TP2 — Sistema de Mensagens Orientado a Eventos

Prática de programação usando sockets

Data de entrega: 19/06/2017
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*. 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 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, um programa *exibidor* para exibição das mensagens recebidas e um programa *emissor* para envio de mensagens para o servidor. Os programas exibidores e emissors, também chamados de *clientes* se comunicam por intermédio do programa servidor. Cada programa cliente se identifica com um valor inteiro único no sistema, alocados pelo servidor. Programas emissores podem enviar mensagens de texto para todos os programas exibidores (*broadcast*) ou apenas para um programa exibidor (*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:

- **Tipo da mensagem** (2 bytes): um dos 7 tipos de mensagens definidos abaixo.
- **Identificador de origem** (2 bytes): Cada mensagem carrega o identificador da origem (ou um valor pré-definido, no caso da mensagem OI, a seguir).
- **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.

Tipos de mensagens

O protocolo possui os seguintes tipos de mensagem. Todos os tipos de mensagem carregam um cabeçalho com todos os campos definidos anteriormente. O número após 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`. 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.
- **`OI (3):`** Primeira mensagem de um programa cliente (emissor/exibidor) 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 ele é um exibidor; um valor diferente de zero identifica um emissor. Se esse valor estiver entre 2^{12} e $(2^{13} - 1)$, ele identifica um exibidor (que já deve estar em execução) e que deve ser associado àquele emissor. Se o servidor aceitar a mensagem do cliente, ele envia uma mensagem `OK` contendo no campo de destino o identificador que deverá ser usado a partir daí pelo cliente. Todo cliente deve exibir uma mensagem informando o identificador recebido do servidor.
- **`FLW (4):`** Última mensagem de/para um cliente para registrar sua desconexão e saída do sistema. A partir dessa mensagem o servidor/cliente que recebe a mensagem envia um `OK` de volta e fecha a conexão. Se o servidor recebe a mensagem de um emissor, ele deve verificar se há um exibidor associado (identificado por ele na mensagem `OI`). Se houver, ele (servidor) envia uma mensagem `FLW` para aquele exibidor. Se um exibidor recebe a mensagem, ele deve terminar sua execução depois de responder.
- **`MSG (5):`** Uma mensagem é originada por um emissor, enviada ao servidor, e repassada pelo servidor para um ou mais exibidores. O emissor deve colocar no campo `destino` o identificador do exibidor para o qual a mensagem deve ser repassada, ou de um emissor associado a um exibidor. Se o campo `destino` possuir o valor zero, o servidor irá enviar a mensagem para todos os exibidores (*broadcast*). Se o campo `destino` possuir o identificador de um emissor, a mensagem deve ser enviada para o exibidor associado a ele (se existir) ou um ERRO deve ser retornado. Ao repassar uma `MSG`, todos os campos, inclusive os campos `origem` e `destino`, devem permanecer inalterados.

  Uma `MSG` começa com um inteiro (2 bytes, network byte order) logo após o cabeçalho, indicando o número de caracteres sendo transmitidos, *C*. Depois do inteiro, seguem *C* bytes contendo os caracteres da mensagem em ASCII. Note que o valor *C* determina quantos bytes a mais devem ser lidos.
- **`CREQ (6):`** Enviada pelo emissor para o servidor. Indica no campo de destino um receptor para o qual deve ser enviada a lista de clientes (emissores e exibidores) que estão conectados ao sistema. O servidor deve responder com um `OK` para o emissor da mensagem `CREQ`. O servidor deve também enviar uma mensagem do tipo `CLIST` para o destinatário indicado na mensagem `CREQ`. (Opcional: seu servidor pode suportar *broadcast* de `CLIST` para todos os exibidores caso o campo `destino` do `CREQ` tenha o valor zero.)
- **`CLIST (7):`** Essa mensagem possui um inteiro (2 bytes, network byte order) indicando número de clientes conectados, *N*. A mensagem CLIST possui também uma lista de *N* inteiros (2 bytes cada, todos em network byte order) que armazena os identificadores de cada cliente (exibidor e emissor) conectador 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 um cliente informado como destinatário da mensagem `CREQ`. O cliente deve responder uma mensagem `CLIST` com uma mensagem `OK`.

