TP2: sistema de mensagens

Prática de programação usando sockets

Data de entrega: verifique o calendário do curso
Conteúdo da entrega: `.tar.gz` ou `.zip` contendo um ou mais `.py` do código, um `.py` da topologia mininet e um `.pdf` do relatório

Objetivos e etapas

O objetivo deste trabalho é especificar e implementar um protocolo em nível de aplicação, utilizando interface de *sockets* e em topologias emuladas pelo mininet. O trabalho envolve as seguintes etapas:

1. Definição do formato das mensagens e do protocolo.
2. Implementação utilizando *sockets* em python.
3. Descrição/implementação da topologia de teste no mininet.
4. Escrita do relatório.

O problema

Você desenvolverá um **servidor** para um sistema simples de troca de mensagens em python, sem módulos especiais (siga as recomendações em "Módulos recomendados"), utilizando comunicação via **protocolo UDP**.

Três programas devem ser desenvolvidos: um programa servidor, que será responsável pelo controle da troca de mensagens, e dois programas clientes, um para exibição das mensagens recebidas (o chamaremos de **exibidor**) e um programa para envio de mensagens para o servidor (o chamaremos de **emissor**). Cada programa cliente se identifica com um valor inteiro único no sistema, por intermédio do servidor. Cada mensagem de texto pode ser enviada para todo mundo ou para um único usuário, não há outras opções. Esse funcionamento ficará mais claro ao longo desta especificação.

O protocolo

O protocolo de aplicação desta vez deverá funcionar sobre UDP, portanto todas as mensagens terão limites bem definidos e serão entregues como uma unidade, porém sem garantias. Será responsabilidade do programa da aplicação implementar a entrega garantida utilizando um modelo *para-e-espera*.

As mensagens possuem um formato bem definido, com quatro campos binários (inteiros de dois bytes): o tipo de mensagem, um campo de identificação de usuário (origem ou destino, dependendo do caso), um campo de numeração de sequência das mensagens e um campo de *checksum*. No caso da mensagem conter um texto, esse texto segue após o cabeçalho e pode ter no máximo 140 caracteres (como o twitter).

Os seguintes tipos de mensagens são definidos (identificadas por um valor inteiro associado):

1. **OI (0):** Primeira mensagem de um programa cliente (emissor/exibidor) para se identificar para o servidor.
2. **FLW (1):** Última mensagem de um cliente para registrar a sua saída. A partir dessa mensagem o servidor envia um OK de volta e não envia mais mensagens para aquele cliente.
3. **MSG (2):** Mensagem enviada pelo emissor para o servidor e do servidor para o exibidor. Quando enviado pelo usuário, o campo de identificação contém o número do exibidor alvo, **ou zero, para indicar que a mensagem deve ser enviada a todos os exibidores**. Quando enviada pelo servidor, o campo contém o identificador do emissor. Lembre-se que o retorno do `recvfrom` de um *socket* UDP retorna a mensagem e o endereço de origem da mesma (host e porta).
4. **OK (3):** Cada uma das mensagens anteriores, ao ser recebida pelo servidor ou pelo cliente, devem ser respondidas com uma mensagem OK. Essa mensagem funcionará como um ACK, com alguns adicionais. O campo de identificação deve ser preenchido com o identificador do cliente ou zero, se for enviada pelo servidor.

5. **ERRO (4):** Em situações onde uma mensagem não pode ser aceita por algum motivo, ela deve ser respondida com essa mensagem. O campo de identificação não precisa ser preenchido. Esta mensagem também é uma espécie de ACK, porém utilizada para indicar que alguma coisa deu errado.

6. **QEM (5):** Enviada pelo emissor para o servidor. Indica que o emissor deseja saber quais clientes (emissores e exibidores) estão registrados com o servidor. A resposta do servidor é uma mensagem do tipo OKQEM, contendo os identificadores dos clientes (veja a seguir).

7. **OKQEM (6):** É a resposta de uma mensagem QEM. O remetente dessa mensagem é sempre um servidor e o destinatário é sempre o emissor que requisitou QEM. A mensagem serve também como confirmação, necessária para a implementação de entrega confiável. O conteúdo deste tipo de mensagem é:

    - `#emissores` (`unsigned short`): número de emissores registrados com o servidor.

    - (sequência de `unsigned shorts`): `#emissores` identificadores representando os emissores.

    - `#exibidores` (`unsigned short`): número de exibidores registrados com o servidor.

