

TRABALHO PRÁTICO 2

Sistema de Mensagens Orientado a Eventos

Aluno: Yuri Diego Santos Niitsuma - 2011039023

Aluno: Alison de Oliveira Souza - 2012049316

Introdução

Neste trabalho, implementamos um sistema simples de troca de mensagens utilizando *sockets* da biblioteca POSIX, através do protocolo TCP, e a técnica conhecida como *orientação a eventos*.

A programação orientada a eventos é um paradigma de programação que, diferentemente dos outros paradigmas, segue um fluxo de controle padronizado, guiados por indicações externas ao programa chamadas de eventos.

Desenvolvemos três programas para realizar as trocas de mensagens. Um é o servidor, que é responsável pelo gerenciamento dos clientes (tanto emissores quanto exibidores) e pelo controle da troca de mensagens entre os clientes. Os outros dois programas são do tipo clientes, divididos em exibidor, responsável pela exibição das mensagens recebidas, e emissor, responsável pelo envio de mensagens para outros clientes através do servidor. Cada cliente é único no sistema, e isso é garantido atribuindo um valor de identificação diferente a cada cliente pelo servidor (entre 1 e $(2^{12}-1)$ identifica um emissor, e entre 2^{12} e $(2^{13}-1)$ identifica um exibidor). Com essa identificação, os clientes podem enviar mensagens apenas entre si ou para todos os clientes ativos (broadcast).

Essa é uma representação dos campos comuns a todas as mensagens:

Tipo da mensagem (2 bytes)	Identificador de origem (2 bytes)	Identificador de destino (2 bytes)	Número de sequência (2 bytes)
-------------------------------	--------------------------------------	---------------------------------------	----------------------------------

Alguns tipos de mensagens tem apenas, e exatamente, os campos acima definidos. É o caso das mensagens de tipo OI (mensagem de conexão entre um cliente e servidor), FLW (mensagem de desconexão entre cliente e servidor), OK (mensagem de confirmação de processamento correto da mensagem anterior), ERRO (mensagem de identificação de erros no processamento da mensagem

anterior) e CREQ (mensagem de requisição da lista de clientes conectados ao servidor). Em alguns casos há mais alguns campos nas mensagens enviadas entre clientes e servidor, dependendo do tipo de mensagem enviado entre tais. Por exemplo, mensagens do tipo MSG tem um campo com um número inteiro N e um campo de dados de tamanho N bytes, que é a mensagem propriamente dita. Já as mensagens do tipo CLIST também possui um inteiro N e um campo de N bytes correspondente a lista de clientes retornada pelo servidor após um cliente solicitar através de uma mensagem CREQ.

Arquitetura

Nosso trabalho foi implementado em Python 3, e dividimos o código em 5 arquivos os quais são descritos abaixo:

- utils.py:

Arquivo que contém algumas definições de classes e constantes úteis em todo o sistema e funções auxiliares para depuração.

- server.py:

Arquivo que contém todos os métodos e classes do servidor do sistema. Nele, definimos uma classe **Connection**, que é destinada a guardar informações sobre a conexão do servidor com o cliente. Definimos também uma classe **Server** que define o comportamento do servidor. O servidor deverá esperar por alterações nas conexões com os clientes (seja um novo cliente ou alguma mensagem) ou por comandos dados pela entrada padrão, que serão citados mais tarde. Quando o servidor identifica uma alteração em um ou mais sockets, ele irá executar uma ação dependendo do tipo de alteração, como por exemplo criar uma nova conexão caso seja um novo cliente (com a mensagem OI), ou receber uma mensagem de um cliente (diferente de OI), que será tratada de acordo com seu tipo, ou mesmo tratar o comando recebido pela entrada padrão.

Os comandos permitidos, e suas respectivas descrições, são:

1. /help
Exibe a lista de comandos permitidos com uma breve descrição.
2. /status
Mostra as conexões ativas e seus identificadores.
É útil para saber qual ID um exibidor está associado para linkar com um emissor.
3. /list
Exibe todas as conexões na tela.
4. /quit
Encerra o servidor.

- client.py:

Esse arquivo é responsável pela base dos exibidores e emissores do sistema. Nele, temos a classe **Client** que define os métodos e variáveis comuns aos emissores e exibidores, como os métodos de conexão com o servidor, e de envio e recebimento de mensagens.

- exibidor.py:

Neste arquivo temos os métodos e classes exclusivas aos exibidores do sistema. Criamos a classe **Exibidor**, estendendo a classe **Client**, e definimos alguns métodos auxiliares para o tratamento das mensagens. O exibidor deve simplesmente aguardar as mensagens direcionadas a ele, exibi-las na tela e enviar uma mensagem de OK ou ERRO para o servidor.

- emissor.py:

Neste arquivo, definimos os métodos e classes exclusivas dos emissores do sistema. Criamos a classe **Emissor**, estendendo a classe **Client** citada acima, e definimos alguns métodos auxiliares para o tratamento das mensagens. O emissor deve enviar suas mensagens ao servidor e verificar sua conexão com o servidor, esperando a mensagem de confirmação, seja ela um **OK** ou **ERRO**.

