Trabalho Prático 3 - Redes de Computadores

Priscila Izabelle De-Stefano Santos - 2011054758
Silvana Mara Ribeiro - 2011022147

9 de julho de 2018

1 Introdução

O presente trabalho trata da implementação de um servidor e clientes de troca de mensagens, nos quais os clientes se inscrevem ou desinscrevem para receber mensagens em que estão interessados e enviam mensagens com tags para o servidor e o servidor mantém a lista de tags de interesse de cada cliente e repassa mensagens pertinentes para os clientes interessados nestas.

2 Implementação

2.1 Servidor

Ao ser inicializado, o servidor abre uma conexão na porta informada pelo usuário e espera por mensagens. A forma como se decidiu armazenar os interesses de cada cliente foi em um dicionário onde a chave é a combinação do IP e porta do cliente e o valor é uma lista com as tags de interesse. Assim que recebe uma mensagem o servidor verifica se os caracteres da mensagem estão dentre os caracteres aceitos. Caso estejam continua o fluxo do programa e caso não estejam a escolha de implementação foi apenas não continuar o fluxo.

```
15    def is_message_valid(message):
16        valid_chars = ",.?!:;+-*/=@#$%()[]{}1234567890abcdefghijklmnopqrstuvxwyzABCDEFGHIJKLMNOPQRSTUVXWYZ "
17        for c in message:
18            if c not in valid_chars:
19                print('nao eh valida')
20                return False
21        return True
```

Figura 1: Trecho de código da validação das mensagens

Caso a mensagem seja válida, o próximo passo é decodificá-la de modo a identificar as tags que ela contém. Isto foi feito utilizando uma expressão regular, que percorre *string* e extrai dela trechos que iniciem com algum caracter delimitador (#, + ou -). Tomou-se a decisão de implementação de que uma tag não contem espaços. Ou seja, a *string* "Olha a #copa gente"possui a tag #copa.

```
23    def decode_message(message):
24        final_result = re.findall('[#\-+][0-9a-z-A-Z]*', message)
25        return final_result
```

Figura 2: Trecho de código que decodifica a mensagem, separando-a em tags

Em seguida, as tags identificadas na mensagem são percorridas e se começarem com + são inseridas na lista de interesses de cliente se já não existirem. Se começam com - são removidas da lista de interesses do cliente. Se começam com #, percorre-se o dicionário contendo os clientes e seus interesses e se algum cliente estiver inscrito para receber mensagens contendo esta tag, toda a mensagem é enviada ao mesmo. Como decisão de implementação, mensagens são enviadas ao cliente caso ele tente adicionar uma tag que já exista em sua lista de interesses ou tente remover uma tag que não exista em sua lista.

```python
def add_tag_to_client(ip_port, tag, dict_client_tags):
    tag = tag[1:]
    if dict_client_tags:
        if ip_port in dict_client_tags:
            temp_list = dict_client_tags[ip_port]
            if tag not in temp_list:
                temp_list.append(tag)
                dict_client_tags[ip_port] = temp_list
            else:
                return False
        else:
            dict_client_tags[ip_port] = [tag]
    else:
        return False
    return dict_client_tags

def remove_tag_from_client(ip_port, tag, dict_client_tags):
    tag = tag[1:]
    if dict_client_tags:
        if ip_port in dict_client_tags:
            temp_list = dict_client_tags[ip_port]
            try:
                temp_list.remove(tag)
            except:
                return False
            dict_client_tags[ip_port] = temp_list
    else:
        return False
    return True

