

Trabalho Prático (T)

Identificação da Disciplina

Código da Disciplina	Nome da Disciplina	Professor	Período
CIC_4MA e EGS_4MA	Redes de Computadores e Internet	Jeremias Moreira Gomes	2024/1

1. Objetivo

O objetivo deste trabalho é capacitar o aluno a estudar e compreender conceitos envolvendo redes de computadores e internet. Utilizando o conhecimento adquirido nas aulas, o aluno deverá implementar um servidor e um cliente de troca de mensagens utilizando os princípios básicos de programação em sockets.

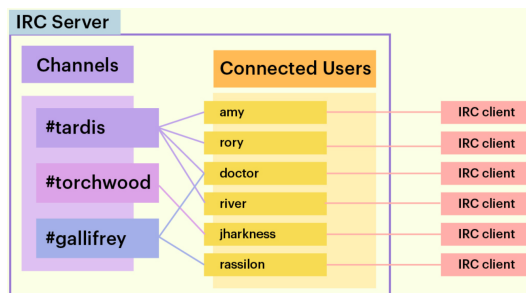
Ao cumprir os objetivos do trabalho, o aluno terá adquirido: (i) uma compreensão prática dos conceitos e características-chave de redes de computadores e internet; (ii) maior aprimoramento das suas habilidades de escrita técnica, organização de informações e comunicação eficaz, uma vez que a documentação de implementação exigirá uma apresentação clara e estruturada das principais características da rede de computadores e internet; e (iii) essa experiência proporcionará maior confiança e capacidade para explorar esses conceitos na solução de problemas computacionais.

2. Enunciado

O Instituto de Diálogo e Processamento (IDP) é uma organização sem fins lucrativos que tem como objetivo criar e desenvolver softwares que utilizem padrões abertos para a Internet.

Com a iminente problemática envolvendo a negociação e o uso abusivo de dados pessoais no de acesso a redes sociais, o IDP resolveu investir na redescoberta de mensageiros instantâneos e simplificados. Para isso, eles resolveram escolher o protocolo IRC como pontapé inicial para lançamento de um novo mensageiro instantâneo.

Como novo contratado do IDP, você recebeu a incumbência de implementar tanto o servidor quanto o cliente do mensageiro. O servidor funciona ininterruptamente, aceitando conexões de clientes e distribuindo mensagens entre usuários em canais. Já o cliente, por sua vez, é aquele que o usuário final irá baixar e dará acesso à rede de canais e mensagens.



Apesar do uso de sistemas de mensagens terem se modernizados e migrado para plataformas mais robustas e interativas, o protocolo IRC ainda é bastante utilizado em ambientes corporativos e em comunidades de

software livre. Ele permite a criação de canais de comunicação, onde usuários podem se conectar e trocar mensagens em tempo real de maneira simples e eficiente.

3. Protocolo IRC

O protocolo IRC (Internet Relay Chat) é um protocolo de comunicação baseado em texto, que permite a troca de mensagens entre usuários conectados a um servidor. Ele foi um dos primeiros protocolos de comunicação em tempo real a ser amplamente utilizado na Internet, e ainda é bastante utilizado principalmente em comunidades de software livre.

A base de implementação deste protocolo é regido por duas RFCs: a RFC 2812, especificamente sobre o protocolo do cliente, e a RFC 2813, que trata do protocolo do servidor. Por padrão, o servidor IRC funciona na porta 6667, e trabalha de maneira assíncrona, ou seja, ele aceita conexões de diferentes clientes e os trata de maneira independente e ao mesmo tempo. Assim, serão implementadas versões simplificadas desses protocolos, que serão detalhadas nas seções a seguir.

3.1. Servidor

O servidor funciona como um intermediário entre os clientes, recebendo mensagens de um cliente e enviando para os outros clientes conectados. Ele funciona por padrão na porta 6667, e aceita conexões de clientes utilizando o protocolo TCP.

Toda mensagem trocada possui uma única linha de texto, que é composta por **um comando**, **seus parâmetros** e **deve ser finalizada por \r\n**. Ao receber uma linha, o servidor deve **processar a mensagem e enviar uma resposta para o cliente que a enviou**.