Diferente das outras classes, o **Emissor**, não imprime nada, pois seu papel serve apenas em enviar comandos para o servidor e exibir respostas ao seu exibidor associado. Mas, assim como no **server.py**, eles possuem alguns comandos para requisições.

Os comandos permitidos, e suas respectivas descrições, são:

1. /list
Exibe todas as conexões ativas no servidor, a resposta é exibida no exibidor associado ao emissor.
2. /listb
Exibe todas as conexões ativas no servidor, a resposta é exibida em todos os exibidores.
3. /msg <id_de_destino> <mensagem>
Envia mensagem privada para o ID associado. A mensagem é enviada ao exibidor associado do emissor e do associado ao id_de_destino. Lembrando que os ID's de usuários pertencem aos emissores.

** Nos métodos de listagem foi utilizado a definição de CREQ e CLIST definido na especificação.*

O formato das mensagens é:

[id:ID_USER(SEQ_NUM)]> MENSAGEM

- ID_USER: id de usuário pertencente ao **emissor**.
- SEQ_NUM: Número de sequência da mensagem, definimos que é incrementado a cada OK recebido.
- MENSAGEM: Mensagem de texto ascii. (Não é possível utilizar acentos)

O **FLW** foi implementado do seguinte modo nos clientes e servidor.

- Se o servidor se fechar, ele manda iterativamente FLW a cada cliente espera um OK antes de fechar a conexão.
- Se o cliente for fechar, ele manda um FLW para o servidor e espera um OK do servidor para enfim se encerrar.

Execução

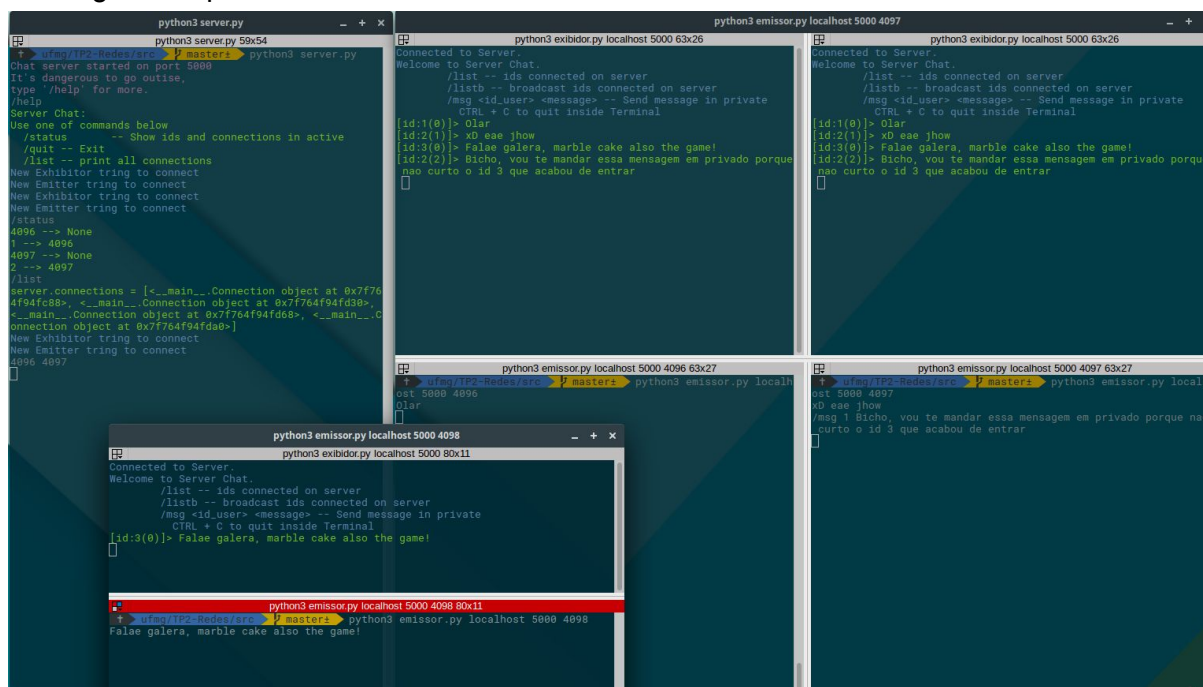
Para executar o sistema, deve-se executar os seguintes comandos no terminal:

1. `python3 server.py <PORT>`
 2. `python3 exibidor.py <HOSTNAME> <PORT>`
 3. `python3 emissor.py <HOSTNAME> <PORT> <EXIBIDOR_ID>`
- O servidor sempre deve ser o primeiro a ser executado. Caso não seja passado o número da porta <PORT> como parâmetro, ele executará na porta 5000.
 - O emissor pode ser executado antes do exibidor, desde que esse emissor não for se associar ao exibidor.
 - Para se executar em uma única máquina, <HOSTNAME> poderá receber 'localhost' como parâmetro.

Caso queira associar um emissor a um exibidor, deve ser passado o identificador do exibidor como o parâmetro <EXIBIDOR_ID>. Esse parâmetro não precisa ser passado caso não haja associação entre emissor e exibidor.

