Relatório Final - Protocolos de Comunicação

Victor Miguel de Morais Costa (vmmc2)
Zilde Souto Maior Neto (zsmn)

1 Introdução

- O presente documento tem como objetivo: relatar, de forma detalhada, o processo de desenvolvimento da biblioteca que implementa protocolo a ser usado no projeto final da disciplina.

2 Objetivo

- O protocolo foi elaborado com o objetivo de ser utilizado em um sistema de votação seguro em uma rede Wi-Fi não-confiável.

3 Decisões Iniciais de Implementação

- Para o desenvolvimento da biblioteca responsável por implementar o protocolo em questão, foram tomadas algumas decisões técnicas e conceituais importantes:
  - A biblioteca que implementa o protocolo foi desenvolvida na linguagem Python por questões de simplicidade e familiaridade dos integrantes.
  - A biblioteca foi desenvolvida para sistemas operacionais Linux.
  - Além disso, o protocolo, implementado pela biblioteca, foi desenvolvido segundo a arquitetura Cliente-Servidor.
  - Foi escolhido o TCP (Transmission Control Protocol) como protocolo da camada de transporte devido ao fato desse protocolo fornecer: confiabilidade, entrega ordenada, controle de fluxo e controle de congestionamento.
  - Por fim, o protocolo elaborado funciona em 3 etapas sequenciais e distintas (”Handshake”, ”Generate Symmetric Key”e ”Vote Session”) que serão descritas a seguir.

4 Etapas do Protocolo

4.1 Handshake

- No início da primeira etapa, o **Client** enviará um pacote chamado **auth**, que tem como parâmetros a chave pública do *Client clientPublicKey* e um valor de nonce gerado pelo *Client* e encriptado pela chave pública do *Server*.
- O **Server** enviará um pacote chamado **msg**, que tem como único parâmetro o valor de *nonce* encriptado desta vez com a chave pública do *Client*.

- O **Client**, após verificar o valor de *nonce* recebido, enviará um pacote **msg** contendo um único parâmetro que pode assumir dois valores: "ack", caso o valor de *nonce* recebido coincida com o valor de *nonce* previamento gerado, ou "nack", caso contrário.

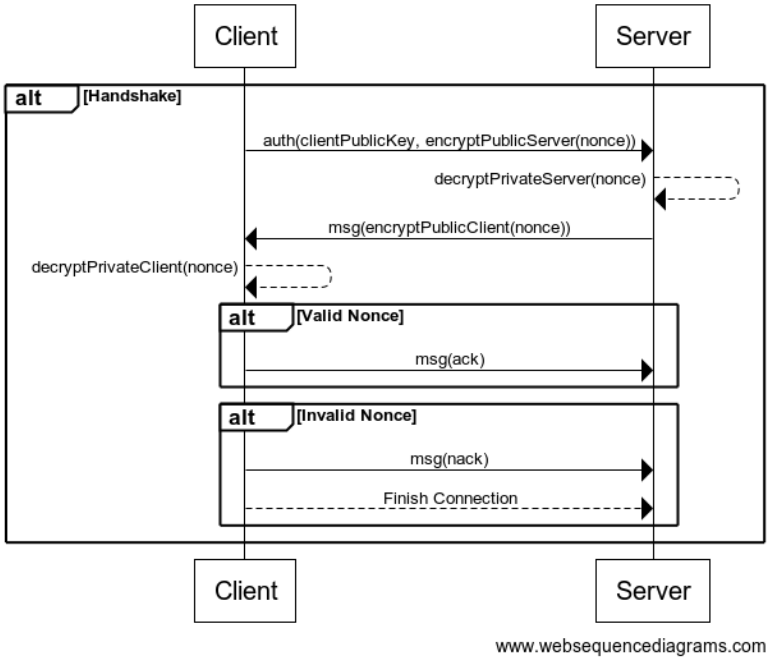

Figura 1: Etapa de Handshake

4.2 Generate Symmetric Key

- A segunda etapa é iniciada pelo **Server**, que gera a chave simétrica, a ser compartilhada com o **Client** na conexão individual. Após gerar sua chave simétrica, o **Server** enviará um pacote ao **Client** chamado **msg**. Tal pacote possui um único parâmetro cujo valor é chave simétrica previamente encriptada pela chave pública do **Client**. Essa encriptação é feita por meio da função *encryptPublicClient(symmetricKey)*.