    - (sequência de unsigned short): `#exibidores` identificadores representando os exibidores.

Para padronizar, vamos fixar um formato único para as mensagens:

- **Tipo da mensagem** (`unsigned short`): um dos 7 tipos definidos anteriormente.
- **Identificador de usuário** (`unsigned short`): o papel de usuário é implementado por um dos programas clientes. Vamos adotar identificadores entre 1 e 999 para usuários emissores e identificadores +1000 para usuários exibidores. Essa distinção entre identificadores de emissores e exibidores tem o objetivo de facilitar o encaminhamento das mensagens pelo servidor.
- **Número de sequência** (`unsigned short`): mantém estado entre os pares e permite a implementação da entrega confiável *para-e-espera*.
- ***Checksum*** (`unsigned short`): verificador de integridade das mensagens.
- **Mensagem**: Apenas mensagens do tipo MSG e OKQEM possuem esse campo não nulo.

    - (sequência de `unsigned char`): campo de texto de até 140 caracteres, no caso de MSG.

    - (sequência de `unsigned short`): quantidade e identificadores dos clientes, no caso de OKQEM. No máximo, podem haver 30 emissores e 40 exibidores.

Cada mensagem enviada por um programa deve receber um número de sequência diferente naquele programa, sendo que a primeira mensagem deve ser numerada com zero e as seguintes em sequência. As mensagens de resposta (OK, OKQEM e ERRO) devem carregar o número de sequência da mensagem que está sendo respondida (não têm número próprio).

O *checksum* deve ser calculado como na Internet. Por exemplo, um código que fazer esse cálculo pode ser:

```
def inet_cksum(bytes):
    len_bytes = len(bytes)
    i = cksum = 0
    while i < len_bytes:
        if i+1 >= len_bytes:
            cksum += ord(bytes[i]) & 0xFF
        else:
            w = ((ord(bytes[i]) << 8) & 0xFF00) + (ord(bytes[i+1]) & 0xFF)
            cksum += w
        i += 2
    while (cksum >> 16) > 0:
        cksum = (cksum & 0xFFFF) + (cksum >> 16)
    return ~cksum
```

Este código itera sobre pares de bytes (16 bits) da mensagem, sempre somando no acumulador `cksum`. Se o número de bytes da mensagem não for par, a última palavra será obtida com apenas 1 byte. No final, todos os bits mais significativos do que o décimo sexto (ou seja, os carries da soma, se houver) são também acumulados no resultado. O retorno é o complemento de 1 da soma, em dois octetos (2 bytes).

Detalhes de implementação

Pequenos detalhes devem ser observados no desenvolvimento de cada programa que fará parte do sistema. É importante observar que o protocolo é simples e único, de forma que programas de todos os alunos deverão ser inter-operáveis, funcionando uns com os outros.

Uso de sockets UDP

Como mencionado anteriormente, a implementação do protocolo da aplicação utilizará UDP. Isso significa que haverá apenas um socket por processo, independente de quantos outros programas se comunicarem com aquele processo. O programador deve usar as funções `sendto` e `recvfrom` para indicar explicitamente para que processo uma mensagem deve ser enviada e para determinar quem foi que enviou cada mensagem. No caso do servidor, convém manter uma estrutura de dados que já tenha a variável com a identificação do endereço de cada processo de usuário para simplificar.

Servidor

Um sistema simples de mensagens de texto apresenta apenas um processo servidor ou repetidor e a sua função é distribuir as mensagens de texto dos emissores para os exibidores a ele conectados. O servidor deve controlar os programas que se identificam por mensagens OI, descartando os que enviem mensagens FLW.

Ao receber uma mensagem QEM de um emissor, o servidor deve consultar seu estado e determinar quais clientes (emissores e exibidores) estão ativos e criar uma resposta do tipo OKQEM com as quantidades e identificadores correspondentes. Como mencionado, OKQEM também irá servir como confirmação de QEM para o emissor.