3.1.1. NICK

Ao abrir uma nova conexão, o cliente envia para o servidor duas mensagens: uma de NICK e outra de USER. A primeira mensagem informa o nome de usuário que o cliente deseja utilizar, e a segunda informa o nome real do usuário. O formato das mensagens é o seguinte:

```
NICK <username>  
USER <username> 0 = :realname
```

As possíveis respostas do servidor para essas mensagens são:

- 432 * <username> :Erroneous Nickname - Quando o nome de usuário for inválido.
- 433 * <username> :Nickname is already in use - Quando o nome de usuário já estiver em uso por outro cliente.
- 001 <username> :Welcome to the Internet Relay Network <username> - Quando o nome de usuário for aceito.

O formato de nick aceito (especificamente para esta atividade), deverá **iniciar por uma letra**, pode conter em seu conteúdo letras, números e o caractere _ (sublinhado), e ter no máximo 9 caracteres. **O servidor deve ignorar a diferença entre letras maiúsculas e minúsculas**.

A mensagem do tipo 001 que é enviada pelo servidor é o indicativo de que o cliente foi aceito e está conectado ao servidor. Além disso, a mensagem `NICK` também é utilizada para alterar o nome de usuário de um cliente já conectado. O formato da mensagem permanece o mesmo, e a resposta do servidor é a seguinte:

```
:<nick_antigo> NICK <novo_nick>
```

Além do formato de resposta, um detalhe importante é que o servidor deve informar a todos os clientes conectados que encontram-se no mesmo canal que o cliente que alterou o nome de usuário (inclusive o próprio usuário), que o nick foi alterado.

3.1.2. MOTD

A mensagem do tipo 001 que é enviada pelo servidor é o indicativo de que o cliente foi aceito e está conectado ao servidor. A partir desse momento, o servidor envia para o usuário a mensagem do dia (MOTD), que é uma mensagem informativa sobre o servidor. O formato da mensagem é:

```
:server 375 <username> :- <host> Message of the Day -  
:server 372 <username> :- <message>  
:server 376 <username> :End of /MOTD command.
```

A mensagem do tipo 375 indica o início da mensagem do dia, a mensagem do tipo 372 é a mensagem do dia, que pode aparecer múltiplas vezes para cada linha da mensagem, e a mensagem do tipo 376 indica o fim da mensagem do dia.

A partir desse momento, o cliente está conectado ao servidor e pode enviar e receber mensagens de outros clientes.

3.1.3. PING

Periodicamente, tanto o servidor quanto o cliente devem estar aptos a responder a mensagens do tipo `PING`. Essas mensagens são utilizadas para verificar se o host ainda está conectado. O formato da mensagem é:

```
PING :<payload>
```

O servidor deve responder com uma mensagem do tipo `PONG`, que possui o mesmo *payload* da mensagem `PING`. O formato é o seguinte:

```
PONG :<payload>
```

Observação: como a implementação não será necessariamente assíncrona, atente-se para verificar se há mensagens de `PING` sempre que for tratar algum novo comando, ou algum outro evento.

3.1.4. JOIN

A mensagem `JOIN` é utilizada para que um cliente entre em um canal. O formato da mensagem é:

JOIN #<canal>

O nome do canal deve começar com o caractere #, pode possuir letras, números e o caractere _ (sublinhado), e ter no máximo 63 caracteres, ignorando a diferença entre letras maiúsculas e minúsculas. A resposta do servidor pode ser uma das seguintes:

- :host 403 <username> #<canal> :No such channel - Quando o canal não existir, ou o nome do canal for inválido.
- :<username> JOIN :#<canal> - Quando o cliente entra no canal.

A mensagem de entrada no canal também deve ser enviada para todos os clientes que estiverem nesse canal (inclusive o próprio). Além disso, o cliente deverá receber duas mensagens do tipo 353 e 366, que são utilizadas para listar os usuários do canal e indicar o fim da lista, respectivamente. O formato das mensagens é:

```
:<host> 353 <username> = #<canal> :<user_1> <user_2> ... <user_n>
:<host> 366 <username> #<canal> :End of /NAMES list.
```

3.1.5. PART

A mensagem PART é utilizada para que um cliente saia de um canal. O formato da mensagem é:

PART #<canal>

Caso o cliente não esteja no canal que está tentando sair, o servidor deve responder com a mensagem 442, cujo o formato é o seguinte:

```
:<host> 442 <username> #<canal> :You're not on that channel
```

Já se o cliente estiver no canal, o servidor deve responder com a mensagem PART, que indica que o cliente saiu do canal, para todos os clientes que estiverem no canal. O formato da mensagem é:

```
:<username> PART #<canal> :<motivo>
```

Nessa mensagem, <motivo> é uma mensagem opcional que o cliente pode enviar para explicar o motivo da saída do canal.