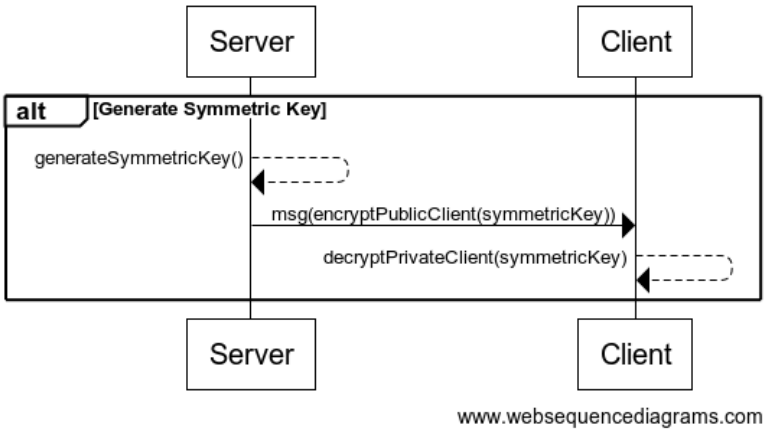

Figura 2: Etapa de Generate Symmetric Key

4.3 Vote Session

- Dentro da terceira etapa, existe um loop, no qual o **Client** enviará um pacote chamado **vote**, que possui um único parâmetro cujo valor é a pessoa votada, previamente encriptada através da chave simétrica (isso é feito pela função *encryptSymmetric*).
- Após as validações do voto (nas etapas de desencriptação), se ele for considerado um voto válido é enviado um pacote **msg** do **Server** para o **Client** que possui um único parâmetro que é uma mensagem *ack* encriptada através da função *encryptSymmetric*.
- Em caso contrário (voto inválido), é enviado um pacote **msg** do **Server** para o **Client** que possui um único parãmetro que é uma mensagem *nack* encriptada através da função *encryptSymmetric*.