Alocação de identificadores

O servidor aloca identificadores entre 1 e $(2^{12} - 1)$ para emissores e identificadores entre 2^{12} e $(2^{13} - 1)$ para exibidores. Essa distinção entre identificadores de emissores e exibidores tem o objetivo de facilitar o encaminhamento das mensagens pelo servidor. O servidor tem identificador $(2^{16} - 1)$.

Associação entre emissores e exibidores

Não há obrigatoriedade de que um emissor esteja sempre associado a um emissor. Podem haver emissores sem um exibidor, bem como exibidores sem um emissor. A associação acontece no momento da conexão do emissor, se o identificador fornecido for um exibidor presente no sistema. É responsabilidade do servidor manter o controle sobre quem está associado a quem, quando for o caso.

Isso pressupõe que a informação sobre a identificação dos exibidores é manipulada pelo usuário: primeiro ele(a) dispara um exibidor e observa o identificador que ele recebe do servidor e exibe na sua saída; em seguida o usuário deve disparar o emissor, passando como parâmetro para ele o identificador que foi informado pelo exibidor.

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.

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` e `recv` para enviar e receber mensagens. 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.

Parâmetros de execução

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

Um exibidor deve ser disparado com um parâmetro obrigatório que identifica a localização do servidor como um par "endereço IP:porto".

Um emissor deve ser disparado com um parâmetro obrigatório que identifica a localização do servidor como um par "endereço IP:porto", que pode ser seguido por um segundo parâmetro, opcional, com o identificador de um exibidor, caso uma associação deva ser feita.

Clientes: emissores e exibidores

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.

Depois de sua inicialização, o 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 exibidor específico ou todos os exibidores no sistema), bem como permitir que o usuário indique que deseja montar uma mensagem CREQ, identificando o exibidor 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).

Depois de sua inicialização, o exibidor também executa um *loop* simples, esperando por mensagens do servidor e exibindo-as na tela. Ele também deve ser capaz de interpretar as mensagens `CLIST` para apresentar a informação para o usuário. O programa exibidor deve exibir a informação de identificação de quem enviou a mensagem (algo como "Mensagem de 37: Oi, tudo bem?").

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 repassar mensagens entre os programas que se identificam por mensagens `0I`, desconectando os que enviem mensagens `FLW`.

O código do servidor deverá ser organizado ao redor da função `select`, que permite a observação de diversos sockets em paralelo. 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 um só). A função permite indicar sockets também 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 ao select. Uma vez chamada, a função `select` bloqueia até que algo ocorra nos sockets que foram indicados na chamada. Ao retornar, o conjunto indica quais sockets têm operações pendentes.

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

Depois disso, 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 enviada a todos os exibidores conectados ao sistema. Caso o identificador seja diferente de zero, ele deve verificar se o valor indica um emissor ou um exibidor. Se for um exibidor, o servidor deve procurar pelo registro de um exibidor com o valor indicado e repassar a `MSG` apenas para aquele cliente. Se for um emissor, o servidor deve verificar se há um exibidor associado a ele e enviar a mensagem para aquele cliente. Caso um exibidor não seja encontrado, o servidor deve responder ao emissor com uma mensagem `ERRO`, caso contrário uma mensagem `OK`. Ao enviar uma mensagem `MSG` a qualquer exibidor, o servidor deve 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 pelo menos 255 clientes.

Ao receber uma mensagem `CREQ` de um emissor, o servidor deve consultar seu estado e determinar quais clientes (emissores e exibidores) estão ativos e criar uma mensagem do tipo `CLIST` com o contador e lista de identificadores de clientes. Como mencionado, `CLIST` será enviada para um exibidor identificado pelo emissor.

Pontos extras

Protocolo de terminação

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 que recebe um `Ctrl-C` envia uma mensagem `FLW` para o servidor antes de terminar; o servidor que recebe um `Ctrl-C` envia uma envia uma mensagem FLW para todos os exibidores terminarem, espera o OK de cada um deles 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 uma biblioteca de controle de interface gráfica em Python, pode implementar emissor e exibidor como duas partes/threads de um mesmo processo, com uma interface integrada. Nesse caso, o resultado final é realmente 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 código

Cada aluno/dupla/trio 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, 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 formato da mensagem e tenha certeza que o envio e recebimento da mesma está correto antes que se envolver com o sistema de mensagens.

Última alteração: 12 de junho de 2017