3.1.6. QUIT

A mensagem QUIT é utilizada para que um cliente saia do servidor. O formato da mensagem é:

QUIT :<motivo>

O servidor deverá encaminhar para todos os clientes conectados a mensagem QUIT, que indica que o cliente saiu do servidor, no seguinte formato:

```
:<username> QUIT :<motivo>
```

Nessa mensagem, <motivo> é uma mensagem opcional que o cliente pode enviar para explicar o motivo da saída do servidor.

3.1.7. PRIVMSG

A mensagem `PRIVMSG` é utilizada para que um cliente envie uma mensagem privada para outro cliente ou para um canal. Nesta atividade, o cliente enviará mensagens somente para canais, visto que em servidores públicos, mensagens privadas somente são autorizadas entre usuários autenticados. O formato de envio de mensagens para canais é o seguinte:

```
PRIVMSG #<canal> :<mensagem>
```

Nessa mensagem, o servidor não responde ao cliente que enviou a mensagem. Este apenas encaminha a mensagem para todos os clientes que estiverem no canal, no seguinte formato:

```
:<username> PRIVMSG #<canal> :<mensagem>
```

Onde `<mensagem>` é a mensagem enviada pelo cliente.

3.1.8. NAMES

A mensagem `NAMES` é utilizada para listar os usuários de um canal. O formato da mensagem é:

```
NAMES #<canal>
```

O servidor deve responder com uma mensagem do tipo 353, que lista os usuários do canal, e uma mensagem do tipo 366, que indica o fim da lista. O formato das mensagens é similar ao já apresentado na seção `JOIN`.

3.1.9. Orientações Específicas de Implementação do Servidor

O arquivo `servidor-base.py`, que encontra-se ao final deste documento, deve ser utilizado como base para a implementação do servidor. Ele contém duas classes, `Cliente` e `Servidor`. Nessas classes, você poderá acrescentar métodos e propriedades de acordo com a necessidade da implementação, atentando-se para aqueles métodos que não podem ser modificados (métodos `listen` e `start`), pois já estão corretos para receber a conexão de múltiplos clientes.

A classe `Cliente` deverá conter as informações sobre a conexão de um cliente, bem como métodos para envio e recebimento de mensagens. O envio de mensagens não deve exceder 512 bytes, exigindo que o método faça o tratamento necessário para que a mensagem seja particionada e enviada corretamente. Além disso, deve-se atentar na hora de implementar o método `receber_dados` pois, ao lidar com `sockets`, o funcionamento pela rede não garante que a mensagem será recebida de uma vez só, ou que cada mensagem irá conter somente um comando. Dadas as características da rede, é possível que mensagens cheguem de maneira similar aos exemplos abaixo:

```
linha\r\n  
linh  
a\r\n  
linha\r\nlinha\r\nlin  
a\r\n
```

O servidor deverá funcionar corretamente para todos esses casos apresentados, tratando comandos e guardando o restante da mensagem para a próxima iteração. Utilize alguma propriedade da classe `cliente` para guardar essa informação residual.

Por último, como a classe `Servidor` é responsável por manipular múltiplas conexões, qualquer necessidade de uma estrutura de dados para realizar o controle de objetos, deverá utilizar as estruturas `Queue` ou `deque` dos módulos `queue` e `collections`. Isso se dá pela necessidade de garantir a integridade dos dados, visto que essas estruturas são consideradas `thread-safe`, e o tratamento de problemas oriundos dessas peculiaridades não é o foco desta atividade ou disciplina.

3.2. Cliente

O cliente é a interface do usuário com o servidor, e é responsável por receber comandos do usuário, e enviar e receber mensagens do servidor. Ele se conecta ao servidor utilizando o protocolo TCP, abrindo uma única conexão por vez.

Comandos do usuário são tratados e enviados para o servidor. Comandos incorretos podem ser tratados, de acordo com a especificação apresentada abaixo, ou então ignorados. **Toda mensagem enviada pelo cliente deve ser finalizada por `\r\n`.** Ao receber uma mensagem do servidor, o cliente deve processar a mensagem e exibir o conteúdo para o usuário, quando necessário.