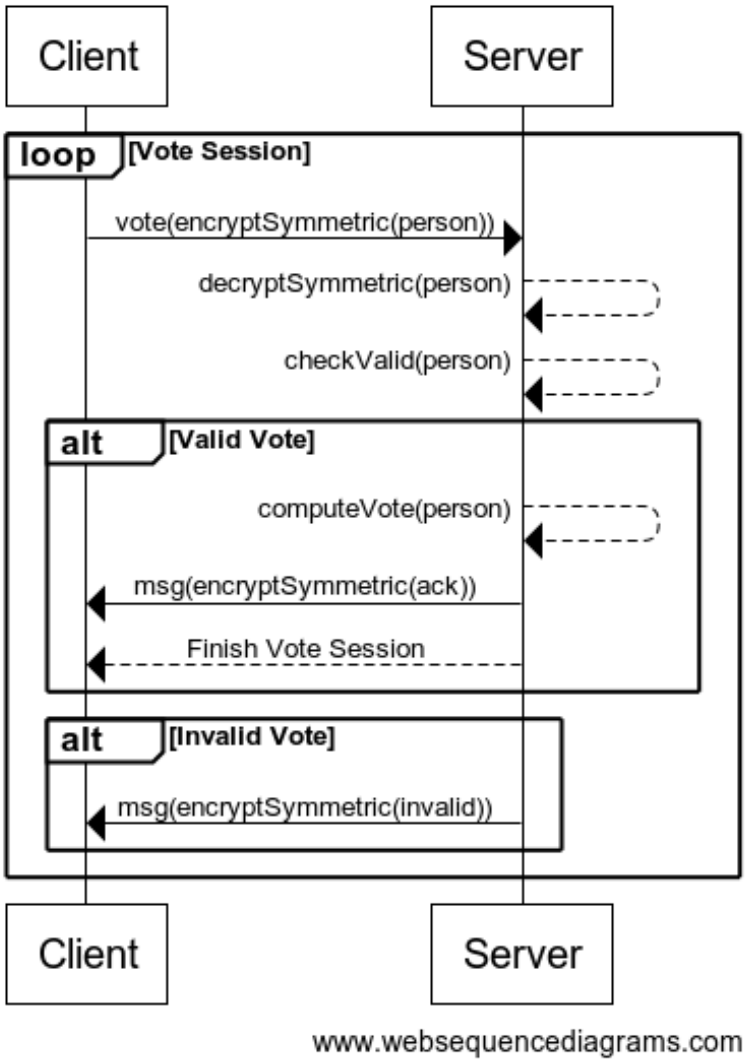

Figura 3: Etapa de Vote Session

5 Garantia dos Requisitos de Segurança

- A presente seção tem como objetivo descrever como a autenticidade e a confidencialidade são garantidas através do protocolo desenvolvido.

5.1 Autenticidade

- Pensamos em garantir apenas a autenticidade do **Server**, já que ele é, basicamente, o responsável pelo processo de votação.
- Para que esse processo fosse possível, assumimos que a *public key* do **Server** estará disponível com antecedência para todos os **Clients** na rede. Dessa forma, inicialmente, o **Client** envia um pacote para o **Server** iniciando a comunicação, no qual os parâmetros enviados são a *public key* do **Client**, que está se comunicando, e um *nonce* (string aleatória), que é criptografado com a *public key* do **Server**.
- É esperado, portanto, que o **Server** possua a *private key* e que, com ela, possa desencriptar o pacote contendo o *nonce* e obter seu valor.
- Para finalmente estabelecer a conexão, é necessário que o **Server** envie para o **Client** uma mensagem contendo o mesmo *nonce* recebido só que desta vez encriptado com a *public key* do **Client**, que foi obtida no primeiro pacote da conversação.
- O **Client** então irá realizar o mesmo processo de desencriptação com a sua *private key* e, se o valor do *nonce* desencriptado corresponder ao valor do *nonce* enviado, assumimos então que o **Server** é autêntico e prosseguimos com a conexão.

5.2 Confidencialidade

- Após a etapa de *handshake*, o **Server** irá gerar uma chave simétrica por meio da função *generateSymmetricKey* e enviará um pacote **msg**, que contém essa chave simétrica encriptada pela *public key* do **Client**.
- Recebido o pacote, o **Client** irá desencriptá-lo com sua *private key* e obterá a chave simétrica. Feito isso, o **Client** iniciará a etapa de *Vote Session* enviando o pacote **vote**, que contém a sua escolha de voto encriptada pela chave simétrica.
- Perceba que, a chave simétrica é primeiro encriptada pela *public key* do **Client** antes de ser enviada. Sendo assim, ela só pode ser descoberta por meio de uma desencriptação usando a *private key* desse mesmo **Client**. Como apenas o **Client** em si possui acesso a sua *private key*, não é possível que um atacante consiga descobrir nem a *private key* do **Client** e nem a chave simétrica, garantindo assim a confidencialidade do voto.
- Após isso, é feita no **Server** uma etapa de validação do voto e independente do voto ser válido ou não, é enviado um pacote **msg** contendo o status do voto (válido ou inválido) encriptado pela chave simétrica compartilhada anteriormente. Isso assegura que um possível atacante não será capaz de saber se um voto foi de fato computado ou não.

6 A Biblioteca

- Para que a biblioteca desenvolvida possa ser utilizada em uma aplicação qualquer, é necessário adicionar o arquivo "protocol.py"ao projeto da aplicação desenvolvida em Python.
- Para fazer uso da biblioteca, deve-se adicionar a linha: `import protocol` no início de cada arquivo ".py"que fará uso das funcionalidades da biblioteca.
- Como a biblioteca desenvolvida faz uso de outros pacotes de terceiros, é necessário instalar alguns deles por meio do terminal, como será explicado abaixo.