```

Figura 3: Trecho de código que adiciona e remove tags

2.2 Cliente

Ao ser inicializado, o cliente abre uma conexão na porta informada pelo usuário e espera por mensagens ou por comandos digitados via teclado. Como decisão de implementação, foi decidido que o cliente pode enviar comandos para adicionar ou remover tags no meio de frases. Sendo assim, a mensagem "Quero saber da +copa e do +brasil"resultaria nas tags "copa"e "brasil"sendo inseridas na lista de interesses do cliente. Além disso, um cliente não recebe mensagens enviadas por ele mesmo. Conforme orientado na especificação, o controle de leitura do teclado e recebimento de dados da rede é feito utilizando *select*.

```python
        while running:
            inputready,outputready,exceptready = select.select(input,[],[],)

            for s in inputready: # PARA LINUX
                if s == server:
                    data = s.recv(500).decode('ascii')
                    print(data)
                elif s == stdin:
                    # handle standard input
                    TEXT = stdin.readline()
                    TEXT = TEXT.replace('\n', '')
                    send_message(HOST, PORT, TEXT)
                else:
                    # handle all other sockets
                    print("outros sockets")
                    data = s.recv(500).decode('ascii')
                    print(data)
                    if data:
                        s.send(data)
                    else:
                        s.close()
                        input.remove(s)
```

Figura 4: Trecho de código que realiza o controle de leitura do teclado/recebimento de dados