Testes

Segue um exemplo em que 3 usuários utilizando o chat e um deles enviando uma mensagem em privado.



```
python3 server.py
python3 server.py 5000
Chat server started on port 5000
It's dangerous to go outside,
type '/help' for more.
/help
Server Chat:
Use one of commands below
-- Show ids and connections in active
/status
-- Exit
/quit
-- print all connections
/list
-- Exhibitor trying to connect
New Exhibitor trying to connect
New Exhibitor trying to connect
New Exhibitor trying to connect
/status
4096 --> None
4097 --> None
4098 --> None
/list
server.connections = [c._main...Connection object at 0x7f764f94fc88, c._main...Connection object at 0x7f764f94fd30, c._main...Connection object at 0x7f764f94fd68, c._main...Connection object at 0x7f764f94fd90]
New Exhibitor trying to connect
New Exhibitor trying to connect
4096 4097
[]

python3 exibidor.py localhost 5000 63x26
Connected to Server.
Welcome to Server Chat.
/list -- ids connected on server
/listb -- broadcast ids connected on server
/msg <id_user> <message> -- Send message in private
CTRL + C to quit inside Terminal
[id:1(0)]> Olá.
[id:2(1)]> xD eae jhow
[id:3(0)]> Fala galera, marble cake also the game!
[id:2(2)]> Bicho, vou te mandar essa mensagem em privado porque nao curto o id 3 que acabou de entrar
[]

python3 emissor.py localhost 5000 4097
Connected to Server.
Welcome to Server Chat.
/list -- ids connected on server
/listb -- broadcast ids connected on server
/msg <id_user> <message> -- Send message in private
CTRL + C to quit inside Terminal
[id:1(0)]> Olá.
[id:2(1)]> xD eae jhow
[id:3(0)]> Fala galera, marble cake also the game!
[id:2(2)]> Bicho, vou te mandar essa mensagem em privado porque nao curto o id 3 que acabou de entrar
[]

python3 emissor.py localhost 5000 4096 63x27
Connected to Server.
Welcome to Server Chat.
/list -- ids connected on server
/listb -- broadcast ids connected on server
/msg <id_user> <message> -- Send message in private
CTRL + C to quit inside Terminal
[id:1(0)]> Olá.
[id:2(1)]> xD eae jhow
[id:3(0)]> Fala galera, marble cake also the game!
[id:2(2)]> Bicho, vou te mandar essa mensagem em privado porque nao curto o id 3 que acabou de entrar
[]

python3 emissor.py localhost 5000 4098 80x11
Connected to Server.
Welcome to Server Chat.
/list -- ids connected on server
/listb -- broadcast ids connected on server
/msg <id_user> <message> -- Send message in private
CTRL + C to quit inside Terminal
[id:3(0)]> Fala galera, marble cake also the game!
[]
```

A seguir, segue a figura com exemplo do `/list` (listagem de usuários ativos em privado)

The screenshot displays a chat application interface with four terminal windows. The top-left window shows the command `/list` being executed, resulting in a list of connection IDs: 4098, 3, 4096, 4097, 1, 2. The top-right window shows the command `/listb` being executed, resulting in a list of connection IDs: 4096, 1, 4097, 2, 4098, 3. The bottom-left window shows the command `/list` being executed, resulting in a list of connection IDs: 4096, 1, 4097, 2, 4098, 3. The bottom-right window shows the command `/listb` being executed, resulting in a list of connection IDs: 4096, 1, 4097, 2, 4098, 3.

A seguir, segue a figura com exemplo do `/listb` (listagem de usuários ativos em broadcast)

The screenshot displays a chat application interface with four terminal windows. The top-left window shows the command `/listb` being executed, resulting in a list of connection IDs: 4096, 1, 4097, 2, 4098, 3. The top-right window shows the command `/listb` being executed, resulting in a list of connection IDs: 4096, 1, 4097, 2, 4098, 3. The bottom-left window shows the command `/listb` being executed, resulting in a list of connection IDs: 4096, 1, 4097, 2, 4098, 3. The bottom-right window shows the command `/listb` being executed, resulting in a list of connection IDs: 4096, 1, 4097, 2, 4098, 3.

Considerações finais

Vemos que este trabalho realmente se tornou mais simples devido aos conhecimentos sobre sockets introduzidos nos trabalhos anteriores. Assim, a grande dificuldade foi tratar dois programas independentes como um só, nesse caso os emissores e exibidores. Associar os dois programas para trabalharem em conjunto deixou um pouco confuso na hora de receber uma mensagem. Ter que definir se o emissor iria receber as mensagens, ou se as mensagens enviadas para o emissor seriam repassadas ao exibidor nos deixou um pouco em dúvida sobre o que fazer. Outra dúvida era sobre o tratamento de mensagens quando enviadas em broadcast, se quando enviamos uma mensagem para todos os usuários o emissor que não tenha exibidores associados deveria realizar o papel de exibidor nesse caso. Decidimos que um emissor sem exibidor associado nunca exibirá uma mensagem, simulando um servidor de mensagens de operadora telefônica, por exemplo, que envia mensagens para seus clientes mas ignora as mensagens recebidas.