Ao receber uma mensagem MSG de um emissor, deve observar o identificador contido na mensagem. Se o identificador indicado na MSG for zero, a mensagem deve ser enviada a todos os exibidores cadastrados. Caso contrário (identificador diferente de zero), deve procurar pelo registro de um exibidor com o valor indicado e enviar apenas para ele. Caso esse exibidor não seja encontrado a resposta deve ser uma mensagem ERRO, caso contrário uma mensagem OK. Em qualquer caso, ao enviar mensagens aos exibidores, o servidor deve sempre preencher o campo de identificação com a identidade do programa que enviou a mensagem. O programa exibidor deve exibir essa informação (algo como "Mensagem de 12345: Oi, tudo bem?"). Ao enviar uma mensagem MSG a qualquer exibidor o servidor deve sempre esperar pela mensagem OK em resposta (mais sobre esse processo a seguir). O servidor deve ser capaz de lidar com até 30 emissores e 40 exibidores.

Por se tratar de um protocolo sobre UDP, entretanto, enquanto o servidor espera por uma mensagem OK do exibidor, outras mensagens podem chegar para ele (por exemplo, um novo programa tentando se conectar). Há diferentes formas de se tratar esse caso. O mais eficiente seria reagir a cada mensagem da forma apropriada, mas o mais simples (e aceitável) é simplesmente **ignorar qualquer mensagem que não seja aquela que se espera e voltar a esperar pela mensagem certa** — quem mandou a mensagem for de hora vai mandar de novo depois, de qualquer forma (lembre-se que você deve implementar a entrega confiável *para-e-espera* e por isso existirá um timer no emissor responsável por retransmitir mensagens não confirmadas).

Emissores e exibidores (tipos de cliente)

Emissores e exibidores desempenham papéis diferentes. Um emissor deve ser capaz de consultar o servidor sobre quais clientes estão ativos (QEM). Com essa informação (resposta OKQEM), o emissor mostra na tela as opções de destino (ou seja, **identificadores de exibidores**). O emissor deve ser capaz de enviar uma mensagem para todos os exibidores (quando o campo identificador for igual a zero) ou para um exibidor específico (identificador diferente de zero, com valor ≥ 1000, como estabelecemos anteriormente). O emissor deve utilizar entrada/saída padrão para isso. A cada iteração ele deve ler uma linha do teclado, segundo algum formato definido pelo programador, que deve permitir ao usuário indicar se ele deseja enviar uma mensagem para um usuário específico ou todos, ou se ele quer saber quais clientes estão ativos. Deve também haver uma forma de indicar ao programa que ele deve terminar (quando uma mensagem FLW deve ser enviada para o servidor). Esse processo se repete no emissor, para cada mensagem de texto enviada.

Além disso, quando a mensagem chegar em um exibidor, a mesma deve vir com a **informação da origem da mensagem**, ou seja, o identificador do emissor. Essas são todas as considerações sobre a parte de cliente, sinta-

se livre para decidir (e documentar!) qualquer decisão extra. Em particular, defina a interface de interação do usuário com o emissor.

Ponto extra: terminação bem comportada

Segundo esta descrição, apenas os emissores podem ser encerrados pelo usuário de forma bem comportada (dependendo de como o programador vai interpretar comandos da entrada). Você pode estender o funcionamento de cada programa para terminar de forma comportada em dois casos diferentes:

1. o usuário executa um Ctrl-C no programa cliente e ele envia uma mensagem FLW para o servidor antes de terminar;
2. o usuário executa um comando especial (definido por você) em um emissor e uma mensagem diferente (vamos chamá-la de FIM) é enviada para o servidor. Se houver mais de um emissor conectado, o servidor responde com um ERRRO e continua sua operação normal; se aquele for o único emissor, o servidor envia uma mensagem de END para cada exibidor cadastrado, depois envia um OK para o emissor e termina sua execução; o emissor então também termina sua execução.

Identificadores

Exibidores serão identificados por inteiros positivos (≥ 1000). Esses identificadores devem ser controlados pelos servidores: um servidor não deve aceitar uma identificação nova de um cliente se já houver outro cliente conectado localmente com o mesmo identificador (uma mensagem de ERRO deve ser enviada de volta nesse caso). Não há, entretanto, nenhuma validação: desde que um identificador ainda não esteja sendo usado, ele pode ser usado por qualquer usuário.

Assim, o emissor deve ler da entrada padrão tanto a mensagem quanto o identificador do exibidor destino dela (um número pelo menos 1000 ou 0 indicando o "para todos"). O servidor deve responder com ERRO caso o emissor destino não esteja registrado. Assim, os exibidores precisam ser disparados primeiro em seus testes.