3 Testes

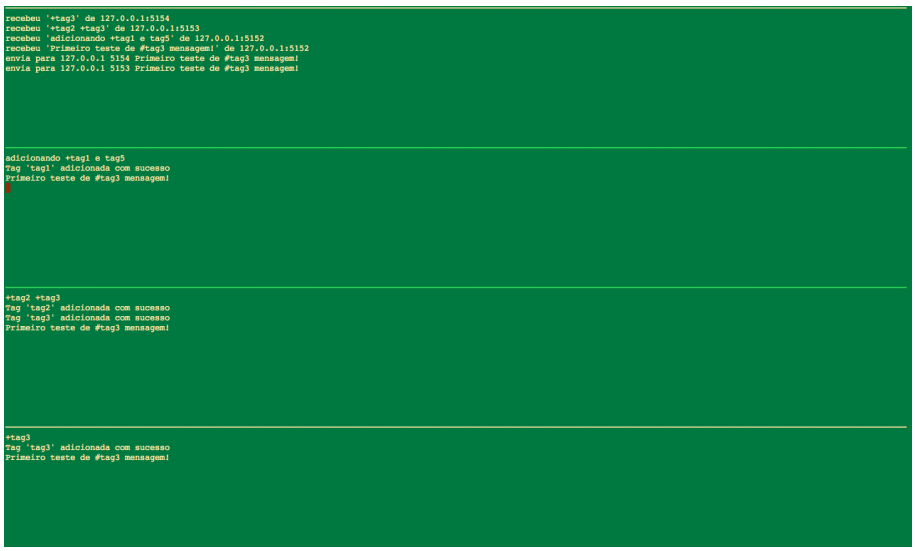

Figura 5: Teste 1: Adicionando tags que aparecem no meio de uma mensagem

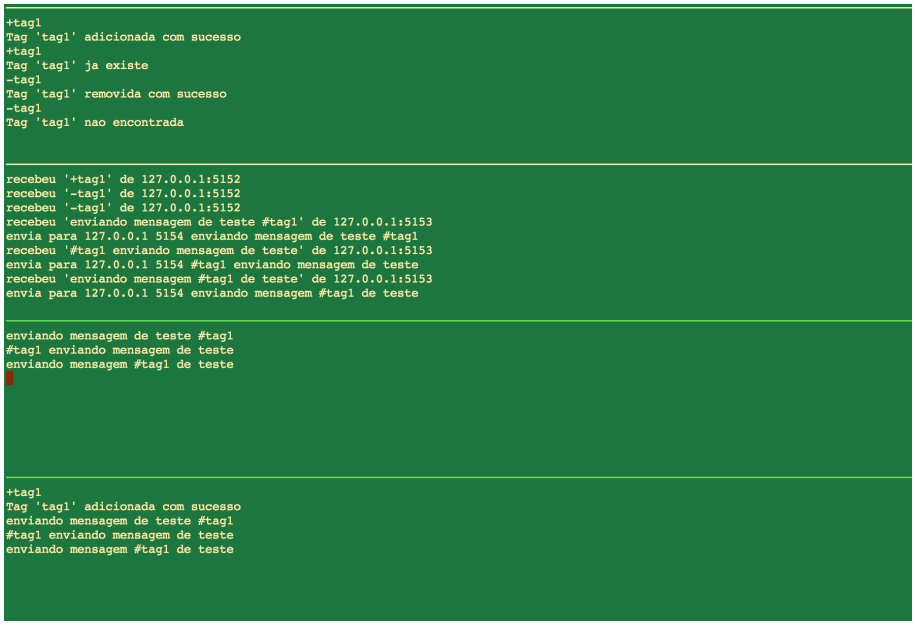

Figura 6: Teste 2: Adicionando e removendo tags

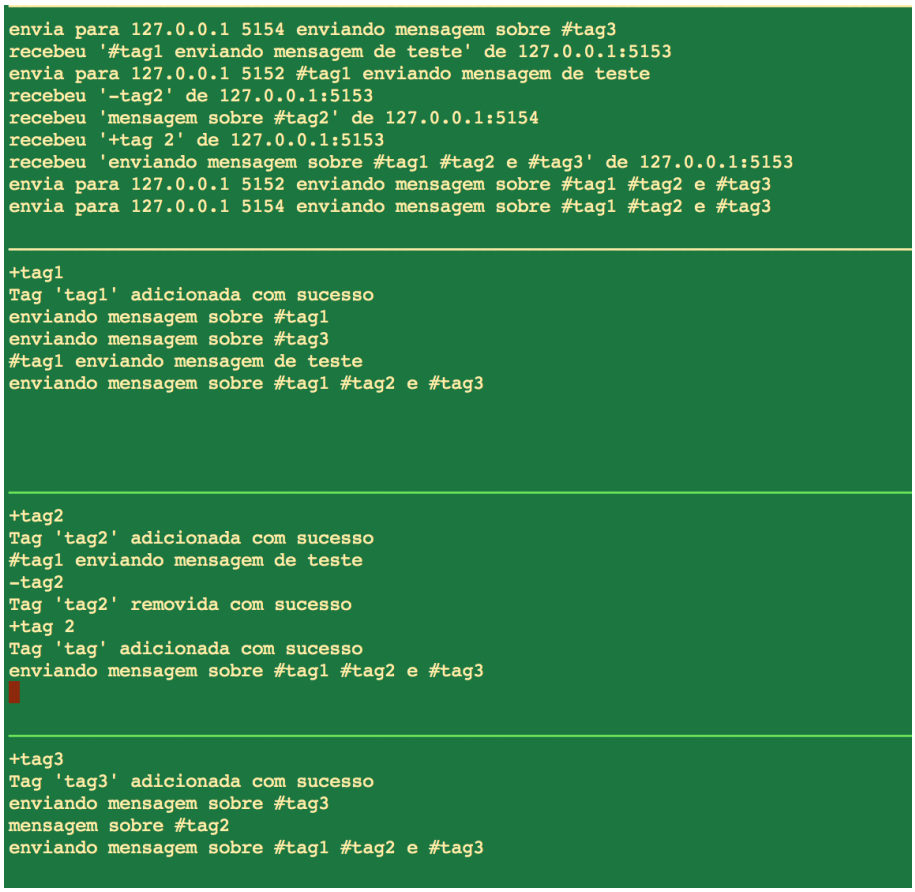

Figura 7: Teste 3: mandando mensagens com mais de uma tag

4 Conclusão

O presente trabalho prático contribuiu para que conceitos respectivos ao funcionamento do protocolo UDP e do *select* fossem colocados em prática, contribuindo para a nossa compreensão no assunto. Não foram encontradas muitas dificuldades de implementação e a documentação esclareceu bem as decisões tomadas em relação a implementação do trabalho. Os testes foram realizados utilizando o *tmux* e ocorreram sem nenhum problema, conforme é evidenciado na seção de testes.

Referências

[1] "Threading - Manage Concurrent Operations Within a Process." Datetime – Date/Time Value Manipulation - Python Module of the Week, pymotw.com/3/threading/.