Servidor de Mensagens Publish/Subscribe

Execução: Individual
Data de entrega: 30 de maio de 2022

Introdução

Neste trabalho iremos implementar um servidor e clientes para troca de mensagens de forma similar ao serviço da plataforma Twitter. Clientes enviam mensagens para o servidor informando em quais mensagens estão interessados. O servidor recebe mensagens de clientes e repassa cada mensagem para todos os clientes que estão interessados naquela mensagem. O tópico de uma mensagem é definido pelos *tags* que ela contém. Cada cliente envia ao servidor em quais *tags* está interessado, e o servidor irá repassar ao cliente todas as mensagens que contém pelo menos uma *tag* de seu interesse. Esse paradigma de comunicação é conhecido como *publish/subscribe*.

Protocolo

Servidores e clientes trocam mensagens curtas de até 500 bytes usando o protocolo TCP. Mensagens carregam texto codificado segundo a tabela ASCII. Apenas letras, números, os caracteres de pontuação `,.?!:;+-*/=@#$%()[]{}` e espaços podem ser transmitidos. (Caracteres acentuados não podem ser transmitidos.)

Clientes informam ao servidor que estão interessados em receber mensagens com um *tag* enviando uma mensagem contendo o caractere `+` (mais) seguido do identificador do *tag*. Clientes podem informar ao servidor que não estão mais interessados num *tag* enviando uma mensagem contendo o caractere `-` (menos) seguido do identificador da *tag*. Identificadores de *tag* são qualquer sequência de letras (sem números e sem pontuação). O servidor deve confirmar mensagens de declaração de interesse "`+tag`" com uma mensagem de texto contendo "`subscribed +tag`"; de forma similar, o servidor deve confirmar mensagens de declaração de desinteresse "`-tag`" com uma mensagem de texto contendo "`unsubscribed -tag`". Por exemplo, abaixo segue um exemplo de comunicação do cliente com o servidor, onde linhas começando com > são enviadas pelo cliente e linhas começando com < foram recebidas do servidor:

```
> +dota
< subscribed +dota
> -overwatch
< unsubscribed -overwatch
```

Clientes podem enviar mensagens com um *tag* colocando um caractere `#` (tralha) seguido do identificador do *tag*. Uma mensagem pode ter mais de um *tag*. O servidor replica uma mensagem para todos os clientes que informaram interesse em qualquer um dos *tags* de uma mensagem. Por exemplo, o cliente acima poderia poderia receber as seguintes mensagens:

```
< bloodcyka noob hero #dota
```

```
< perdeu eh culpa do suporte #dota #overwatch
```

Caso um cliente informe interesse em um *tag* pro qual já tenha informado interesse antes, o servidor deve responder com uma mensagem contendo "`already subscribed +tag`". Caso um cliente declare desinteresse por uma *tag* pra qual não tenha informado interesse, o servidor deve responder com uma mensagem "`not subscribed -tag`". A sequência abaixo exemplifica esses casos:

```
> +dota
< subscribed +dota
> +dota
< already subscribed +dota
> -dota
< unsubscribed -dota
> -dota
< not subscribed -dota
```

Uma mensagem sem um *tag* pode ser enviada ao servidor, mas não será repassada a nenhum cliente.

Detalhes de implementação do protocolo:

- O servidor deve enviar no máximo uma cópia de cada mensagem a um cliente independente do número de *tags* na mensagem nas quais o cliente informou interesse.
- As mensagens de interesse (`+tag`) e desinteresse (`-tag`) devem ter o sinal (`+` ou `-`) no primeiro caractere e apenas um *tag*, sem nenhum texto adicional.
- O servidor deve esquecer todas as *tags* de interesse de um cliente quando o cliente desconectar-se do sistema. (Em outras palavras, quando um cliente conecta-se ao servidor, ele precisa enviar para os servidor em quais *tags* tem interesse.)
- *Tags* e especificações de interesse e desinteresse devem estar precedidas e sucedidas por espaço, início da mensagem, ou término da mensagem. Em outras palavras, um string como "`#dota#overwatch`" não será considerado como *tags*.
- As mensagens são terminadas com um caractere de quebra de linha '`\n`'.
- O servidor deve descartar mensagens com caractere(s) inválido(s). O servidor pode desconectar o cliente que enviou a mensagem com caractere inválido, mas precisa continuar a operação sem impacto para os demais clientes conectados. Em outras palavras, o servidor não deve fechar ao receber uma mensagem com caracteres inválidos. Clientes podem simplesmente abortar operação caso recebam mensagens com caractere(s) inválido(s) do servidor.
- Para funcionamento do sistema de correção semi-automática (descrito abaixo), seu servidor deve fechar todas as conexões e terminar execução ao receber uma mensagem contendo apenas "`##kill`" de qualquer um dos clientes.

Como especificado acima, mensagens podem ter até 500 bytes e o **fim de uma mensagem é identificado com um caractere '\n'**. Uma mensagem não pode ultrapassar 500 bytes (i.e., um caractere '\n' deve aparecer entre os primeiros 500 bytes). Caso essas condições sejam violadas, o servidor pode inferir que há um *bug* no cliente e desconectá-lo.

