TP2 — Sistema de Mensagens Orientado a Eventos

Prática de programação usando sockets

Data de entrega: 06/11/2017
Conteúdo da entrega: `.tar.gz` ou `.zip` contendo o código fonte
e um relatório sobre a implementação.

Objetivos e etapas

O objetivo deste trabalho é especificar e implementar um protocolo em nível de aplicação, utilizando interface de *sockets*, especialmente a função *select*. O trabalho envolve as seguintes etapas:

1. Definição do formato das mensagens e do protocolo.
2. Implementação utilizando *sockets* em python ou C em uma arquitetura orientada a eventos (centrada ao redor da função `select`).
3. Escrita do relatório.

O problema

Você desenvolverá programas para um sistema simples de troca de mensagens utilizando apenas funcionalidades equivalentes às da biblioteca de *sockets* POSIX, utilizando comunicação via protocolo TCP. O código dos programas deverá ser organizado ao redor da função `select`, usando uma técnica conhecida como *orientação a eventos*.

Dois programas devem ser desenvolvidos: um programa *servidor*, que será responsável pelo controle da troca de mensagens, e um programa *cliente* para emissão e exibição de mensagens. Os programas clientes se comunicam por meio do programa servidor. Cada programa cliente se identifica com um valor inteiro único no sistema, alocado pelo servidor. Clientes podem enviar mensagens de texto para todos os clientes (*broadcast*) ou apenas para um cliente (*unicast*). Esse funcionamento ficará mais claro ao longo desta especificação.

O protocolo

O protocolo de aplicação deverá funcionar sobre TCP. Isso implica que as mensagens serão entregues sobre um canal de bytes com garantias de entrega em ordem, mas é sua responsabilidade determinar onde começa e termina cada mensagem.

Campos comuns a todas as mensagens

Cada mensagem do protocolo de comunicação possui um cabeçalho com os seguintes campos, todos inteiros de 2 bytes representados em *network byte order*:

- **Tipo da mensagem** (2 bytes): um dos 7 tipos de mensagens definidos abaixo.
- **Identificador de origem** (2 bytes): Cada mensagem carrega o identificador a origem.
- **Identificador de destino** (2 bytes): Cada mensagem carrega também o identificador do destino.
- **Número de sequência** (2 bytes): Cada mensagem enviada por um programa deve receber um número de sequência, local ao programa. A primeira mensagem deve ter número de sequência zero e as mensagens seguintes devem ter o número de sequência da mensagem anterior mais 1. Se algum programa atingir o maior número de sequência possível, a numeração deve recomeçar de zero.

Tipos de mensagens

O protocolo possui os seguintes tipos de mensagem, cada um carregando um cabeçalho com todos os campos definidos anteriormente. Na lista a seguir, o número após o nome de cada tipo é o valor do campo "tipo da mensagem" na lista anterior.

- **`OK (1):`** Todas as mensagens do protocolo devem ser respondidas com uma mensagem de `OK` ou `ERRO` (a seguir). Essa mensagem funcionará como uma confirmação. As mensagem de `OK` devem carregar o `número de sequência` da mensagem que está sendo confirmada. O envio de uma mensagem de `OK` *não* incrementa o `número de sequência` das mensagens do cliente. Mensagens de `OK` não têm `número de sequência` próprio.

- **`ERRO (2):`** Idêntica à mensagem `OK`, mas enviada situações onde uma mensagem não pode ser aceita, por qualquer motivo. Essa mensagem também é uma espécie de confirmação, porém utilizada para indicar que alguma coisa deu errado. (Por exemplo, quando um cliente envia mensagem para um destinatário inexistente ou desconhecido.)

- **`OI (3):`** Primeira mensagem de um programa cliente para se identificar para o servidor. Apenas clientes enviam essa mensagem. O destinatário é sempre o servidor. Como o cliente envia essa mensagem antes de saber qual é o seu identificador, ele deve preencher o identificador de `origem` com o valor zero. Se o servidor aceitar a mensagem do cliente, ele envia uma mensagem `OK` contendo no campo de destino o identificador que deverá ser usado pelo cliente. O programa cliente deve exibir uma mensagem na tela informando o identificador recebido do servidor.

