

Trabalho Prático 1

Douglas W Lopes e Lucas Pereira Monteiro

11 de Novembro 2015

1 Introdução

O trabalho a seguir realiza a implementação de um chat, onde o principal objetivo é demonstrar a aplicação de cliente e servidor: onde o servidor recebe acessos simultâneos (conforme protocolo IRC-2). Mas não bastando acessos simultâneos, o servidor ajuda o cliente no controle e repasse dessas mensagens, usando comandos, como: Mute e Post, dizendo ao servidor ou cliente o que deve ser feito ou o que foi feito; esse é o protocolo IRC-2 em suma.

Os clientes enviarão Posts para outros clientes, porém na verdade um Post enviado por um cliente primeiramente passa ao servidor e de acordo com as requisições dos clientes remetentes e destinatários, ele distribui as mensagens. O que acontece é que alguns clientes podem querer ou não ouvir outros clientes, ou seja, um cliente B pode não querer ouvir uma mensagem de A; então caso A envie uma mensagem para outros presentes no Chat, o B (e quem mais tiver mutado A), não receberá a mensagem enviada de A. Essa é a rotina desafio desse trabalho prático: garantir essa entrega e esse controle por parte do servidor, cliente e usuários.

Portanto são abordados nesse projeto várias estruturas de dados e suas aplicações para garantir o funcionamento desse Chat. A linguagem de implementação é C, porém nessa documentação tem o objetivo da clareza para que qualquer linguagem pensada, possa usar nossa referência lógica, ou ao menos servir de base para tal. Além disso, esse trabalho tem como objetivo a aplicação de conhecimento adquiridos na disciplina de redes até o momento como sockets, protocolos, e etc. Também sendo aplicado conhecimentos de semestres anteriores. Segue abaixo a imagem para representação de Clientes em um Chat e seu Servidor:

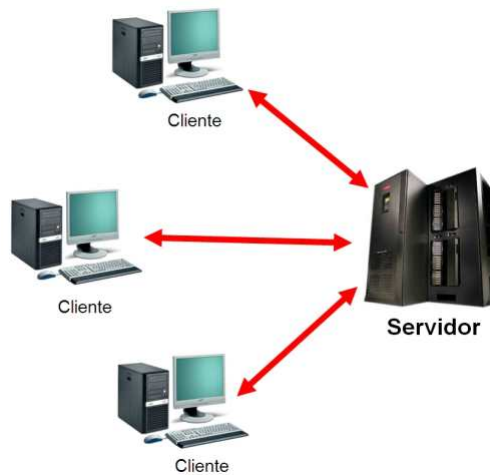


Figure 1: Representação de um Chat usando Clientes e um Servidor **Fonte:** Google

2 Solução do Problema

Abaixo apresentamos as principais abstrações, em alto nível, do principal código de cada programa: servidor e cliente. Essa abstração tem por objetivo representar a solução e que essa seja de fácil entendimento; e que seja base para uso em qualquer linguagem e para qualquer especialista ou entendedor de programação, saiba como funciona. O objetivo é entender o funcionamento geral da parte mais técnica do projeto, independente do algoritmo em qualquer linguagem de programação, ou seja, entender em alto nível através do baixo nível como o chat funciona e foi programado, junto com sua lógica.

Na próxima seção, será abordado mais detalhamento das estruturas de dados usados e seus respectivos serviços. A intenção dessa seção atual é demonstrar em alto nível funcionamento do cliente e servidor e onde está sendo aplicado as nossas estruturas para garantir os requisitos funcionais do trabalho prático.

2.1 Servidor

Nesse ponto declaramos o nosso socket que será a interface da rede servidor com o clientes. Inicializamos a porta disponível e fazemos o "bind" do socket com o endereço e porta disponíveis. E finalmente o Listen para aguardar as conexões com o os clientes. A diferença aqui entre esse trabalho prático 1 e o Trabalho prático 0: é que possuímos um vetor de sockets prontos para receber a comunicação de um conjunto de clientes.

Esse vetor de sockets que garantirá que vários clientes conseguirão conexões simultâneas com o servidor, no caso: Cliente Sockets. Tomando como in-

struções (dos professores e monitores da disciplina) e por determinar que os leitores dessa documentação já sabem como funciona o Select (para conexão simultânea de vários clientes), representarei aqui somente os lugares que foram interferidos com o nosso código para funcionamento desse trabalho prático 1.