A seguir serão apresentados os comandos que o cliente deve ser capaz de processar.

3.2.1. `/nick`

O comando `/nick` é utilizado para alterar o nome de usuário do cliente. O formato do comando é:

```
/nick <username>
```

Caso o cliente já esteja conectado ao servidor, o comando deve ser enviado para o servidor, utilizando o seguinte formato:

```
NICK :<username>
```

Lembre-se que, no cliente, é possível tratar o nick do usuário (saber se o mesmo está de acordo) antes de enviar a mensagem para o servidor.

3.2.2. `/connect`

O comando `/connect` é utilizado para conectar o cliente ao servidor. O formato do comando é:

```
/connect <IP>
```

Esse comando é o indício para abrir uma conexão do tipo TCP com o servidor, utilizando o endereço IP fornecido, na porta 6667.

Após a abertura da conexão, o cliente deve enviar uma mensagem de `NICK` e `USER` para o servidor, informando o nome de usuário e o nome real do cliente. O formato é o já apresentado na seção do servidor. Lembre-se de, após enviar essas mensagens, tratar o recebimento da mensagem do dia.

3.2.3. /disconnect

O comando `/disconnect` é utilizado para desconectar o cliente do servidor. O formato do comando é:

```
/disconnect :<motivo>
```

O cliente deve enviar a mensagem `QUIT` para o servidor, com o motivo da desconexão. O formato é o seguinte:

```
QUIT <motivo>
```

3.2.4. /quit

O comando `/quit` é utilizado para sair do cliente. O formato do comando é:

```
/quit :<motivo>
```

O cliente deve encerrar a conexão com o servidor e finalizar a execução.

3.2.5. /join

O comando `/join` é utilizado para entrar em um canal. O formato do comando é:

```
/join #<canal>
```

O cliente deve enviar a mensagem `JOIN` para o servidor, com o nome do canal. O formato é o seguinte:

```
JOIN #<canal>
```

O servidor irá responder mensagens de diferentes códigos, que deverão ser tratadas pelo cliente, de acordo com o apresentado na seção do servidor.

3.2.6. /leave ou /part

O comando `/leave` é utilizado para sair de um canal. O formato do comando é:

```
/leave #<canal> <motivo>
```

Onde `<motivo>` é uma mensagem opcional que o cliente pode enviar para explicar o motivo da saída do canal.

O cliente deve enviar a mensagem `PART` para o servidor, com o nome do canal e o motivo da saída. O formato é o seguinte:

```
PART #<canal> <motivo>
```

3.2.7. /channel

Para facilitar o uso do cliente pelo usuário, o comando `/channel` é utilizado para selecionar um canal como padrão, quando o usuário estiver participando de dois ou mais canais simultaneamente. O formato do comando é:

```
/channel #<canal>
```

Onde `#<canal>` é um parâmetro opcional. Caso o parâmetro esteja omitido, o cliente deve listar os canais que o usuário está participando. Caso o parâmetro esteja presente, o cliente deve alterar o canal padrão para o canal informado.

3.2.8. /list

O comando `/list` é utilizado para listar os usuários presentes em um canal. O formato do comando é:

```
/list #<canal>
```

Onde `#<canal>` é um parâmetro opcional. Caso o parâmetro esteja omitido, o cliente deve listar os usuários do canal padrão. Caso o parâmetro esteja presente, o cliente deve listar os usuários do canal informado apenas se o usuário estiver participando desse canal.

3.2.9. /msg

O comando `/msg` é utilizado para enviar uma mensagem para um canal. O formato do comando é:

```
/msg #<canal> <mensagem>
```

O cliente deve enviar a mensagem `PRIVMSG` para o servidor, com o nome do canal e a mensagem. O formato é o seguinte:

```
PRIVMSG #<canal> :<mensagem>
```

No comando `/msg`, o comando e o canal poderão ser omitidos juntos, onde, na presença de somente uma mensagem, o cliente deverá enviar esta para o canal padrão.

3.2.10. /help

O comando `/help` é utilizado para listar os comandos disponíveis para o usuário. O formato do comando é:

```
/help
```

O cliente deve exibir uma lista de comandos disponíveis com uma descrição sumária de cada um.