7 Dependências

- A biblioteca desenvolvida faz uso de outros pacotes, necessários para auxiliar na implementação de certas funcionalidades da biblioteca. Seu uso se faz necessário principalmente na criptografia presente no protocolo desenvolvido. Os pacotes utilizados foram:

  1. **os**
  2. **socket**
  3. **ast**
  4. **random**
  5. **collections**
  6. **base64**
  7. **Crypto** (Esse pacote específico deve ser instalado por meio do comando: "pip3 install pycryptodome" no terminal). A documentação desse pacote em particular pode ser encontrada em: https://pycryptodome.readthedocs.io/en/latest/index.html

8 Estrutura

- A biblioteca desenvolvida foi estruturada de forma que possui 5 classes distintas, que implementam, de forma modularizada, as características do protocolo previamente descritas.
- As classes em questão são: **bcolors**, **Socket**, **Server**, **Client** e **Protocol**.
- Por questões de organização, a estrutura foi pensada de forma hierárquica, de maneira que existem superclasses e subclasses na biblioteca. A hierarquia mencionada pode ser vista abaixo:

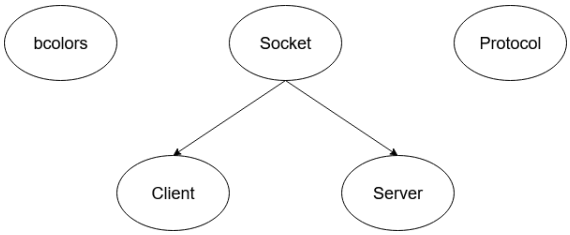

Figura 4: Hierarquia da Biblioteca

- Iniciaremos agora a descrição de cada uma das classes, passando por seu objetivo, seus atributos e seus métodos.

8.1 bcolors

8.1.1 Objetivo

– A classe tem como objetivo agrupar uma série de códigos de cores e de modificadores de texto, de forma que as mensagens geradas pela biblioteca possam ser visualizadas mais facilmente no terminal pelos usuários, caso seja necessário.

8.1.2 Atributos

- **bcolors.HEADER:** Atributo responsável por guardar o código responsável por realizar a alteração no texto presente no terminal para o formato cabeçalho/header. `Tipo: str.`
- **bcolors.OKBLUE:** Atributo responsável por guardar o código referente à cor Azul. O atributo é utilizado para emitir mensagens de sucesso pela biblioteca. `Tipo: str.`
- **bcolors.OKCYAN:** Atributo responsável por guardar o código referente à cor Ciano. O atributo é utilizado para emitir mensagens de sucesso pela biblioteca. `Tipo: str.`
- **bcolors.OKGREEN:** Atributo responsável por guardar o código referente à cor Verde. O atributo é utilizado para emitir mensagens de sucesso pela biblioteca. `Tipo: str.`
- **bcolors.WARNING:** Atributo responsável por guardar o código utilizado quando é preciso emitir uma mensagem de aviso/warning. `Tipo: str.`
- **bcolors.FAIL:** Atributo responsável por guardar o código referente à cor Vermelho. O atributo é utilizado para emitir mensagens de erro/fail pela biblioteca. `Tipo: str.`
- **bcolors.ENDC:** Atributo responsável por guardar o código que indica que não serão usados outros atributos para modificar a mesma string em questão. Ele sempre é utilizado após o final da string. `Tipo: str.`
- **bcolors.BOLD:** Atributo responsável por guardar o código responsável por realizar a alteração no texto presente no terminal para o formato negrito/bold. `Tipo: str.`
- **bcolors.UNDERLINE:** Atributo responsável por guardar o código responsável por realizar a alteração no texto presente no terminal para o formato sublinhado/underline. `Tipo: str.`

8.1.3 Métodos

- Não há métodos nessa classe.