Sabido da complexidade da estrutura Select, decidimos escrever como o nosso servidor funcionará em alto nível:

Primeiramente sempre adicionaremos novos clientes que solicitaram conexão ao vetor que guarda os sockets, declarado em select. Atualizaremos então o vetor do tipo usuário que armazena usuários e um de Ids que armazena os indexes de cada novo cliente no vetor de sockets. Ou seja temos três vetores, um que armazena os sockets, outro que armazena onde existe sockets ativos com clientes e o outro que armazena usuários.

Enfim para cada usuário novo, aguardamos seus respectivos comandos enviados pelo buffer do cliente. Sendo Nick, daremos um nome a esse usuário. Em seguida aguardamos qualquer outro comando: Post, Mute, Unmute, etc. Caso seja Post o comando no buffer, recebemos um string de até 499 caracteres (essa verificação será feita no cliente para enviar o tamanho especificado corretamente), sendo o dado maior que o valor especificado, o texto será truncado. Se o post obedecer aos requisitos, enviamos ele aos usuários que podem receber, checando a tabela de usuários mudos daquele que enviou o post para não enviar para os usuários que não querem receber a mensagem de determinado usuário.

Caso seja Mute o comando, procedemos em entrar na lista de mudo da estrutura usuários armazenados no vetor de usuários e mudamos aquele com index desejado, encontrando tal pelo nome. No caso de Unmute o funcionamento é o mesmo, a diferença que queremos desmutar.

Em caso de Close fechamos tal cliente e excluímos eles e seu index de todos os vetores citados acima.

Essa explicação é apenas de alto nível para ajudar a compreender a lógica usada para elaboração desse trabalho prático. A seguir, na próxima sessão, após o algoritmo do cliente, cada estrutura de dados utilizada será explicada e contextualizada, ajudando ainda mais no entendimento. A intenção aqui é desenhar em sua cabeça como trabalhamos com o servidor e nossas estruturas e que sim, dessa é claro o funcionamento correto do programa.

2.2 Cliente

Algorithm 1 Cliente

```
1: função SERVIDOR(IP(ou nome),Porta,Usuário) Conectar o cliente a porta
   correta do servidor,já iniciado anteriormente, já informando o usuário. Para
   tanto, iniciamos com um socket que será a interface do nosso cliente com a
   rede e nos conectaremos a esse e a partir daí temos o seguinte algoritmo:
2:   enquanto A conexão existir faça
3:     se Socket está presente para leitura então
4:       Aguardo ao comando Nick
5:       Armazeno o comando mais o Nome
6:       Envia o Buffer para o Servidor
7:     fim se
8:     se Se Socket está pronto para leitura e escrita então
9:       Aguardo comando Post
10:      Leio Post
11:      Armazeno Post no Buffer
12:      Envio o Buffer para o servidor
13:    fim se
14:  fim enquanto
15:  Feche Socket.
16: fim função
```

3 Estrutura de Dados

3.1 Servidor

Nessa etapa serão demonstradas a utilização das estruturas do lado do servidor. Em seguida será explicitados o uso dessas mesmas estruturas ou de outras usadas do lado do cliente.

3.1.1 User

Esse TAD, usado no servidor, tem como precedimentos e tipos relacionados ao usuário. O Tipo principal é o tipo usuário, que tem como campos: um index, um nick e uma lista com usuários mutados por tal usuário. O index é a posição do cliente no vetor de sockets (cliente socket - disponibilizado no código Select na página da disciplina). Ou seja, sabemos exatamente qual socket representa tal cliente.

Temos também 2 vetores básicos para o funcionamento do programa: Usuario IDs (que seta as respectivas posições que existe um cliente, de acordo com o vetor Cliente sockets (de Select)) e All Users (que é um tipo User que armazena tipos usuários do chat na respectiva posição também dado pelo Cliente Sockets). Por exemplo: se em Cliente Sockets (dado em Select via Moodle) um cliente foi setado na posição 0 (zero), em All Users será criado esse usuário com Index 0. E em Usuários IDs: será setado 1 na posição 0, sinalizando a existência de um cliente naquela posição (e 0 (zero) onde não tem).