- **`FLW (4):`** Última mensagem de um cliente para para o servidor, para registrar sua desconexão e saída do sistema. A partir do recebimento dessa mensagem o servidor um `OK` de volta e fecha a conexão. O cliente deve esperar pelo OK, fechar a conexão e sair.

- **`MSG (5):`** Uma mensagem de texto é originada por um cliente, enviada ao servidor e repassada por ele para um ou mais clientes. O cliente deve colocar no campo `destino` o identificador do cliente para o qual a mensagem deve ser repassada. Se o campo `destino` possuir o valor zero, o servidor irá enviar a mensagem para todos os demais clientes (*broadcast*). Caso a mensagem seja repassada com sucesso, o servidor deve responder ao cliente com uma mensagem `OK`. Se o campo `destino` possuir o identificador de um cliente inexistente, a mensagem não deve ser repassada e o servidor deve responder com uma mensagem de `ERRO`. Ao repassar uma `MSG`, todos os campos, inclusive os campos `origem` e `destino`, devem permanecer inalterados.

  Logo depois do cabeçalho padrão descrito anteriormente, uma `MSG` contém um inteiro C (2 bytes em *network byte order*), indicando o número de caracteres da mensagem sendo transmitida. Depois dele, seguem C bytes contendo os caracteres da mensagem, em ASCII. Note que o valor C determina quantos bytes a mais devem ser lidos para completar a mensagem. Para simplificar a interface, considere que o tamanho de uma mensagem de texto (C) nunca será maior que 400 bytes.

  Um cliente que recebe uma `MSG` deve exibí-la na tela e enviar ao servidor uma mensagem `OK`.

- **`CREQ (6):`** Enviada de um cliente para o servidor para pedir a lista de clients. O servidor deve responder com uma mensagem do tipo `CLIST` e não com uma mensagem de `OK`.

- **`CLIST (7):`** Essa mensagem começa com um inteiro (2 bytes, network byte order) indicando número de clientes conectados, N. Em seguida, a mensagem `CLIST` carrega uma lista de N inteiros (2 bytes cada, todos em network byte order) que contém os identificadores de todos os clientes conectados ao sistema. Note que o valor N determina quantos valores a mais devem ser lidos. O remetente dessa mensagem é sempre o servidor e o destinatário é sempre um cliente que enviou uma mensagem `CREQ`. O cliente deve responder uma mensagem `CLIST` com uma mensagem `OK`.

Alocação de identificadores

O servidor aloca identificadores entre 1 e $2^{16} - 2$ (65.534) para clientes. O servidor tem identificador $2^{16} - 1$ (65.535).

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 interoperáveis, funcionando uns com os outros.

Parâmetros de execução

O servidor deve ser iniciado recebendo como parâmetro apenas o número do porto onde ele deve ouvir por conexões dos clientes.

```
$ ./servidor <porto>
```

Um cliente deve ser disparado com dois parâmetros especificando o IP e porto do servidor onde ele deve se conectar para entrar no sistema de chat.

```
$ ./cliente <ip> <porto>
```

Uso de *sockets* TCP

Como mencionado anteriormente, a implementação do protocolo da aplicação utilizará TCP. Haverá apenas um *socket* em cada cliente, independente de quantos outros programas se comunicarem com aquele processo. O programador deve usar as funções `send`, `recv` para enviar e receber mensagens e a função `select` par identificar os eventos que precisam da atenção de cada programa. No caso do servidor, ele deve manter um *socket* para receber novas conexões (sobre o qual ele executará `accept`) e um *socket* para cada cliente conectado.

Monitoramento de múltiplos eventos com a função `select`

Em um programa sequencial normal, apenas uma coisa acontece a cada momento e o programador sabe exatamente o que esperar a qualquer momento. Por exemplo, nos trabalhos anteriores, os programas envolvidos sabiam exatamente quando deveriam fazer um `recv` para receber uma mensagem em um certo *socket*. Em programas que interagem com o usuário ou com vários outros programas, entretanto, isso nem sempre é verdade: um servidor que está conectado a diversos clientes não sabe em que socket vai chegar a próxima mensagem, então ele não pode parar em um `recv` para uma conexão específica. Nesse caso, o programa tem que ser escrito de forma a esperar por qualquer evento que possa ocorrer e reagir com base no tipo de evento que ocorreu.