Controle de recebimento de mensagens

Como mencionado anteriormente, toda mensagem recebida deve ser respondida com uma mensagem OK ou ERRO. Isso tem inclusive o efeito de confirmar se a mensagem foi recebida corretamente, já que o protocolo utilizado, UDP, não garante isso. Há diferentes formas de lidar com isso, vamos considerar duas opções (você pode escolher :D):

(Opção 1): A retransmissão ocorre na mesma thread que está esperando pela confirmação. Neste caso, a API permite associar um temporizador com o socket criado para a comunicação. Então, se a confirmação não retornar para o `recvfrom` dentro de um intervalo, uma exceção é invocada. Basicamente, você trata essa exceção com uma retransmissão e repete o processo, até que a confirmação chegue dentro do tempo. Assuma uma temporização de 1 segundo em seus testes.

(Opção 2): A retransmissão ocorre em uma thread diferente da que enviou a mensagem e que está esperando confirmação. Neste caso, você cria uma thread do tipo Timer e associa a ela um procedimento de retransmissão. Se a mensagem chegar dentro do tempo, você cancela o temporizador. Senão, a mensagem será retransmitida. Não se esqueça de programar um novo timer a cada retransmissão !

Esta é a descrição em alto nível das opções. Na seção de módulos recomendados, você encontra as direções certas para escrever o seu código.

Duas retransmissões são suficientes para o servidor, três para os clientes. Se um servidor tentar as três vezes (1+2) e não obtiver resposta, ele deve supor que o cliente não está mais ativo e retirá-lo da sua lista de clientes ativos.

Relatório e scripts

Cada aluno deve entregar junto com o código um relatório curto que deve conter uma descrição da arquitetura adotada para o servidor, os refinamentos das ações identificadas no mesmo, as estruturas de dados utilizadas e decisões de implementação não documentadas nesta especificação. Como sugestão, **considere incluir as seguintes seções no relatório: introdução, arquitetura, emissor, exibidor, servidor, discussão.** O relatório deve ser entregue em formato `.pdf`.

Os scripts devem implementar os três programas discutidos na especificação do protocolo. A forma de modularizar fica a seu critério, mas é importante descrever no relatório como executar/testar o seu código. Além disso, inclua o script da topologia mininet que você utilizou em seus testes, ou seja, o referente à figura 1.

Topologia mininet

Utilize a topologia da figura 1 para disparar os seus programas. **Faz parte da entrega o script que define esta topologia no mininet** (modifique o arquivo de topologia padrão). Como este trabalho não envolve medição de desempenho, você pode manter os parâmetros de banda e delay dos links com os valores padrões. Nos próximos trabalhos, o mininet será melhor explorado. O objetivo disso é, portanto, exercitar o uso do emulador. Com essa topologia, realize (pelo menos) os seguintes testes:

1. `emissor 1` envia "Ob-la-di" para todos os exibidores.
2. `emissor 1` envia "Ob-la-da" para `exibidor 1001`.
3. `emissor 2` envia "Life goes on, brah !!" para `exibidor 1001`.

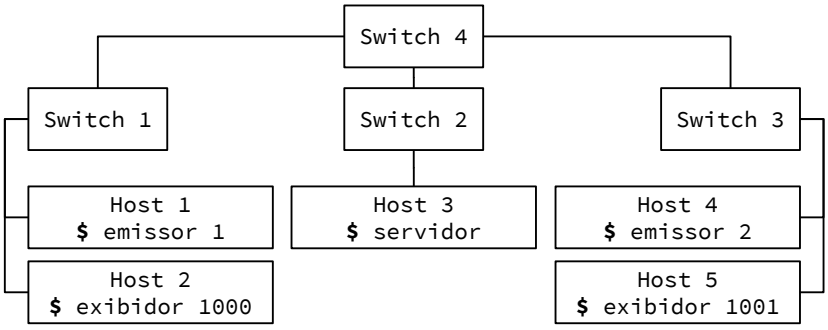

Figura 1: Topologia mininet que deverá ser utilizada nos testes

Módulos recomendados

A linguagem python possui muitos módulos destinados ao desenvolvimento de aplicações em rede, tão alto nível quanto se queira. Não queremos isso. Faz parte do trabalho de vocês aprender a lidar com os principais desafios em redes de computadores. Para os exemplos a seguir, leia a variável `s` como o socket UDP criado para a comunicação. Considere os seguintes pontos na hora de escrever o código:

- Envio e recebimento de mensagens UDP. O módulo vocês já conhecem: `sockets`. Envie mensagens com

  ```
  s.sendto(msg_bytes,addr)
  ```

  e receba mensagens com

  ```
  addr, msg_bytes = s.recvfrom(MSGLEN)
  ```

- De alguma forma você precisa traduzir suas mensagens para um array de bytes, tanto no envio quanto no recebimento. **Observe que independente da opção, o formato da mensagem continua sendo fixo. Dessa forma você tem sempre um limite superior para a quantidade de bytes enviados e esperados na entrega.** Esse limite deve respeitar o MTU, para não ultrapassar os 1492 bytes de dados do datagrama UDP.