Principais serviços de User Nesse tópico explicitaremos os principais serviços oferecidos pelo TAD User na aplicação do nosso chat, conforme abaixo:

User Create User Esse serviço cria um usuário, quando ocorre uma conexão de algum cliente com o servidor. O que essa função recebe como parâmetros é a posição do socket, no vetor de cliente sockets, que foi inicializado, i ; um Nick Vazio (por enquanto), o endereço de um vetor de Ids, onde nesse vetor, do mesmo tamanho que cliente sockets, setaremos 1 na posição i desse vetor, para saber exatamente a posição respectiva desse cliente em cliente-sockets e armazena lá para mais rápido acesso; e por fim passamos o endereço de um vetor do tipo User, para armazenar esse cliente nesse vetor, também na posição i .

Void UpdateCurrUsers Esse serviço tem por função, atuar no momento da criação de um usuário. Portanto quando um novo usuário é adicionado, atualizamos a lista de clientes mudos de todos os outros usuários do chat. Ou seja, se tenho usuário A e B conversando, quando um usuário C entra no Chat, lá na listas de mutáveis de A e B a posição referente a C nessas listas é desmutada. Lembro que essa lista de mutáveis utiliza de clientes Sockets (em Select) e usuários IDs descritas acima para saber a posição de cada cliente, e a posição é seguida de maneira respectiva na lista de mutáveis. Logo se A é um cliente na posição 0 (zero), em todas as listas de mutáveis de todos os clientes ele também está na posição 0 (Zero). Lembro também que clientes que não existem ainda estão mudos, ou seja, posições de clientes que ainda não foram inicializados estão com 1 (Mudos), nos clientes existentes, quando aquela posição é inicializada por um cliente C, recebe 0 em A e B, mostrando que agora aquele novo usuário inicialmente será ouvido por tal cliente. Bem como a lista de mutáveis desse novo cliente (C) também o permite ser ouvido por todos (A e B) inicialmente.

User RemoveUser Esse usuário remove um usuário de User Ids e da estrutura All Users, caso ocorra uma desconexão de tal usuário. Ele remove usando o Index como chave de procura.

Void UpdateRemoveUser Esse serviço atualiza a lista de mutáveis dos clientes restantes, após a saída de um cliente (usuário). Funciona de maneira similar a UpdateCurrUsers, porém ele muda a posição na lista de mutáveis, para que um cliente não converse com um cliente que não mais existe. Ou seja, retorna 1 para tal posição que representa um cliente excluído, para que o cliente que possui essa lista de mudo, não converse mais com esse.

MuteUser e unMuteUser Esses dois serviços são o coração de todo o trabalho. Como o nome diz, eles são os responsáveis por mutar ou desmutar um cliente. MuteUser recebe como parâmetro um usuário A que deseja mutar, o usuário que será mudado B e o vetor do tipo User All Users. Dessa maneira ele encontra o usuário B (dentro de all Users) e o index A dentro da lista de mutáveis de B, e coloca 1 (um) no index representado por A. Dessa forma quando B enviar um Post, A não receberá a mensagem, pois está mudo na lista de mutáveis de B.

UnmuteUser funciona de maneira similar, porém oposta: se A deseja agora escutar B, passamos os mesmos parâmetros ditos anteriormente; porém a diferença está que agora em B, vamos até sua lista de mutáveis e trocamos o index que

representa A para 0 (zero), logo A passará escutar posts de B.

3.2 Cliente

Nosso Cliente não utiliza nenhuma estrutura especial. Nosso cliente é um cliente simples, que apenas envia e recebe mensagens do Servidor. O Servidor é o único que utiliza a estrutura `User.h` explicitada acima para controle dos comandos do protocolo IRC-2.

O cliente lê os dados e armazena tudo em buffer e envia para o Servidor. Uma observação é que Post no cliente é lido usando `fgets`, para que possamos ler todos os dados de um post, inclusive espaços.

Entretanto, se o cliente tentar enviar um Nick vazio, nada acontecerá. Também nada acontece se o cliente tenta enviar um Nick que contém espaços. Se o cliente tentar enviar um Nick com mais 16 caracteres, o nick será truncado para 16 caracteres. O processo de truncar também acontece para qualquer Post se o mesmo não respeitar os 500 caracteres definidos na especificação. Entretanto, para Post os espaços em branco são permitidos. Mute e Unmute recebem apenas mais um parâmetro, o qual é o nick, portanto são tratados com tal. Close não recebe nenhuma adição, apenas o próprio comando.

