - **send_key(self, dupla):** Envia uma chave simétrica criptografada para outro cliente na rede.
        - **handle_key(self, dupla):** Lida com a recepção de uma chave simétrica criptografada de outro cliente na rede.
        - **ask_key(self, dupla, certificadora):** Solicita a troca de chaves simétricas com outro cliente, utilizando criptografia assimétrica.

- **rcv_msg(self, dupla):** Recebe mensagens criptografadas simetricamente de outro cliente na rede e realiza a descriptografia.
- **send_msg(self, dupla, rede):** Envia mensagens criptografadas simetricamente para outro cliente na rede.
- **broadcast(self):** Envia uma mensagem de broadcast para todos os clientes na rede.
- **confirm_broadcast(self):** Confirma a recepção de uma mensagem de broadcast por parte do cliente
- **rcv_confirmation(self):** Recebe uma confirmação de que a mensagem de broadcast foi recebida pelo outro cliente
- **create_node(self):** Cria socket do cliente, insere ele na rede e obtém certificado
- **main(self):** Função de execução do cliente na rede simulada

Primeira etapa: Topologia e Comunicação entre elementos



O que fizemos

Implementamos a comunicação entre os elementos da topologia conforme foi pedido nas especificações do projeto:

- Cada nó está sendo representado por um processo (cliente.py), o qual será executado seis vezes para a criação de seis nós com sockets distintos baseados em seu id. O id de cada cliente é determinado por um contador presente na rede, a qual sempre será incrementado após um novo nó ser adicionado.

- Configuramos o objeto cliente (cliente.py) de forma que ele só terá contato direto apenas com seus vizinhos (self.prox e self.ant) e com autoridade certificadora (certificadora.py).
- A rede constrói a topologia dos nós por meio de uma estrutura de lista duplamente ligada.

Problemas:

- Como fizemos uma estrutura de lista duplamente ligada, necessitamos do objeto cliente para termos as referências de nós anteriores e próximos. No primeiro momento, era funcional e de fácil compreensão, no entanto, ao termos que separar as classes em diferentes arquivos e não podermos referenciar os objetos no arquivo do cliente (visto que ao criar 6 Clientes, também seriam criadas 6 Redes e Certificadores), a comunicação foi feita inteiramente por sockets. Esses não permitem o envio e recebimento do objeto Cliente por conter um socket, de modo que não podíamos referenciar o próximo e anterior pelos nós, mas sim por seus endereços, o que criou um problema no roteamento mais adiante.

Segunda etapa: Autoridade Certificadora

O que fizemos

- Quando o nó é instanciado, a função ***certificate()*** é ativada, assim gerando chaves assimétricas para o cliente e envia sua chave pública à unidade certificadora
- Na geração de chaves utilizamos a biblioteca rsa, padrão do python
- Quanto à geração das chaves, colocamos o cliente para fazer essa operação, já que não faria sentido a autoridade certificadora criar a chave privada, pois apenas o cliente pode ter acesso a ela. Dessa forma, em nosso projeto, o cliente cria ambas as chaves (pública e privada) e envia a pública à autoridade certificadora.
- Para garantir a autenticidade do usuário que compartilhar as informações, será utilizada o método ***ask_key(), send_key()*** e ***handle_key()****, os quais se encarregarão de realizar a troca de chaves*
- Como forma de sinalização de solicitação de geração de chaves, fizemos com que o cliente escreva uma frase já definida: “oi, vamos trocar chaves”
- Além disso, utilizamos o AES para criptografia simétrica e RSA para assimétrica

Problemas

- Como já dito, foi necessário separar a certificadora em um socket próprio, o que nos fez mudar algumas lógicas dentro do código. Além disso, tivemos alguns problemas com a criptografia do RSA, o que atrasou a implementação.