8.2 Socket

8.2.1 Objetivo

- A classe tem como objetivo não só criar um socket para que a comunicação entre cliente e servidor ocorra, mas também implementar algumas funcionalidades de criptografia necessárias para o funcionamento adequado do protocolo desenvolvido.

8.2.2 Atributos

- **Socket.socket:** Esse atributo é responsável por guardar o objeto `socket.socket`, criado no construtor dessa classe e presente no módulo "socket". `Tipo: socket.socket.`
- **Socket.address:** Esse atributo é responsável por armazenar o endereço IP referente ao hospedeiro em questão. `Tipo: str.`
- **Socket.port:** Esse atributo é responsável por armazenar o número de porta referente ao socket criado no hospedeiro em questão. `Tipo: str.`
- **Socket.publicKey:** Esse atributo é responsável por armazenar a chave pública do socket no hospedeiro em questão. `Tipo: Crypto.PublicKey.RSA.RsaKey.`
- **Socket.privateKey:** Esse atributo é responsável por armazenar a chave privada do socket no hospedeiro em questão. `Tipo: Crypto.PublicKey.RSA.RsaKey.`

8.2.3 Métodos

- **Socket(address, port):** Esse método é o construtor da classe. O método possui como parâmetros: address (O endereço IP do hospedeiro no qual o socket foi criado. `Tipo: str`) e port (O número de porta a ser usado pelo socket. `Tipo: str`). O método retorna um objeto do `tipo: protocol.Socket`.
- **Socket.connect():** Esse método é responsável por realizar a conexão de um hospedeiro com outro. Uma vez que a forma como a conexão é executada varia de cliente para servidor, essa implementação está vazia e a implementação de fato desse método está presente nas subclasses da classe "Socket": classe "Client" e classe "Server". Em outras palavras, esse método funciona como um método abstrato. O método não possui nenhum parâmetro e não retorna nada.
- **Socket.generateAsymmetricKeys():** Esse método é responsável por gerar as chaves assimétricas (chave privada e chave pública) do socket no hospedeiro em questão. Tanto a chave privada como a chave pública são geradas com o auxílio do pacote "Crypto" mencionado anteriormente. Além disso, é executado automaticamente quando o construtor da classe "Socket" é chamado. O método não possui nenhum parâmetro e não retorna nada.
- **Socket.encryptMessage(message, key, encryptType="asymmetric"):** Esse método é responsável por realizar a encriptação de uma mensagem fornecida, dada uma chave e um modo de encriptação. Perceba que o último parâmetro é opcional e, por padrão, será usado o modo de encriptação "asymmetric". O método possui os seguintes parâmetros: message (A mensagem a ser encriptada. `Tipo: bytearray`), key (A chave a ser usada na encriptação. `Tipo: Crypto.PublicKey.RSA.RsaKey`) e encryptType (O modo de encriptação a ser usado. `Tipo: str`. Esse parâmetro pode possuir apenas dois valores: "symmetric" ou "asymmetric"). O método pode retornar dois tipos de objetos dependendo de qual modo de encriptação foi usado. Se foi usado o modo de encriptação "asymmetric", o método retorna um objeto `tipo: bytes`. Por outro lado, se foi usado o modo de encriptação "symmetric", o método retorna um objeto `tipo: dict`.
- **Socket.decryptMessage(message, key, encryptType="asymmetric", iv=""):** Esse método é responsável por realizar a desencriptação de uma mensagem fornecida, dada uma chave, um modo de encriptação e um iv/vetor de inicialização. Perceba que os dois últimos parâmetros são opcionais e, por padrão, será utilizado o modo de encriptação "asymmetric" e o iv terá o valor: "". O método possui os seguintes parâmetros: message (A mensagem a ser encriptada. `Tipo: bytearray`), key (A chave a ser usada na desencriptação. `Tipo: Crypto.PublicKey.RSA.RsaKey`), encryptType (O modo de desencriptação a ser usado. `Tipo: str`. Esse parâmetro pode possuir apenas dois valores: "symmetric" ou "asymmetric") e iv (O vetor de inicialização a ser usado, caso o modo de encriptação seja "symmetric". `Tipo: bytes`). O método retorna apenas um tipo de objeto, independente de qual modo de encriptação foi usado. O método retorna um objeto `tipo: bytes`.