4 Algumas decisões de implementação

Decisões importantes foram tomadas para esse trabalho, segue algumas:

Utilizamos vetores estáticos para a representação da lista de mutáveis, para representação dos usuários no vetor do tipo `User` e na representação de qualquer vetor utilizado no programa. Escolhemos essa representação por acreditar que seria mais simples, seria o suficiente para tratar o Trabalho e demonstrar que entendemos e chegamos ao objetivo final do trabalho. Algumas dificuldades com Listas com ponteiros em conjunto com o tempo foram fatores também relevantes para a nossa decisão.

Outro ponto importante, como pode se perceber até aqui temos basicamente três vetores principais: Clientes Sockets, Clientes Ids e All Users, onde os dois primeiros são do tipo `int` e o último do tipo `User`. O vetor Clientes Ids como explicitado acima representa as posições onde existem clientes ativos em Clientes Sockets. Escolhemos usar mais um vetor para a representação dessas posições, para que pudessemos obter uma maior velocidade para localizar determinado cliente, apesar do detrimento em complexidade de espaço.

A última decisão importante e bem relevante é a consideração que o `scanf` é um método bloqueante de leitura, então utilizamos o `Fgets` para leitura do teclado e aplicamos alguns recursos específicos da linguagem C e suas bibliotecas, para dar um "toque" mais real ao Chat. Portanto um usuário pode estar pronto para digitar determinada informação, ele simultaneamente e em tempo real pode receber as mensagens de outros usuários, mesmo que ele ainda esteja criando sua mensagem.

5 Testes

Os testes possuem algumas adições de prints para comprovar o funcionamento correto e entendimento por nossa parte. Fizemos todas as etapas da conexão de três diferentes usuários ao servidor, o processo de um mutar um ao outro, processo de desmutar, entre os outros comandos para comprovar o funcionamento adequado do nosso trabalho.

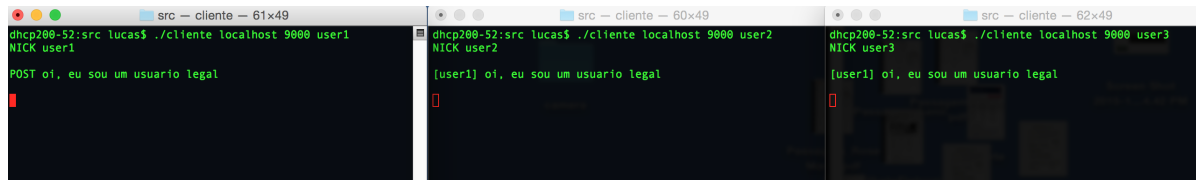


Figure 2: Essa imagem demonstra a conexão de sucesso com o servidor e com os clientes. Note que inicialmente todos os clientes conversam com todos. Perceba que os parâmetros passados do cliente estão corretos. **Fonte:** Aplicação TP1 em funcionamento.

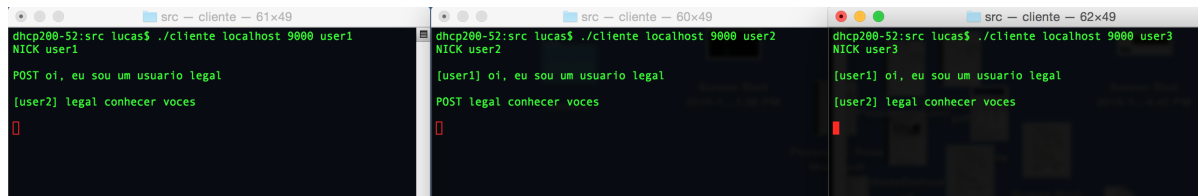


Figure 3: Essa imagem demonstra que o User2 Postou algo e todos os outros receberam a informação, pois até então não existe ninguém mutado. **Fonte:** Aplicação TP1 em funcionamento.