A função `select` tem exatamente essa função. Ao chamá-la, o programador indica quais sockets ou arquivos devem ser monitorados para leitura, escrita ou situações especiais. Vamos considerar leituras, para simplificar. Se um programa pode receber mensagens de vários outros programas, cada um com sua conexão, representada por sockets, ele chama o `select` com esse conjunto de *sockets*. Essa função bloqueia até que chegue dados em algum desses *sockets* (ou até que um certo tempo passe — veja a documentação para os detalhes). Ao retornar a função indica quais eventos ocorreram (quais *sockets* receberam mensagens e devem devem ser lidos). O programador escreve o código para identificar esses eventos e reagir adequadamente.

Além de *sockets*, `select` também pode receber descritores de arquivos tradicionais. Assim, um descritor de arquivo associado ao teclado (p.ex., o *stdin*, descritor zero) ou a um *pipe* também podem ser passados para a função observar.

Como mencionado, `select` também permite indicar *sockets* onde se deseja escrever ou onde se procura por alguma exceção, mas esses dois conjuntos não interessam neste trabalho. Também não é necessário usar o temporizador que pode ser associado à chamada.

Servidor: monitoramento de múltiplos clientes em uma única *thread*

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 repassar mensagens entre os programas que se identificam por mensagens `OI`, desconectando os que enviem mensagens `FLW`.

O código do servidor deverá ser organizado ao redor da função `select`. O programa deve montar um conjunto de descritores indicando todos os *sockets* dos quais espera uma mensagem (inclusive o *socket* usado para fazer o `accept`, que deve ser único). Quando a chamada retornar, o servidor deve verificar quais *sockets* devem ser tratados e realizar a ação adequada em cada caso.

Ao receber qualquer mensagem (exceto a `OI`), o servidor deve confirmar que o identificador de `origem` corresponde ao do cliente que a enviou, verificando se o cliente indicado na `origem` é o cliente conectado no *socket* onde a mensagem foi recebida. Esse teste evita que um cliente se passe por outro.

Se a mensagem é do tipo MSG, o servidor deve observar o identificador de destino contido na mensagem. Se o identificador de `destino` indicado na `MSG` for zero, a mensagem deve ser respondida à `origem` com um `OK` e repassada a todos os outros clientes conectados ao sistema. Caso o identificador seja diferente de zero, ele deve verificar se o valor indica um cliente válido. Em caso positivo, o servidor deve responder um `OK` à `origem`, procurar pelo registro do destinatário e repassar a `MSG` apenas para aquele cliente. Caso o `destino` não seja um cliente válido, o servidor deve responder à `origem` com uma mensagem `ERRO`. Ao repassar uma mensagem `MSG` a qualquer cliente, o servidor deve esperar pela mensagem `OK` em resposta para confirmar que o cliente estava ativo e exibiu corretamente a mensagem. O servidor deve ser capaz de lidar com pelo menos 255 clientes concorrentes, que podem ter qualquer identificador válido.

Ao receber uma mensagem `CREQ` de um cliente, o servidor deve consultar seu estado, determinar quais clientes estão ativos, criar uma mensagem do tipo `CLIST` com o número de clients e a lista com seus identificadores e enviar a mensagem de volta para o cliente. O servidor deve então esperar uma mensagem `OK` do cliente confirmando o recebimento da mensagem `CLIST`.

Ao repassar uma mensagem `MSG` a um cliente, o servidor deve sempre esperar pela mensagem `OK` de volta. Para evitar que o servidor espere indefinidamente por um cliente falho, uma temporização deve ser associada a cada *socket* para que todo `recv` retorne dentro de 5 segundos. Se algum cliente não responder nesse tempo ele deve ser removido da lista de clientes ativos e sua conexão deve ser fechada.

Clientes: leitura de teclado e recebimento de mensagems em uma única *thread*

Para operar corretamente, um cliente precisa realizar duas funções principais: (1) receber mensagens da rede e exibi-las na tela e (2) ler mensagens do teclado e transmiti-las pela rede.