Qualquer incoerência ou ambiguidade na especificação deve ser apontada para o professor; se confirmada a incoerência ou ambiguidade, o aluno que a apontou receberá um ou dois pontos extras dependendo da gravidade da incoerência ou ambiguidade.

Implementação

Você deve implementar uma versão do servidor e uma versão do cliente. Seu servidor deve rastrear quais clientes estão interessados em quais *tags* e repassar mensagens. O servidor e o cliente devem utilizar o protocolo TCP, criado com [socket(AF_INET, SOCK_STREAM, 0)], para comunicação.

Seu cliente deve receber mensagens do teclado e imprimir as mensagens recebidas na tela. O cliente precisa ser capaz de ler mensagens do teclado e receber mensagens da rede simultaneamente. Para isso, você pode utilizar a função [select] ou múltiplas *threads* para ler dados da entrada padrão [stdin, teclado] e do soquete de rede ao mesmo tempo.

O servidor deve suportar um número arbitrário de clientes. Os clientes podem enviar mensagens a qualquer momento, e o servidor não deve adicionar atrasos desnecessários ao repasse de mensagem. Seu servidor deve utilizar a função [select] ou múltiplas *threads* para ler dados de vários soquetes de rede ao mesmo tempo. O servidor deve imprimir na saída padrão todas as mensagens recebidas de clientes.

Seu servidor deve receber um número de porta na linha de comando especificando em qual porta ele vai receber conexões. Seu cliente deve receber o endereço IP e a porta do servidor para estabelecimento da conexão. Exemplo de execução dos programas em dois terminais distintos:

```
no terminal 1: ./servidor 51511
no terminal 2: ./cliente 127.0.0.1 51511
```

O servidor pode dar bind em todos os endereços IP associados às suas interfaces usando a constante `INADDR_ANY`. O seu programa deve funcionar com os protocolos IPv4 e IPv6.

Avaliação

Este trabalho deve ser realizado individualmente e deve ser implementado em Python3 utilizando a biblioteca Sockets (interface POSIX de soquetes de rede). Seu programa deve rodar no sistema operacional Linux e, em particular, não deve utilizar bibliotecas do Windows. Seu programa deve interoperar com qualquer outro programa implementando o mesmo protocolo (você pode testar com as implementações dos seus colegas). Procure escrever seu código de maneira clara, com comentários pontuais e bem indentados; isto facilita a correção dos monitores e tem impacto positivo na avaliação.

Correção Semi-automática

Seu servidor será corrigido de forma semi-automática por uma bateria de testes. Cada teste testa uma funcionalidade específica do servidor. O seu servidor será testado por um cliente implementado pelo professor com funcionalidades adicionais para realização dos testes. Os testes avaliam a aderência do seu servidor ao protocolo de comunicação inteiramente através dos dados trocados através da rede (a saída do seu servidor na tela, e.g., para depuração, não impacta os resultados dos testes).

O cliente implementado pelo professor para realização dos testes bem como um teste de exemplo estão disponíveis no Portal Didático para que você possa testar a compatibilidade de seu servidor com o ambiente de testes.

Pelo menos os seguintes testes serão realizados:

- Notificações do servidor para de interesse e desinteresse em *tags*.
- Recebimento de mensagens particionadas em múltiplas partes (mensagem recebida parcialmente no primeiro [recv]).
    - Este teste cobre o caso de uma mensagem com, por exemplo, 20 caracteres, que é enviada em dois pacotes. Neste teste, o servidor pode receber os primeiros 10 caracteres da mensagem em um `recv` e receber os últimos 10 caracteres da mensagem em um `recv` subsequente. É tarefa do servidor detectar que os primeiros 10 caracteres não possuem o `'\n'`, determinar que a mensagem ainda não chegou por completo, e chamar `recv` novamente até terminar de receber a mensagem para então processá-la.
- Recebimento de múltiplas mensagens em uma única chamada ao [recv].
    - Note que a sequência de bytes
      “`boa tarde #MaisUmDia bom almoço #DiarioAlimentar`**`\n`**”
      contém uma mensagem, enquanto a sequência de bytes
      “`boa tarde #MaisUmDia`**`\n`**`bom almoço #DiarioAlimentar`**`\n`**”
      contém duas mensagens. Este teste cobre o segundo caso acima. Uma sequência de bytes, que pode ser lida em um único `recv`, contendo duas

mensagens será enviada ao servidor. O servidor deve processar as duas mensagens corretamente.

- Envio de uma mensagem a múltiplos clientes interessados em seus *tags*.

Note que apesar do programa cliente não ser avaliado no conjunto de testes, ele ainda será avaliado manualmente.

Entrega

Cada aluno deve entregar documentação em PDF de até 4 páginas (duas folhas), sem capa, utilizando fonte tamanho 10, e figuras de tamanho adequado ao tamanho da fonte. A documentação deve discutir desafios, dificuldades e imprevistos do projeto, bem como as soluções adotadas para os problemas.

Cada aluno deve entregar o código fonte em Python3 (ou em C com Makefile) com dois arquivos chamados: “`cliente`” e “`servidor`”.