3.2.11. PING

Além dos comandos citados, periodicamente, tanto o servidor quanto o cliente devem estar aptos a responder a mensagens do tipo `PING`. Essas mensagens são utilizadas para verificar se o host ainda está conectado. O formato da mensagem é:

```
PING :<payload>
```

O cliente deve responder com uma mensagem do tipo `PONG`, que possui o mesmo *payload* da mensagem `PING`. O formato é o seguinte:

```
PONG :<payload>
```

Observação: atente-se para responder a mensagem `PING` mesmo que estiver tratando outros eventos.

3.2.12. PRIVMSG

A mensagem `PRIVMSG` é utilizada para que um cliente envie uma mensagem privada para outro cliente ou para um canal. Nesta atividade, o cliente receberá mensagens de canais o qual está participando. O formato de recebimento de mensagens para canais é o seguinte:

```
:<username> PRIVMSG #<canal> :<mensagem>
```

Onde `<username>` é o nome do usuário que enviou a mensagem, e `#<canal>` é o nome do canal. O cliente deve exibir a mensagem para o usuário.

Observação: levando em consideração o alarme de eventos para entrada do usuário, as mensagens somente serão recebidas ou quando o usuário enviar um comando ou quando o alarme (ver seção de orientações do cliente) for acionado.

3.2.13. NAMES

Uma mensagem do tipo `NAMES` pode ser enviada para o servidor, para listar os usuários de um canal. O formato da mensagem é:

```
NAMES #<canal>
```

A resposta do servidor é similar ao já apresentado na seção `NAME` do servidor.

Observação: não há um momento específico para enviar essa mensagem, visto que o previsto é um funcionamento assíncrono. Assim, uma sugestão é que a mesma seja enviada quando o usuário solicitar a execução de algum outro comando, como por exemplo o envio de alguma mensagem. Dessa forma, a lista atualizada de usuários estará sempre disponível para ele.

3.2.14. Orientações Específicas de Implementação do Cliente

O arquivo `cliente-base.py`, que encontra-se ao final deste documento, deve ser utilizado como base para a implementação do cliente. Ele contém uma classe `Cliente` que possui métodos para conexão com o servidor. Você poderá acrescentar métodos e propriedades de acordo com a necessidade da implementação.

O método `executar` é aquele que irá coletar os comandos do usuário. Como auxílio, foi disponibilizado um interruptor de entrada, utilizando o alarme do sistema, para que outros eventos possam ser tratados periodicamente, enquanto o usuário não envia comandos.

4. Implementação

A implementação do trabalho deve ser feita em Python 3.8 ou superior. As soluções devem ser divididas entre servidor e cliente, e devem ser implementadas em arquivos separados, utilizando como base os arquivos disponibilizados. Além disso, os testes e validação do trabalho, tanto do cliente quanto do servidor, serão feitos em um ambiente Linux, similar ao disponibilizado no ambiente virtual da disciplina.

O arquivo principal do servidor (pode ser que haja mais de um arquivo) deve se chamar `servidor.py`. A execução do servidor deve ser feita por meio do seguinte comando:

```
python3 servidor.py
```

De maneira análoga, o arquivo principal do cliente (pode ser que haja mais de um arquivo) deve se chamar `cliente.py`. A execução do cliente deve ser feita por meio do seguinte comando:

```
python3 cliente.py
```

Ao executar o cliente, este deverá exibir uma mensagem de boas-vindas e aguardar a entrada de comandos do usuário. O cliente deverá então se conectar ao servidor apenas quando o usuário enviar o comando `/connect` (com os seus respectivos parâmetros de maneira correta). A figura 1 possui um exemplo de execução do servidor e do cliente em duas janelas diferentes.



```
[71670] j3r3mias@serenity:trabalho|main > python servidor.py
Servidor funcionando...

[71669] j3r3mias@serenity:trabalho|main > python cliente-simples.py
[+] Bem-vindo ao IDP IRC client!
[+] Digite /help para visualizar os comandos disponiveis
>
> █
```

Figura 1. Exemplos de execução do cliente e do servidor

4.1. Sumário

Ao final do trabalho, o aluno/grupo deverá entregar a implementação do protocolo, tanto o cliente quanto o servidor.