O recebimento e transmissão de mensagens devem ser realizados utilizando o *socket* TCP como descrito na subseção anterior. A leitura de mensagens do teclado e exibição de mensagens na tela devem ser feitos via entrada e saída padrão (também conhecidos como `stdin` e `stdout`).

O cliente não tem como saber se ele deve parar para ler um comando digitado pelo usuário no teclado ou ler uma mensagem da rede para exibi-la na tela. Assim, ele deve ser capaz de receber comandos e mensagens a qualquer momento. Isso é possível usando a função `select` para monitorar o descritor da entrada do teclado e o *socket* usado para a conexão com o servidor, como mencionado anteriormente. Quando alguma mensagem é recebida em algum desses descritores, a função `select` retorna e dá ao programa a oportunidade de tratar o evento. (Dica: Para obter o descritor de arquivos da entrada padrão, utilize a função `fileno` em C ou Python.)

Ao receber uma mensagem `MSG` do servidor, o cliente deve exibir a informação de identificação de quem enviou uma mensagem. Por exemplo, "Mensagem de 37: Oi, tudo bem?".

A interface do cliente com o usuário deve permitir que o mesmo use três comando: (1) enviar uma mensagem para um outro cliente ou para todos, (2) pedir a lista de todos os clientes ativos e (3) sair do sistema. Por exemplo, uma interface simples seria o usuário começar cada linha com um `M`, `L` ou `S` para indicar se quer enviar uma mensagem, pedir a lista de clientes ou sair do sistema; no caso da mensagem, depois do M ele poderia incluir o identificador do destino e o texto, por exemplo..

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. O seu relatório deve detalhar como funciona a interface do seu cliente. É importante notar, entretanto, que o protocolo está pré-definido e não pode ser alterado, exceto para incluir as funcionalidades dos pontos extras, se for o caso.

Pontos extras

Identificação de clientes

Estenda o protocolo, cliente e servidor para dar suporte ao uso de apelidos (*nicknames*) no sistema de chat. Suas extensões devem suportar, no mínimo:

1. O cliente pode receber o apelido na linha de comando e informá-lo na mensagem `OIAP (13)`.
2. O cliente pode pedir a lista de apelidos junto com os identificadores usando uma mensagem `CREQAP (16)` e exibi-los na tela.
3. O servidor deve enviar uma mensagem `CLISTAP (17)` com a lista com identificadores e apelidos para os clientes que enviarem mensagens `CREQAP`.
4. O envio de mensagens pode ser feito usando uma mensagem `MSGAP (15)` que tem o identificador do servidor como destino e o apelido do destinatário adicionado de alguma forma à mensagem. A mensagem repassada pelo servidor para o destino deve ser do tipo `MSG`, de qualquer forma.

Mesmo com as extensões, seu servidor deve continuar funcionando normalmente com clientes não estendidos (já que as mensagens têm códigos diferentes nesse caso).

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 código

Cada aluno ou dupla deve entregar junto com o código um relatório curto que deve conter uma descrição da arquitetura adotada para o servidor e o cliente, 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, cliente, servidor, e discussão. O relatório deve ser entregue em formato PDF.

A forma de modularização do código fica a seu critério, mas é importante descrever no relatório como compilar, executar e utilizar seus programas.

Dicas e cuidados a serem observados

- O guia de programação em rede do Beej (`http://beej.us/guide/bgnet/`) e o Python Module of the Week (`https://pymotw.com/2/select/`) têm bons exemplos de como organizar um servidor com `select`.
- Poste suas dúvidas no fórum específico para este TP na disciplina, para que todos vejam.
- Escreva seu código de maneira clara, com comentários pontuais e indentado. O código será considerado na nota.
- Consulte-nos antes de usar qualquer biblioteca diferente que não seja padrão.
- Não se esqueça de enviar o código junto com a documentação.
- Implemente o trabalho por partes. Por exemplo, crie o código para montar, enviar, receber e entender o cabeçalho das mensagens internas do sistema antes que se envolver o problema de ler e exibir mensagens do usuário.

Última alteração: 23 de outubro de 2017