8.3 Server

8.3.1 Objetivo

- A classe tem com objetivo representar o estado e o comportamento de um servidor, conforme apresentado previamente na descrição do protocolo. Como descrito em seções anteriores, o protocolo desenvolvido separa as funcionalidades do cliente e do servidor em classes distintas. Além disso, ambas as classes mencionadas são subclasses da classe "Socket"descrita acima.

8.3.2 Atributos

- Por ser uma subclasse da classe "Socket", a classe "Server"possui todos os atributos da classe "Socket"descritos previamente. Ademais, possui os seguintes atributos específicos da própria classe "Server":
- **Server.candidates:** Esse atributo é responsável por guardar a lista com todas as opções válidas de candidatos na votação. `Tipo: list`.
- **Server.votes:** Esse atributo é responsável por contabilizar os votos de todas as opções válidas de candidatos na votação. `tipo: dict`.

8.3.3 Métodos

- Por ser uma subclasse da classe "Socket", a classe "Server"possui todos os métodos da classe "Socket"descritos previamente. Ademais, possui os seguintes métodos específicos da própria classe "Server":
- **Server(address, port, candidates):** : Esse método é o construtor da classe. Nesse construtor, ocorre a chamada do método responsável por gerar as chaves assimétricas (chave pública e chave privada) e, além disso, a chave pública do servidor será armazenada em um arquivo do tipo "txt"cujo nome é: "serverPublicKey.txt". O método possui como parâmetros: address (O endereço IP do servidor. `Tipo: str`), port (O número de porta a ser usado pelo socket do servidor. `Tipo: str`) e candidates (A lista de opções válidas de candidatos na eleição. `Tipo: list`). O método retorna um objeto do `tipo: protocol.Server`.
- **Server.connect():** : Esse método é responsável por criar a ligação do socket do servidor com um endereço e porta. Em seguida, ele faz com que o socket do servidor "escute" por novas conexões. O método não possui nenhum parâmetro e não retorna nada.
- **Server.sendMessage(clientSocket, message):** Esse método é responsável por enviar uma mensagem a um cliente específico. O método possui como parâmetros: clientSocket (O socket do cliente para o qual a mensagem proveniente do servidor será enviada. `Tipo: socket.socket`) e message (A mensagem que será enviada para o socket do cliente em questão. `Tipo: bytes`). O método não retorna nada.
- **Server.checkValidCandidate(candidate):** Esse método é responsável por verificar se um candidato recebido é, de fato, uma opção válida de voto. O método possui um único parâmetro: candidate (Um nome representando o candidato votado. `Tipo: str`). O método retorna um booleano (`tipo: bool`), indicando se a opção fornecida é válida ou não.
- **Server.addVote(candidate):** Esse método é responsável por computar o voto referente a uma opção de candidato, que já foi verificada e apontada como válida. O método possui como único parâmetro: candidate (Um nome representando o candidato votado. `Tipo: str`). O método não retorna nada.
- **Server.printVotes():** Esse método é responsável por imprimir na tela do terminal todas as opções válidas de candidatos e a quantidade respectiva dos seus votos computados até o momento da chamada do método. Esse método não possui nenhum parâmetro e não retorna nada.

8.4 Client

8.4.1 Objetivo

- A classe tem com objetivo representar o estado e o comportamento do cliente, conforme apresentado previamente na descrição do protocolo. Como descrito em seções

anteriores, o protocolo desenvolvido separa as funcionalidades do cliente e do servidor em classes distintas. Além disso, ambas as classes mencionadas são subclasses da classe "Socket"descrita anteriormente.

8.4.2 Atributos

– Por ser uma subclasse da classe "Socket", a classe "Client"possui todos os atributos da classe "Socket"descritos previamente. No entanto, a classe "Client"não possui atributos específicos próprios dela mesma.