5. Metodologia

Este trabalho será dividido em três partes: (i) Implementação do Servidor; (ii) Implementação do Cliente; e (iii) Relatório Técnico.

5.1. Implementação do Servidor (45 pts)

Nesta etapa, os alunos deverão implementar o servidor IRC. O servidor deverá ser capaz de realizar e responder corretamente às mensagens de interação entre um ou mais clientes, gerenciando a conexão e a comunicação entre eles, conforme descrito nas seções anteriores.

5.2. Implementação do Cliente (45 pts)

Nesta etapa, os alunos deverão implementar o cliente IRC. O cliente deverá ser capaz de interagir com o usuário e conectar-se a um servidor IRC, por demanda do usuário, e enviar e receber mensagens do servidor, conforme descrito nas seções anteriores.

5.3. Documentação (10 pts)

Nesta etapa, deve-se documentar a implementação do servidor e do cliente, de forma a permitir que outras pessoas possam compreender o funcionamento dos softwares, bem como a implementação realizada. A documentação deverá ser concisa e objetiva, não deixando de ser clara, contendo no máximo **três páginas**, seguindo o modelo de artigos da Sociedade Brasileira de Computação (link).

6. Detalhes Acerca do Trabalho

6.1. Restrições deste Trabalho

Esta atividade poderá ser realizada em até **quatro integrantes**, que deverão ser informados ao professor da disciplina até 15/05/2024 (quarta-feira).

6.2. Datas Importantes

Esse trabalho poderá ser submetido a partir de 10/05/2024 (sexta-feira) - 12:00h até 16/06/2023 (domingo) - 23:55h.

6.3. Por onde será a entrega o Trabalho?

O trabalho deverá ser entregue via Ambiente Virtual (<https://ambientevirtual.idp.edu.br/>), na disciplina de Redes de Computadores e Internet. Na disciplina haverá uma atividade chamada “Trabalho Prático (T)”, para submissão do trabalho.

6.4. O que deverá ser entregue, referente ao trabalho?

Deverá ser entregue a documentação e os códigos produzidos pelos alunos, relacionado a resolução do trabalho.

6.5. Como entregar o trabalho?

A submissão das atividades deverá ser feita em um arquivo único comprimido do tipo zip (Zip archive data, at least v1.0 to extract), contendo documentação e um diretório com os códigos elaborados. Além disso, para garantir a integridade do conteúdo entregue, o nome do arquivo comprimido deverá possuir duas informações (além da extensão .zip):

- As matrículas dos alunos, separada por hífen.
- O *hash* md5 do arquivo e .zip.

Exemplo: 22001101-22011000-23010203-3b1b448e9a49cd6a91a1ad044e67fff2.zip

Para gerar o md5 do arquivo comprimido, utilize o comando `md5sum` do Linux e em seguida faça o renomeamento utilizando o *hash* coletado.

Segue um exemplo de geração do arquivo final a ser submetido no moodle, para um aluno:

```
[13930] j3r3mias@tardis:trabalho-01 > ls
trabalho-01-apc-jeremias.c
[13931] j3r3mias@tardis:trabalho-01 > zip -r aaa.zip trabalho-01-apc-jeremias.c
adding: trabalho-01-apc-jeremias.c (deflated 99%)
[13932] j3r3mias@tardis:trabalho-01 > ls
aaa.zip  trabalho-01-apc-jeremias.c
[13933] j3r3mias@tardis:trabalho-01 > ls -lha aaa.zip
-rw-rw-r-- 1 j3r3mias j3r3mias 335 out 15 16:54 aaa.zip
[13934] j3r3mias@tardis:trabalho-01 > md5sum aaa.zip
44cf46f9237cb506ebde3e9e1cd044f9  aaa.zip
[13935] j3r3mias@tardis:trabalho-01 > mv aaa.zip 160068000-44cf46f9237cb506ebde3e9e1cd044f9.zip
[13936] j3r3mias@tardis:trabalho-01 > ls -lha 160068000-44cf46f9237cb506ebde3e9e1cd044f9.zip
-rw-rw-r-- 1 j3r3mias j3r3mias 335 out 15 16:54 160068000-44cf46f9237cb506ebde3e9e1cd044f9.zip
[13937] j3r3mias@tardis:trabalho-01 > md5sum 160068000-44cf46f9237cb506ebde3e9e1cd044f9.zip
44cf46f9237cb506ebde3e9e1cd044f9  160068000-44cf46f9237cb506ebde3e9e1cd044f9.zip
[13938] j3r3mias@tardis:trabalho-01 >
```