Terceira etapa: Aplicação

O que fizemos

- Para a troca de chaves e o envio/recebimento das cinco mensagens criptografadas, tentamos resumir esse processo nas funções ***send()*** realizada pelo emissor das mensagens, e ***rcv()***, realizada pelo remetente.
- Para a realização do broadcast, criamos os métodos ***broadcast()*** e ***rcv_broadcast()***, os quais realizariam e confirmariam o recebimento do broadcast.
- A aplicação foi desenvolvida utilizando UDP, como é possível ver no método ***create_socket()***, onde realiza a criação de sockets 'SOCK_DGRAM'.
- Quanto à segurança, tentamos garanti-la por meio da troca de chaves simétricas.

Problemas

- Além da separação em diferentes arquivos, a comunicação se tornou o maior desafio na implementação, pois demoramos para encontrar uma lógica que permitisse que o cliente funcionasse como host para o recebimento de mensagens e suas respostas ao mesmo tempo que funcionasse como cliente para enviar as mensagens. Tentamos várias coisas e o que pareceu mais próximo de ter sentido (já que pela documentação, não era prevista a criação de um servidor intermediário ou de mais de um socket por cliente) foi a utilização de uma thread para o recebimento de mensagens enquanto um loop no fluxo principal envia as mensagens seguindo os métodos ***send()*** e ***rcv()***. Apesar disso, não tivemos tempo para testar completamente o funcionamento e verificar se era a abordagem correta.

Quarta etapa: Roteamento

O que fizemos

- Caso um usuário queira realizar uma solicitação de serviço com outro nó não adjacente, será utilizado a função *routing()*, que nada mais é do que nosso mecanismo de roteamento.
- O mecanismo de roteamento basicamente informa se a informação deve percorrer pelo sentido horário ou anti horário até encontrar o remetente, visto que o formato da rede é anelar.
- Para isso, criamos uma tabela de roteamento estática como atributo da rede. O funcionamento da tabela é da seguinte forma: o índice da linha da tabela corresponde ao emissor e o da coluna o remetente. Assim, o resultado[emissor][remetente] retornará ‘prox’ caso seja mais rápido ir pelo sentido horário e ‘ant’ se for melhor anti horário. Se por acaso for igual a distância de ambos os sentidos, ele irá pelo horário.

Visualização da tabela de roteamento

	PC 0	PC 1	PC 2	PC 3	PC 4	PC 5
PC 0	0	prox	prox	prox	ant	ant
PC 1	ant	0	prox	prox	prox	ant
PC 2	ant	ant	0	prox	prox	prox
PC 3	prox	ant	ant	0	prox	prox
PC 4	prox	prox	ant	ant	0	prox
PC 5	prox	prox	prox	ant	ant	0

- Funcionamento da função de roteamento: A função recebe como parâmetros o emissor e o remetente, e a partir disso, identifica se o caminho da mensagem deve ser feito no sentido horário ou anti horário. Depois, o emissor envia ela para seu vizinho que está no sentido determinado. Por fim, a função é chamada novamente de forma recursiva colocando o vizinho que recebeu a mensagem como o novo emissor. Essa função ficará rodando de forma recursiva até a mensagem chegar ao remetente

Problemas

- Como citado anteriormente, o roteamento se tornou um problema no momento em que tivemos de alterar a lógica de comunicação, já que não tínhamos o objeto e seria necessário rotear com apenas os endereços e utilizando uma tabela construída manualmente, de modo que não conseguimos encontrar uma abordagem satisfatória até o fim do prazo.

Conclusão do Projeto

- Nesse sentido, dada a lógica construída e a implementação, fica claro que, ainda que o projeto não tenha terminado 100% funcional, entendemos os processos e estruturas necessárias para a comunicação em uma rede, além de perceber-se que houveram várias tentativas e mudanças, levando em conta todo o histórico de commits no Github. Fato que se torna ainda mais perceptível por termos apenas 2 integrantes no fim das contas, o que demonstra o entendimento e comprometimento com o projeto.