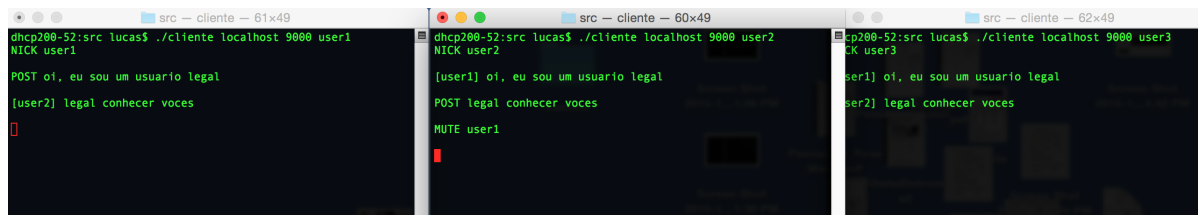
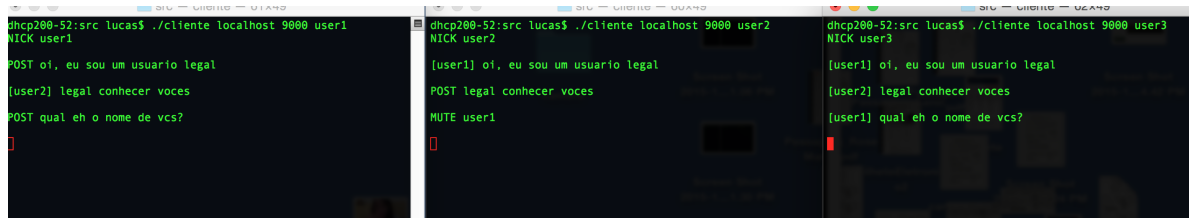


Figure 4: Mostramos aqui User2 mutando o user1. **Fonte:** Aplicação TP1 em funcionamento.

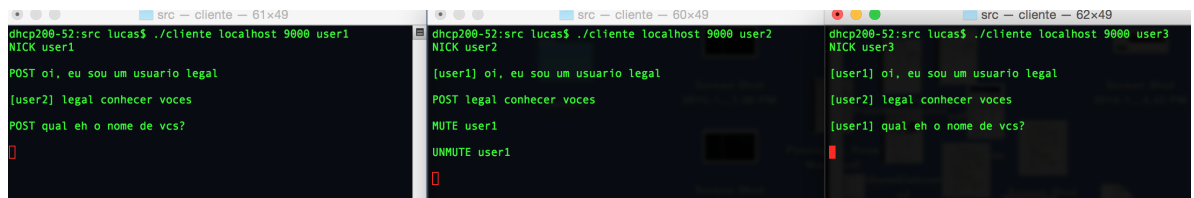


```
dhcp200-52:src lucas$ ./cliente localhost 9000 user1
NICK user1
POST oi, eu sou um usuario legal
[user2] legal conhecer voces
POST qual eh o nome de vcs?

dhcp200-52:src lucas$ ./cliente localhost 9000 user2
NICK user2
[user1] oi, eu sou um usuario legal
POST legal conhecer voces
MUTE user1

dhcp200-52:src lucas$ ./cliente localhost 9000 user3
NICK user3
[user1] oi, eu sou um usuario legal
[user2] legal conhecer voces
[user1] qual eh o nome de vcs?
```

Figure 5: Essa imagem demonstra que quando User1 se comunica, apenas User3 recebe suas mensagens. Uma vez que user2 Mutou User1. **Fonte:** Aplicação TP1 em funcionamento.

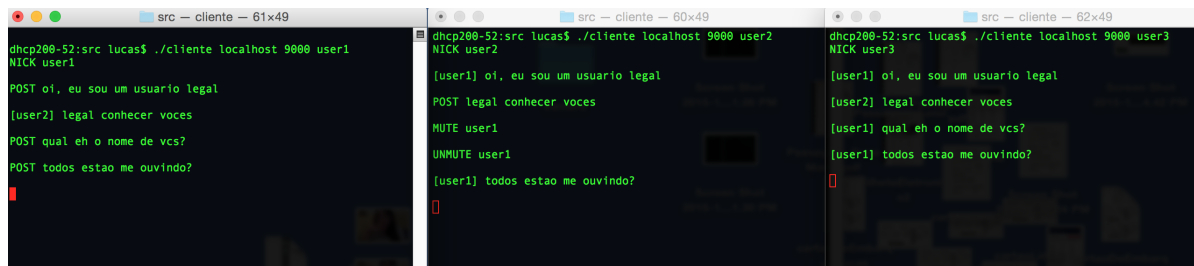


```
dhcp200-52:src lucas$ ./cliente localhost 9000 user1
NICK user1
POST oi, eu sou um usuario legal
[user2] legal conhecer voces
POST qual eh o nome de vcs?

dhcp200-52:src lucas$ ./cliente localhost 9000 user2
NICK user2
[user1] oi, eu sou um usuario legal
POST legal conhecer voces
MUTE user1
UNMUTE user1

dhcp200-52:src lucas$ ./cliente localhost 9000 user3
NICK user3
[user1] oi, eu sou um usuario legal
[user2] legal conhecer voces
[user1] qual eh o nome de vcs?
```

