TP2: sistema de mensagens orientado a eventos

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 em uma arquitetura orientada a eventos (centrada ao redor da função `select`).
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 TCP**. O código do servidor deverá ser organizado ao redor da função `select`, usando uma técnica conhecida como **orientação a eventos**.

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 TCP, portanto todas as mensagens terão serão entregues sobre um canal de bytes, com garantias, mas é sua obrigação determinar onde começa e termina cada mensagem.

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. O cliente deve fechar a conexão.
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.
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.

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 no campo de destino um receptor para o qual deve ser enviada a lista dos clientes clientes (emissores e exibidores) que estão registrados com o servidor. O servidor deve responder com um OK para o emissor e enviar uma uma mensagem do tipo OKQEM, contendo os identificadores dos clientes (veja a seguir) para o receptor indicado.

7. **OKQEM (6):** É o resultado de uma mensagem QEM. O remetente dessa mensagem é sempre um servidor e o destinatário é sempre um receptor indicado pelo cliente que requisitou QEM.

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

- **Tipo da mensagem** (`unsigned short`, 2 bytes): um dos 7 tipos definidos anteriormente.
- **Identificador de origem** (`unsigned short`, 2 bytes): 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. O servidor tem identificador zero.
- **Identificador de destino** (`unsigned short`, 2 bytes): mesmo princípio do anterior.
- **Número de sequência** (`unsigned short`, 4 bytes): mantém estado entre os pares e permite a implementação da entrega confiável *para-e-espera*.
- ***Timestamp*** (`unsigned short`, 4 bytes): gerado pelo nó que envia uma mensagem para o servidor e pelo servidor, quando envia uma mensagem para os clientes, exceto para mensagens do tipo OK, descritas posteriormente. O seu valor pode ser obtido pelo comando `(int(time.time()))`.
- **Mensagem**: Apenas mensagens do tipo MSG e OKQEM possuem esse campo.

  No caso de MSG, contém o tamanho da mensagem (um `unsigned short` de dois bytes) e uma sequência de caracteres (`unsigned char`) com o texto de até 140 caracteres.

  No caso de OKQEM, uma sequência inteiros (`unsigned short`, 2 bytes cada), onde o primeiro indica a quantidade de clientes ativos e os demais são os identificadores de cada um dos clientes. Note que o primeiro valor determina quantos valores a mais devem ser lidos.

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 e o *timestamp* da mensagem que está sendo respondida (não têm número próprio). Quando um servidor repassa uma mensagem de um cliente para outros, ele mantém o número de sequência original, mas substitui o *timestamp* por um novo, gerado no momento em que a mensagem é recebida.

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 TCP

Como mencionado anteriormente, a implementação do protocolo da aplicação utilizará TCP. Isso significa que haverá apenas um socket por cliente, independente de quantos outros programas se comunicarem com aquele processo. O programador deve usar as funções `send` e `recv` para enviar/receber uma mensagem. No caso do servidor, ele deve manter um socket parcial para receber novas conexões (sobre o qual ele executará `accept`) e um socket para cada cliente conectado.

Emissores e exibidores (tipos de cliente)

Emissores e exibidores desempenham papéis diferentes, sendo que juntos compõem as duas partes do que seria uma interface completa de um usuário no sistema. Emissores recebem mensagens do teclado e as enviam para o servidor, enquanto exibidores recebem mensagens do servidor e as exibem na tela. Para facilitar a identificação pelo servidor, que precisará diferenciá-los, emissores terão identificadores entre 1 e 999, enquanto exibidores terão identificadores entre 1000 e 1999 (seu programa deve defir um identificador para cada instância em execução, ou permitir que o usuário forneça esse número).

Um emissor executa um loop simples, lendo linhas do teclado, formatando-as como mensagens e enviando-as para o servidor. Sua interface deve oferecer uma forma do usuário identificar o destino (um emissor específico ou todos os disponíveis), bem como permitir que o usuário indique que deseja montar uma mensagem QEM, identificando o emissor que deve recebê-la. Deve também haver uma forma de indicar ao programa que ele deve terminar (quando uma mensagem FLW deve ser enviada para o servidor). Idealmente, ele também deve ser capaz de enviar uma mensagem especial para um exibidor (você decide o formato dela) para indicar que o mesmo deve terminar sua execução (também enviando uma mensagem FLW para o servidor).

Um exibidor também executa um loop simples, esperando por mensagens do servidor e exibindo-as na tela. Ele também deve ser capaz de interpretar as mensagems OKQUEM para apresentar a informação para o usuário. Como mencionado, idealmente um exibidor também pode aceitar uma mensagem pré-definida de um emissor conhecido indicando que ele deve terminar sua execução. Em um sistema real, certamente haveria algum tipo de autenticação para garantir que o emissor que enviou a mensagem é reconhecido como com autoridade sobre o exibidor em questão. Seu projeto pode ou não definir uma forma de se fazer isso.

Essas são todas as considerações pré-definidas sobre a parte dos clientes. Sinta-se livre para decidir (e documentar!) qualquer decisão extra. Em particular, é sua tarefa definir a interface de interação do usuário com cada programa.

