CAMADA DE TRANSPORTE

NESTA PRÁTICA, vamos implementar um protótipo de um protocolo compatível com o TCP. Focaremos na ponta do servidor. Este roteiro te guiará em passos para que você construa aos poucos uma implementação completa.

Como todos os grupos entregaram a Etapa 1 em Python, vamos assumir que vocês vão continuar usando a mesma linguagem. Caso decida usar outra linguagem, você terá de adaptar também os testes.

Sua implementação deve ser realizada no arquivo `mytcp.py`, que já veio com um esqueleto em cima do qual você vai construir o seu código. Para ajudar na sua implementação, você pode chamar as funções e usar os valores que já vieram declarados no arquivo `mytcputils.py`. Pode ser útil, também, consultar a página sobre o formato do segmento TCP na Wikipedia.

Para testar seu código, execute `./autograde.py`. Cada um dos testes vai usar a sua implementação como uma biblioteca, verificando se ela apresenta o comportamento esperado.

Passo 1 — 1 ponto

Implemente o código necessário para aceitar uma conexão. O código base já veio com um `if` para verificar se chegou um segmento com a *flag* SYN. Como vimos no slide 80 da aula, isso significa que um cliente está querendo abrir uma nova conexão. Você deve responder ao SYN com um SYN+ACK para aceitar a conexão.

Os locais sugeridos para preencher seu código são no método `_rdt_rcv` da classe `Servidor`, ou então no construtor da classe `Conexao`.

Lembre-se que, como resposta, o cliente vai enviar um segmento contendo a *flag* ACK que, além disso, também já pode conter os primeiros dados enviados pelo cliente. Mas disso nós vamos tratar no próximo passo.

Passo 2 — 1 ponto

O recebimento de segmentos no TCP é similar ao GBN, descrito no slide 49 da aula. Ao tratar um segmento proveniente da camada de rede, assegure-se que ele não é duplicado e que foi recebido em ordem antes de repassá-lo para a camada de aplicação.

Por simplicidade, confirme cada segmento que for recebido corretamente mandando um segmento com a *flag* ACK e com *payload* vazio.

Preencha o método `_rdt_rcv` da classe `Conexao` para implementar essa funcionalidade.

Passo 3 — 1 ponto

O funcionamento básico do envio de segmentos pelo TCP é descrito no slide 67 da aula. Vamos quebrar a implementação em duas partes.

Nesta primeira parte, vamos nos preocupar somente com o evento "dados recebidos da camada de aplicação acima". Ou seja, mantenha uma contagem correta do número de sequência a ser inserido no segmento e construa o segmento corretamente, enviando-o em seguida para a camada de rede.

Preencha o método `enviar` da classe `Conexao` para implementar essa funcionalidade.

Por simplicidade, mantenha a *flag* de ACK sempre ligada nos segmentos que você enviar, incluindo corretamente o *acknowledgement number* correspondente ao próximo byte que você espera receber. Note que essa contagem refere-se ao funcionamento da nossa ponta como *receptor*, e não como *transmissor*, mas aqui misturamos os papéis.

Passo 4 — 1 ponto

Antes de prosseguirmos com a segunda parte da implementação da máquina de estados de envio, sugerimos dar uma pausa e tratar antes do fechamento de conexões. Assim, você terá um protótipo básico que poderá ser usado para conversar, por exemplo, com o `nc`, desde que você esteja em uma rede ideal (sem perdas, com banda sobrando, etc.)

O fechamento de conexões é descrito no slide 83 da aula. Uma das pontas escolhe fechar a conexão, enviando um segmento com *flag* `FIN`. A outra ponta confirma o recebimento do `FIN` mandando um ACK. Para que o encerramento da conexão ocorra de forma limpa, a outra ponta deve eventualmente fazer a mesma coisa (fechar a conexão enviando um `FIN`).

Por simplicidade, você pode assumir que o cliente é o primeiro a pedir para fechar a conexão.

Quando receber o pedido de fechamento, faça o *callback* da classe *Conexao* ser chamado passando como o argumento `dados` uma string de bytes vazia (`b''`), para ficar parecido com o comportamento da função `recv` (de *sockets*) que você usou na Etapa 1, além de responder corretamente com um ACK.

Quando o método `fechar` da classe *Conexao* for chamado, envie um segmento com *flag* `FIN`.

Se você implementou todos os passos até aqui, você deve ser capaz de executar o arquivo `exemplo_integracao.py` em um sistema Linux (leia antes os comentários no início do arquivo) e testá-lo conectando-se a ele com o comando `nc localhost 7000`.

Passo 5 — 2 pontos

Vamos, agora, voltar ao slide 67 da aula para implementar a segunda parte do transmissor. Faça a sua implementação ser capaz de lidar com perda de pacotes. Para isso, você vai precisar de um *timer*. Há um exemplo no código, mas ele está posicionado no local errado: você deve movê-lo para os locais do código onde realmente é necessário iniciar um *timer*. Por enquanto, você pode usar um valor constante (não use um valor muito grande!) para o intervalo do *timer*.

Além disso, você precisará tratar os ACKs que chegarem pelo método `_rdt_rcv` da classe `Conexao`.

Passo 6 — 2 pontos

Calcule o `TimeoutInterval` de acordo com as equações dos slides 62 e 63 e passe a usar esse valor como intervalo dos *timers* que você criar.

Para medir o `SampleRTT`, meça o tempo que se passa (por exemplo, com `time.time()`) entre você enviar um segmento e receber o ACK dele. Não leve em conta segmentos que estiverem sendo retransmitidos, apenas aqueles que estiverem sendo transmitidos pela primeira vez.

Quando você medir o primeiro `SampleRTT` de uma conexão, em vez de usar as equações, inicialize os valores atribuindo `EstimatedRTT` e `DevRTT` com `SampleRTT` e `SampleRTT/2`, respectivamente, como sugerido pela RFC 2988.

Passo 7 — 2 pontos

Implemente um mecanismo de controle de congestionamento simplificado similar ao AIMD (descrito no slide 96 da aula). Quando uma conexão for aberta, comece com uma janela de 1 MSS. Sempre que você receber ACK de uma janela inteira, aumente a janela de mais 1 MSS. Se ocorrer um *timeout* no *timer*, reduza a janela pela metade.

Opcional

O arquivo `exemplo_integracao.py` vem com um exemplo de camada de aplicação que faz eco, ou seja, envia de volta para o cliente tudo que receber dele.

Experimente adaptar seu código da Etapa 1 para executar nessa estrutura. Se você tiver escrito o código usando `select`, será necessário fazer uma pequena mudança de estrutura para usar funções de *callback* em vez de ficar esperando atividade nos *sockets* em um *loop*. O

código deve, na verdade, ficar mais simples, pois você já vai receber os dados diretamente como argumento e não precisará chamar `recv`.

Talvez você pegue erros de implementação que não tenham sido detectados pelos testes! Além disso, esta tarefa pode te ajudar a adiantar a etapa final de projeto caso você venha a optar por trabalhar no projeto do servidor.