Figure 6: Nessa imagem temos agora a situação que em User2 desmuta user1. **Fonte:** Aplicação TP1 em funcionamento.



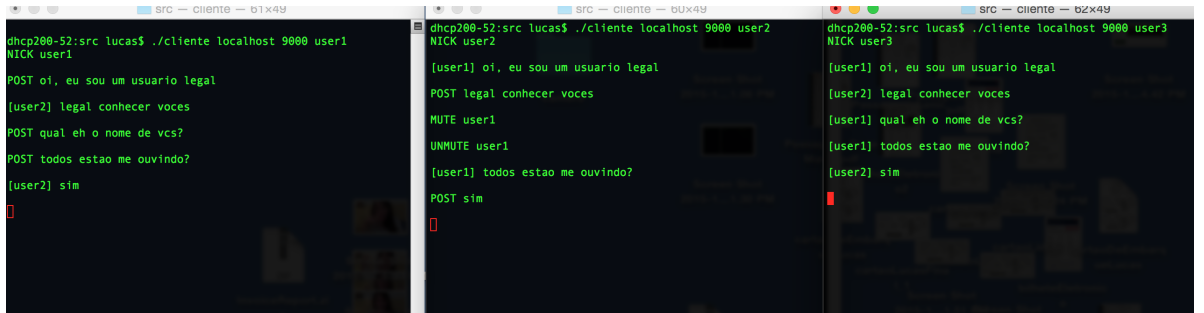
```
dhcp200-52:src lucas$ ./cliente localhost 9000 user1
NICK user1
POST oi, eu sou um usuario legal
[user2] legal conhecer voces
POST qual eh o nome de vcs?
POST todos estao me ouvindo?

dhcp200-52:src lucas$ ./cliente localhost 9000 user2
NICK user2
[user1] oi, eu sou um usuario legal
POST legal conhecer voces
MUTE user1
UNMUTE user1
[user1] todos estao me ouvindo?

dhcp200-52:src lucas$ ./cliente localhost 9000 user3
NICK user3
[user1] oi, eu sou um usuario legal
[user2] legal conhecer voces
[user1] qual eh o nome de vcs?
[user1] todos estao me ouvindo?
```

Figure 7: Note que novamente User1 envia uma nova mensagem, e dessa vez todos voltam a escutar, inclusive o User2. **Fonte:** Aplicação TP1 em funcionamento.

Frizamos novamente que esses testes tem fins didáticos e que nossa intenção é facilitar o entendimento do monitor em relação ao nosso projeto. Quaisquer eventuais dúvidas serão esclarecidas no momento da entrevista. Caso seja necessário podemos demonstrar em alguma entrevista caso haja quaisquer dúvida.