Figura 2. Exemplo de geração de arquivo para submissão

6.6. Quem deverá entregar o trabalho?

O Ambiente Virtual disponibiliza uma opção de trabalhos em grupo, onde um membro do grupo realiza a submissão e o arquivo é disponibilizado para ambos. Caso esta opção não esteja disponível, todos os alunos deverão submeter o trabalho no Ambiente Virtual onde, **somente um aluno deverá gerar a versão final do trabalho (arquivo zip)**, para que o *hash* do arquivo não corra o risco de ficar diferente prejudicando a avaliação da entrega.

6.7. Observações importantes

- Não deixe para fazer o trabalho de última hora.
- Não serão aceitos trabalhos com atraso.
- Não deixe para fazer o trabalho de última hora.
- Cópias de trabalho receberão zero (além disso, plágio é crime).
- Não deixe para fazer o trabalho de última hora.

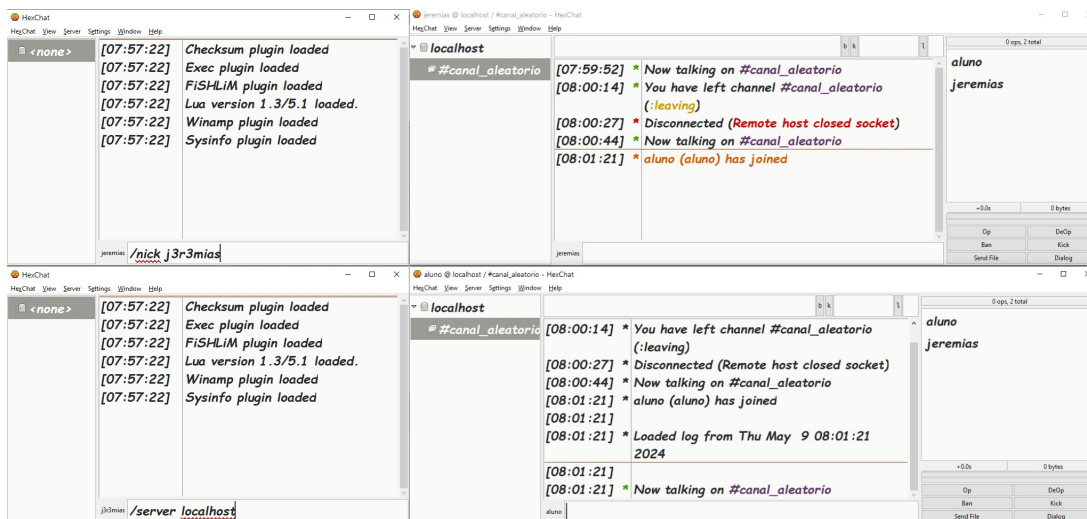



Figura 3. Exemplos de execução do HexChat

6.8. Dicas de Implementação

Para implementar o servidor, você pode utilizar como base de testes algum cliente IRC já existente como, por exemplo, o *HexChat* (link) que é um cliente IRC multiplataforma e de código aberto. Assim, você poderá testar a comunicação entre o servidor e o cliente, e verificar se as mensagens do protocolo estão sendo