8.4.3 Métodos

– Por ser uma subclasse da classe "Socket", a classe "Client"possui todos os métodos da classe "Socket"descritos previamente. Ademais, possui os seguintes métodos específicos da própria classe "Client":

– **Client(address, port):** Esse método é o construtor da classe. O método possui como parâmetros: address (O endereço IP do servidor ao qual o cliente deve se conectar. `Tipo: str`) e port (O número de porta utilizado pelo socket do servidor ao qual o cliente irá se conectar. `Tipo: str`). O método retorna um objeto do `tipo: protocol.Client`.

– **Client.connect():** Esse método é responsável por criar uma conexão entre o cliente em questão e o (socket do) servidor. O método não possui nenhum parâmetro e não retorna nada.

– **Client.sendMessage(message):** Esse método é responsável por enviar uma mensagem do cliente para o servidor. O método possui um único parâmetro: message (A mensagem que será enviada do cliente para o servidor. `Tipo: bytes`). O método não retorna nada.

8.5 Protocol

8.5.1 Objetivo

– A classe tem como objetivo implementar, de forma sequencial e modularizada, as etapas do protocolo descritas na entrega anterior (etapa de "Handshake", etapa de geração de chave simétrica e etapa de sessão de votação). Para isso, como será explicado adiante, existe uma divisão entre as atividades executadas pelo cliente em todas as etapas e as atividades executadas pelo servidor em todas as etapas.

8.5.2 Atributos

– **Protocol.symmetricKey:** Esse atributo é responsável por armazenar a chave simétrica, que será compartilhada entre a conexão individual cliente-servidor. `Tipo: bytes`.

8.5.3 Métodos

– **Protocol():** Esse método é o construtor da classe. O método não possui nenhum parâmetro. O método retorna um objeto do `tipo: protocol.Protocol`.

– **Protocol.generateRandomKey():** Esse método é responsável pela geração, de forma aleatória, da chave simétrica que será compartilhada na conexão individual cliente-servidor. O método não possui nenhum parâmetro e retorna um objeto do `tipo: bytes`.

- **Protocol.clientProcedure(client, serverPublicKey):** Esse método é responsável por executar, de forma sequencial, todas as atividades atribuídas ao cliente nas duas primeiras etapas do protocolo definidas anteriormente. Ou seja, o método executa as funções do cliente nas etapas de: "Handshake"e geração de chave simétrica, respectivamente. O método possui como parâmetros: client (O cliente que está interagindo/conectado com o servidor. `Tipo: protocol.Client`) e serverPublicKey (A chave pública do servidor, que é assumida ser de conhecimento geral. `tipo: Crypto.PublicKey.RSA.RsaKey`). O método não retorna nada.
- **Protocol.clientVote(client, votedPerson, serverPublicKey):** Esse método é responsável por executar as atividades atribuídas ao cliente na última etapa do protocolo (etapa de sessão de votação). O método possui como parâmetros: client (O cliente que está interagindo/conectado com o servidor. `Tipo: protocol.Client`), votedPerson (Um nome indicando o candidato votado. `Tipo: str`) e serverPublicKey (A chave pública do servidor, que é assumida ser de conhecimento geral. `tipo: Crypto.PublicKey.RSA.RsaKey`). O método não retorna nada.
- **Protocol.serverProcedure(server, clientSocket, clientAddress):** Esse método é responsável por executar, de forma sequencial, todas as atividades atribuídas ao servidor nas etapas do protocolo definidas anteriormente. Ou seja, o método executa as funções do servidor nas etapas de: "Handshake", geração de chave simétrica e sessão de votação, respectivamente. O método possui como parâmetros: server (O servidor que está executando o seu procedimento no protocolo. `Tipo: protocol.Server`), clientSocket (O socket do cliente com o qual o servidor em questão está conectado. `Tipo: socket.socket`) e clientAddress (Uma tupla contendo o endereço IP do cliente e o número de porta utilizado pelo socket desse mesmo cliente ao qual o servidor está conectado. `Tipo: tuple`). O método não retorna nada.