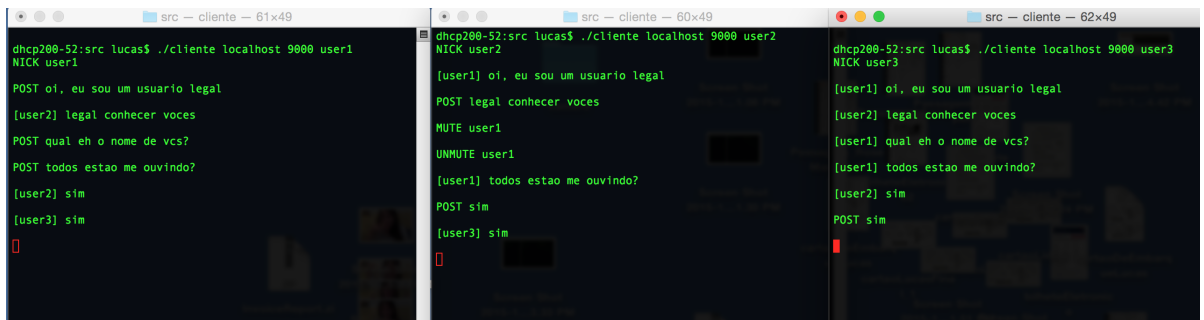


```
dhcp200-52:src lucas$ ./cliente localhost 9000 user1
NICK user1
POST oi, eu sou um usuario legal
[user2] legal conhecer voces
POST qual eh o nome de vcs?
POST todos estao me ouvindo?
[user2] sim

dhcp200-52:src lucas$ ./cliente localhost 9000 user2
NICK user2
[user1] oi, eu sou um usuario legal
POST legal conhecer voces
MUTE user1
UNMUTE user1
[user1] todos estao me ouvindo?
POST sim

dhcp200-52:src lucas$ ./cliente localhost 9000 user3
NICK user3
[user1] oi, eu sou um usuario legal
[user2] legal conhecer voces
[user1] qual eh o nome de vcs?
[user1] todos estao me ouvindo?
[user2] sim
```

Figure 8: User2 responde e como ninguém o mutou, todos os outros usuários do Chat estão ouvindo. **Fonte:** Aplicação TP1 em funcionamento.

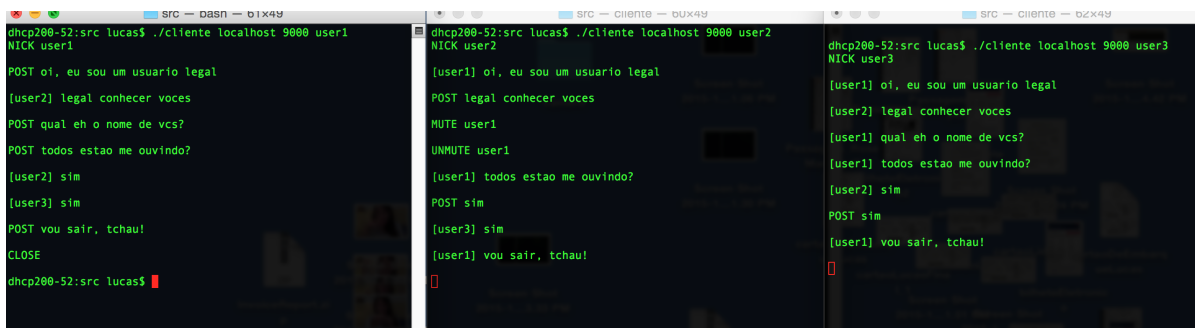


```
dhcp200-52:src lucas$ ./cliente localhost 9000 user1
NICK user1
POST oi, eu sou um usuario legal
[user2] legal conhecer voces
POST qual eh o nome de vcs?
POST todos estao me ouvindo?
[user2] sim
[user3] sim

dhcp200-52:src lucas$ ./cliente localhost 9000 user2
NICK user2
[user1] oi, eu sou um usuario legal
POST legal conhecer voces
MUTE user1
UNMUTE user1
[user1] todos estao me ouvindo?
POST sim
[user3] sim

dhcp200-52:src lucas$ ./cliente localhost 9000 user3
NICK user3
[user1] oi, eu sou um usuario legal
[user2] legal conhecer voces
[user1] qual eh o nome de vcs?
[user1] todos estao me ouvindo?
[user2] sim
POST sim
```

Figure 9: Então o user3 responde, e todos recebem sua mensagem. **Fonte:** Aplicação TP1 em funcionamento.



```
dhcp200-52:src lucas$ ./cliente localhost 9000 user1
NICK user1
POST oi, eu sou um usuario legal
[user2] legal conhecer voces
POST qual eh o nome de vcs?
POST todos estao me ouvindo?
[user2] sim
[user3] sim
POST vou sair, tchau!
CLOSE
dhcp200-52:src lucas$

dhcp200-52:src lucas$ ./cliente localhost 9000 user2
NICK user2
[user1] oi, eu sou um usuario legal
POST legal conhecer voces
MUTE user1
UNMUTE user1
[user1] todos estao me ouvindo?
POST sim
[user3] sim
[user1] vou sair, tchau!

dhcp200-52:src lucas$ ./cliente localhost 9000 user3
NICK user3
[user1] oi, eu sou um usuario legal
[user2] legal conhecer voces
[user1] qual eh o nome de vcs?
[user1] todos estao me ouvindo?
[user2] sim
POST sim
[user1] vou sair, tchau!
```

Figure 10: User1 decide Sair e fechar sua sessão. Note que ele Posta acisando que vai sair, comanda Close e sua sessão foi encerrada. **Fonte:** Aplicação TP1 em funcionamento.