Já para a implementação do cliente, você pode utilizar alguma rede pública existente para testar a comunicação entre o cliente e o servidor. Uma rede pública bastante utilizada é a *freenode* (link), cujo endereço para conexão é `irc.freenode.net:6667`, que é uma rede de IRC voltada para comunidades de software livre. Os únicos cuidados a serem tomados dizem respeito ao excessivo número de mensagens do tipo notícia que são enviadas para os clientes, as quais deverão ser descartadas pelo seu cliente. A figura 4 contém um exemplo de conexão de um cliente na rede *freenode.net*.

```
[71799] j3r3mias@serenity:trabalho$ python cliente-samples.py
[*] Bem-vindo ao IOP IRC client!
[*] Digite /help para visualizar os comandos disponiveis
> /connect irc.freenode.net
[*] Conectando em irc.freenode.net:6667
[*] Hello, World!
[*]
[*] Welcome to the
[*]
[*] 
[*]
[*]
[*]
[*]
[*] irc.freenode.net / chat.freenode.net
[*] 6666-6669
[*] 6697,7000,7070
[*] //webchat.freenode.net
[*]
[*] By connecting to freenode you indicate that you have read and accept our
[*] //freenode.net and agree to
[*] be scanned for an open proxy.
[*]
[*]
[*]
[*] #help           - Help with general IRC and Services.
[*] #chanhelp       - Help with channels and channel services.
[*] #IRC            - Help with the IRC.com by freenode app.
[*]
[*]
[*] //bbs.freenode.net/]
[*] #freenode-bbs-discussion - BBS Discussion
[*] //jobs.freenode.net/]
[*] #freenode       - Chat about freenode.
[*] #anonchan       - Chat anonymously.
[*]
[*] Type /join #<channel>
[*]
[*] Thank you for using freenode!
[*] Conectado em irc.freenode.net
> /join #canal_aleatorio
[*] Entrando em #canal_aleatorio
[*] Entrou em #canal_aleatorio
[*] Usuários em #canal_aleatorio: @j3r3mias,
[*] #canal_aleatorio > |
```

7. Bibliografia

- 14

Código Base do Cliente

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  import signal
5  import socket
6
7  class Cliente(Exception):
8      def __init__(self):
9          # Propriedades do cliente
10         # Exemplo:
11         self.conectado = False
12
13         # Exceção para alarme de tempo (não alterar esta linha)
14         signal.signal(signal.SIGALRM, self.exception_handler)
15
16         # Tratamento de exceção para alarme de tempo (não alterar este método)
17         def exception_handler(self, signum, frame):
18             raise 'EXCEÇÃO_(timeout)'
19
20         def receber_dados(self):
21             pass
22
23         def executar(self):
24             cmd = ''
25             print('Cliente!')
26
27             while True:
28                 # Espera a entrada de um comando por 20 segundos, caso contrário, se uma exceção de tempo
29                 # for disparada, deve-se verificar se há mensagens do servidor para serem lidas e tratadas
30                 # (exemplo: verificar a chegada de ping)
31                 signal.alarm(20)
32                 try:
33                     cmd = input()
34                 except Exception as e:
35                     # Nada foi digitado em 30 segundos, o que fazer?
36                     # (Exemplo: verificar se há mensagens oriundas do servidor)
37                     print()
38                     continue
39                 signal.alarm(0)
40
41                 # Um comando foi digitado. Tratar!
42                 # ...
43
44
45         def main():
46             c = Cliente()
47             c.executar()
48
49
50         if __name__ == '__main__':
51             main()

```

Código Base do Servidor

```

1  #!/usr/bin/python
2
3  import socket
4  import queue
5  import time
6  from collections import deque
7  from _thread import *
8
9  # Mensagem do Dia
10 MOTD = '''servidor'''
11
12 class Cliente:
13     def __init__(self, conn):
14         self.conn = conn
15
16     def receber_dados(self):
17         pass
18
19     def enviar_dados(self, msg):
20         pass
21
22 class Servidor:
23     def __init__(self, port=6667):
24         # Sempre que precisar de uma estrutura de dados que poderá ser acessada em diferentes conexões,
25         # utilize apenas deque ou queue. Exemplo, para armazenar as conexões ativas:
26         self.conns = deque()
27         self.port = port
28
29     def run(self, conn):
30         Cliente(conn)
31         pass
32
33     def listen(self):
34         '''(não alterar)
35         Escuta múltiplas conexões na porta definida, chamando o método run para
36         cada uma. Propriedades da classe Servidor são vistas e podem
37         ser alteradas por todas as conexões.
38         '''
39         _socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
40         _socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
41         _socket.bind(('', self.port))
42         _socket.listen(4096)
43         while True:
44             print(f'Servidor_aceitando_conexões_na_porta_{self.port}...')
45             client, addr = _socket.accept()
46             start_new_thread(self.run, (client, ))
47
48     def start(self):
49         '''(não alterar)
50         Inicia o servidor no método listen e fica em loop infinito.
51         '''
52         start_new_thread(self.listen, ())
53
54         while True:
55             time.sleep(60)
56             print('Servidor_funcionando...')
57
58
59 def main():
60     s = Servidor(host='', port=6667, debug=True)
61     s.start()
62
63 if __name__ == '__main__':
64     main()

```