Servidor

Um sistema simples de mensagens de texto apresenta apenas um processo servidor ou repetidor e 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 mensagem do tipo OKQEM com as quantidades e identificadores correspondentes. Como mencionado, OKQEM será enviada para um exibidor identificado pelo emissor.

Ao receber uma mensagem MSG de um emissor, deve primeiro confirmar que o identificador de origem bate com o cliente que se identificou naquela porta (considere isso um teste para evitar que um cliente se passe por outro). Depois disso, deve observar o identificador de destino contido na mensagem. Se o identificador de destino 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. O programa exibidor deve exibir a informação de identificação de quem enviou a mensagem (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 apenas para confirmar que o cliente estava ativo e exibiu corretamente a mensagem. O servidor deve ser capaz de lidar com até 30 emissores e 40 exibidores (não estamos planejando ficar populares com esse serviço).

Pontos extras

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 tipo de cliente e do servidor para terminar de forma comportada no caso do usuário executar um Ctrl-C no programa. Nesse caso, um cliente envia

uma mensagem FLW para o servidor antes de terminar; o servidor envia uma mensagem pré-definida (por você) para todos os exibidores terminarem e depois termina. Infelizmente, no modelo atual, não há previsão de uma forma bem comportada do servidor avisar aos emissores sobre o seu término.

Interface integrada: se um emissor e um exibidor forem disparados em duas janelas de terminal exibidas próximas uma à outra, tem-se o efeito de uma interface de envio/recepção completa no modelo gtalk/whatsapp. Se você utilizar pacotes de controle de interface gráfica em Python, você pode implementar emissor e exibidor como duas threads de um mesmo processo, com uma interface integrada. Nesse caso, o resultado final é realmente de um programa único. Note que em um caso real, esse programa poderia inclusive usar a mesma conexão para enviar e receber mensagens do servidor. Neste trabalho, para mantermos o protocolo de aplicação que foi definido, você deverá continuar usando uma conexão para cada funcionalidade.

Controle de recebimento de mensagens

Como mencionado anteriormente, toda mensagem deve ser respondida com uma mensagem OK ou ERRO. Isso tem um efeito de confirmar se a mensagem foi processada corretamente pelo servidor. Note que ela não é necessária para confirmação da entrega, já que TCP é usado; apenas para a confirmação no nível da aplicação de que a mensagem foi aceita e processada (ou não).

Relatório e scripts

Cada aluno/dupla 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. 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 1002`.
3. `emissor 2` envia "Life goes on, brah!!!" para `exibidor 1001`.
4. `emissor 1` envia mensagem QEM para o servidor informar o exibidor 1001.

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 TCP criado para a comunicação. Considere os seguintes pontos na hora de escrever o código:

- Envio e recebimento de mensagens TCP. O módulo vocês já conhecem: `sockets`. Envie mensagens com `s.send(msg_bytes,addr)` e receba mensagens com `msg_bytes = s.recv(MSGLEN)`.

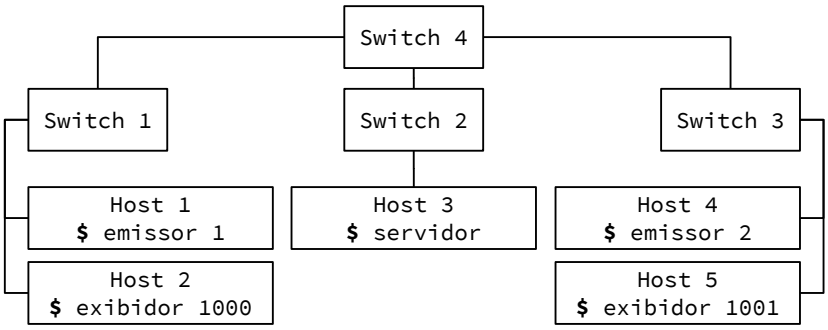

Figura 1: Topologia mininet que deverá ser utilizada nos testes

- De alguma forma você precisa traduzir suas mensagens para um array de bytes, tanto no envio quanto no recebimento. As mensagens deverão ser traduzidas para um formato binário e então transformadas em um array de bytes. 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 estamos usando TCP, o serviço é de um *stream* de bytes. Quando você faz um `send`, nem todos os bytes chegam necessariamente ao mesmo tempo e eles podem se juntar a uma outra mensagem. Para ler uma mensagem de tamanho variável, é preciso ler primeiro a parte de tamanho fixo (o cabeçalho geral) depois tratar a parte variável: ler o campo que representa o número de elementos que vêm depois e, com base no valor dele, ler cada elemento.

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).
- Em sala, foi mencionado que o trabalho precisaria usar um temporizador em algum caso. Essa exigência foi excluída da especificação: não há nenhuma situação em que seja necessário usar um temporizador neste trabalho.

Referências úteis

- Um exemplo do uso do `select`.
- Módulo threading
- Módulo struct
- Tutorial mininet

Última alteração: 27 de maio de 2016