6 Conclusões

Esse trabalho prático teve como grande desafio construir um servidor que suporta múltiplos clientes simultâneos e além disso gerencia as mensagens entre

```
src - bash - 61x49
dhcp200-52:src lucas$ ./cliente localhost 9000 user1
NICK user1
POST oi, eu sou um usuario legal
[user2] legal conhecer voces
POST qual eh o nome de vcs?
POST todos estao me ouvindo?
[user2] sim
[user3] sim
POST vou sair, tchau!
CLOSE
dhcp200-52:src lucas$

src - cliente - 60x49
dhcp200-52:src lucas$ ./cliente localhost 9000 user2
NICK user2
[user1] oi, eu sou um usuario legal
POST legal conhecer voces
MUTE user1
UNMUTE user1
[user1] todos estao me ouvindo?
POST sim
[user3] sim
[user1] vou sair, tchau!
POST tchau user1

src - cliente - 62x49
dhcp200-52:src lucas$ ./cliente localhost 9000 user3
NICK user3
[user1] oi, eu sou um usuario legal
[user2] legal conhecer voces
[user1] qual eh o nome de vcs?
[user1] todos estao me ouvindo?
[user2] sim
POST sim
[user1] vou sair, tchau!
[user2] tchau user1
```

Figure 11: Após a Saída de User1, User2 faz um Post dando se despedindo, porém User1 não recebe mais o post, pois já saiu do Chat; apenas User 3 recebe o post. **Fonte:** Aplicação TP1 em funcionamento.

esses clientes. Todas as estruturas utilizadas no trabalho, como Vetores, Matrizes e etc, são simples se comparado ao problema; porém foram o suficientes para sua solução.

É claro no objetivo do TP a aplicação de protocolo IRC-2, onde além de permitir a conversa entre canais, devemos usar comandos como NICK (para setar um nome), Mute (para mutar um dado usuário), entre outros. Esse foi o maior desafio desse trabalho, definir esses comandos, garantir a transmissão de dados entre clientes e servidor, garantido que não ocorra erro no caminho. E além, entregar um Post feito por dado cliente (com o comando POST) a outros clientes (usando NEW [usuario] POST), mas não para todos, deve ser obedecido certas regras definidas pelos comando MUTE e UNMUTE. Pois clientes só podem ouvir, quem eles desejam ouvir, essa é a grande chave do protocolo IRC-2, a permissão e comodidade de dar ao cliente, através de comando básico no teclado, a opção de mutar e não mais escutar um usuário "chato".

No entanto, apesar das dificuldades o trabalho foi de grande aprendizado, alguns muito importantes, como por exemplo: a limpeza do buffer do teclado, para evitar erros de leitura do teclado, aprendizado de funções e estruturas de dados de bibliotecas específicas; também relembramos o uso de funções básicas para complementação e elegância do trabalho prático.

Esse trabalho 1 demonstrou como é possível com conceitos fundamentais de redes de computadores, fazer uma aplicação de grande uso e tão necessitada nos dias atuais. Como pode perceber, podemos aplicar esse chat na rede e utilizá-lo na sala do trabalho, em casa, entre outros. Esse trabalha mais uma vez evidencia mais uma vez a importância de Redes de Computadores, suas aplicações práticas e porque ela é tão importante para o profissional de Sistemas de Informação.

7 Referências

N.Ziviani, *Projeto de Algoritmos com implementação em C e Pascal*, Cengage Learning, 3ª Ed Revista e Ampliada, 2011.

T.H.Cormen, E.Leiserson, R.L.Rivest, C.Stein, *Algoritmos: Teoria e Prática*, Elsevier, Tradução da 3ª Edição Americana, 2012.

Sockets Tutorial: <http://www.linuxhowtos.org/C-C++/socket.html> - acessado pela última vez para esse trabalho em 05/10/2015

Materiais disponibilizados no moodle: A especificação do TP, exemplos de sockets e bibliotecas de socket.