Guião 06

1.2

No código-fonte do servidor, ele está sempre a aguardar uma mensagem de um usuário no endereço IPv4 0.0.0.0, na porta UDP 5005.

No código-fonte do cliente, há duas funções para interação com o usuário, e o seu endereço IPv4 é 127.0.0.1, usando a mesma porta UDP que o servidor. Em seguida, ele aguarda uma mensagem do usuário para enviá-la ao servidor usando esse endereço IP e porta, para que o servidor receba a mensagem.

No Wireshark, para cada mensagem, podemos capturar um pacote UDP que contém as informações da mensagem (tamanho, conteúdo) e também informações sobre as portas.

2.2

Após analisar o código fonte do servidor, definimos o IP e a porta do servidor (igual ao anterior). Depois disso, o socket é criado e começa a escutar os clientes recebidos (o máximo é 5). Então, depois que um cliente se conecta, ele armazena o endereço e o socket do cliente conectado. Agora ele cria um thread para ouvir mensagens, ao mesmo tempo que pode aceitar mais clientes. Ao receber uma mensagem de um cliente, ecoa a mensagem de volta para ele. No cliente, ele define seu IP e porta (como antes), e cria um soquete, que então se conecta ao servidor. Em seguida, ele espera que uma mensagem seja enviada ao servidor usando aquele endereço IP e porta, para que o servidor receba a mensagem (que está constantemente escutando). Finalmente, antes de permitir que o usuário envie outra mensagem, aguarda que o servidor faça eco da mensagem enviada. Em ambas as extremidades existe um manipulador de exceções, caso um deles se desconecte, ele limpa o soquete utilizado e permite que o servidor escute um novo cliente.

3.2

A diferença entre este e o servidor anterior, é que agora todos os clientes são tratados por uma única thread, utilizando o módulo `seletores`.

Após cadastrar um cliente, ele irá procurar por atividades/eventos (que neste caso serão mensagens), para então acionar a função `handle\_data`.

A função `handle\_data` funciona de forma semelhante à função `handle\_client\_connection` no arquivo anterior. Ele espera por uma mensagem e a envia de volta, e se a conexão for fechada ou perdida, ele fecha o socket.

Cada cliente possui uma chave, fornecida pelo seletor, que é impressa toda vez que ele estabelece uma conexão, recebe dados ou se desconecta.

4.2

Tanto no servidor quanto no cliente, existem algumas alterações. Agora, em vez de enviar uma mensagem simples, o cliente envia uma mensagem estruturada usando o módulo python `struct`. Esta ainda é uma mensagem normal, mas definimos como ela deverá ser decodificada posteriormente.

Se tomarmos o exemplo deste código, a struct `'!BLL20s'` representa uma struct que contém o seguinte:

Desta forma conseguimos enviar não só a mensagem, mas também estamos enviando a versão da mensagem, o número do pedido do cliente e o tamanho da mensagem, sendo este importante para o servidor descompactar a struct já que a mensagem é um matriz de caracteres (byte) de tamanho variável e o servidor precisa saber quantos caracteres descompactar.

No pacote TCP, ainda vemos todos os dados compactados em seu campo Dados.

5.2

Neste caso, o pacote ainda é semelhante ao anterior.

A diferença agora é que, em vez de enviar a estrutura de uma vez, primeiro enviamos o cabeçalho da estrutura. Este cabeçalho contém a versão, ordem e tamanho (os primeiros elementos das estruturas usadas anteriormente). O servidor então solicita o número de bytes passados ​​como tamanho. Desta forma, agora podemos enviar uma mensagem com o tamanho que quisermos

2.2

Após analisar o código-fonte do servidor, definimos o IP e a porta do servidor (iguais aos anteriores). Depois disso, o socket é criado e começa a ouvir clientes que tentam se conectar (máximo de 5).  
Quando um cliente se conecta, o servidor armazena o endereço e o socket do cliente conectado.  
Agora, ele cria uma *thread* para ouvir mensagens, enquanto ainda pode aceitar mais clientes.  
Ao receber uma mensagem de um cliente, ele devolve a mensagem (faz o *echo*) para o cliente.

No cliente, ele define seu IP e porta (iguais aos anteriores) e cria um socket, que se conecta ao servidor. Em seguida, ele aguarda uma mensagem para enviá-la ao servidor usando o endereço IP e porta, para que o servidor receba a mensagem (que está constantemente ouvindo). Por fim, antes de permitir que o usuário envie outra mensagem, aguarda que o servidor faça o *echo* da mensagem enviada.

No Wireshark, capturamos 3 pacotes TCP quando o cliente se conecta ao servidor:

1. Primeiro, há uma sincronização (SYN) do cliente para o servidor solicitando conexão.
2. Depois, o servidor envia um pacote de sincronização/acknowledge (SYN/ACK) para confirmar a conexão e reconhecer as informações do cliente.
3. Por fim, o cliente envia um pacote de confirmação (ACK) de volta ao servidor para confirmar que recebeu as informações.

Capturamos mais 4 pacotes TCP por troca de mensagem na seguinte ordem:

1. Um pacote push/acknowledge do cliente para o servidor, que inclui as informações da mensagem e solicita que o servidor confirme o recebimento.
2. Um pacote acknowledge do servidor para o cliente informando que a mensagem foi recebida corretamente.
3. Um pacote push/acknowledge do servidor para o cliente devolvendo a mensagem enviada e solicitando que o cliente confirme o recebimento.
4. Um pacote acknowledge do cliente para o servidor confirmando que recebeu a mensagem de volta.