  (Opção 1) As mensagens podem ser construídas pela concatenação de strings seguido de uma conversão para array de bytes

  ```
  msg_bytes = bytes(msg_str, 'utf-8')   (ao enviar)
  msg_str = str(msg_bytes, 'utf-8')     (ao receber)
  ```

  A codificação dos caracteres (`utf-8`, neste exemplo) pode ser diferente, o importante é manter a consistência entre envio e recebimento.

(Opção 2) As mensagens são traduzidas para o formato binário e então transformadas em um array de bytes. **Essa alternativa é preferida** por três motivos: desempenho superior, maior proximidade com o que ocorre na realidade e maior independência com relação ao ambiente. O módulo utilizado para fazer isso é o `struct`, que faz uma correspondência direta com structs C. Fica assim,

`msg_bytes = struct.pack(fmt_str, *fields_list)` (ao enviar)

`fields_list = struct.unpack(fmt_str, msg_bytes)` (ao receber)

Veja detalhes na especificação do módulo. Mas a ideia é que `fmt_str` seja uma string que defina os tipos dos campos da mensagem (cabeçalho + corpo). Por exemplo, "!Hi" define uma representação binária para rede ('!') constituída de um `unsigned short` ('H') seguido de um `int` ('i').

- Como mencionado anteriormente, você pode escolher entre duas formas para temporizar mensagens em potenciais retransmissões.

  (Opção 1) A retransmissão é feita na mesma thread que espera pela confirmação. Você só precisa do módulo `socket`, então, nada de novo. Neste caso, você deve tratar a exceção. Fica algo assim:

```
s.settimeout(1) # 1 segundo de intervalo ate a retransmissao
while True:
   try:
      s.sendto(msg_bytes, addr) # envia a mensagem
      resp_bytes, addr = s.recvfrom(MSGLEN) # recebe ACK
      # trate a mensagem, considerando o numero de sequencia dela
      # isso pode implicar em esperar mais ou interromper o loop
      # . . .
   except socket.timeout:
      # apenas continue, a mensagem será retransmitida no inicio do
      # loop
      continue
```

  (Opção 2) A retransmissão ocorre em uma thread separada, através de um `threading.Timer`. É preciso importar o módulo threading em seu script. Neste caso, você escreve a lógica de retransmissão em uma função e cria um timer, que é um tipo especial de thread que é disparada automaticamente depois de um intervalo. Fica mais ou menos assim:

```
s.sendto(msg_bytes, addr) # envia a mensagem

# a thread deve disparar em 1 segundo e executar func com args_func
# como argumentos
timer = threading.Timer (1, func, args_func)

# atenção, isso só ativa, a execução espera o momento
# certo
timer.start()
resp_bytes, addr = s.recvfrom(MSGLEN) # recebe ACK

# se a mensagem é o ACK esperado, cancela o temporizador
timer.cancel_timer()
```

  Uma coisa que não está no código acima é que `func` deve reativar o `timer` a cada retransmissão. Chame o `cancel_timer` só quando tiver certeza de que recebeu a confirmação certa (isso pode envolver a confirmação extra de mensagens com número de sequência menores, naturalmente).

Dicas e cuidados a serem observados

- Poste suas dúvidas no fórum específico para este TP na disciplina, para que todos vejam.

- Procure escrever seu código de maneira clara, com comentários pontuais e indentado.
- Consulte-nos antes de usar qualquer módulo ou estrutura diferente dos indicados aqui.
- Não se esqueça de enviar o código junto com a documentação.
- Implemente o trabalho por partes. Por exemplo, crie o formato da mensagem e tenha certeza que o envio e recebimento da mesma está correto (antes que se envolver com a lógica da entrega confiável e sistema de mensagens).

Referências úteis

- Computing the Internet Checksum (RFC).
- Módulo threading
- Módulo struct
- Tutorial mininet

Última alteração: 13 de abril de 2015