

# Trabalho de Redes de Computadores

—

## Aplicação Chat

Leonardo Claudio de Paula e Silva<sup>1</sup>, Guilherme Caixeta de Oliveira<sup>1</sup>

<sup>1</sup>Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo (USP)  
São Carlos – SP – Brazil

**Abstract.** *This monograph details the creation process and implementation of a chat software. It's aim is to develop the computer network concepts presented in lectures, elaborated sockets in C language to evidence the concepts during implementation. Another objective of this monograph is to present difficulties found in the process and to show the project's final result. The project consists of a few basic features of this type of program implemented in a distributed communication model P2P.*

**Resumo.** *Esta monografia detalha o processo de criação e implementação de um programa para chat. O intuito é desenvolver os conceitos de redes de computadores apresentados em aula com a elaboração de sockets na linguagem C para que os conceitos sejam evidenciados na implementação. Outro objetivo desta monografia é apresentar as dificuldades encontrados durante o processo e apresentar o resultado final do projeto. O projeto consiste em algumas funcionalidades básicas desse tipo de programa implementado sobre o modelo de comunicação descentralizado P2P.*

## 1. Introdução

O projeto visa complementar o conteúdo da disciplina SSC-0641 Redes de Computadores ministrada pelo prof. Julio Cezar Estrella e pelo PAE Carlos Henrique Gomes Ferreira. O intuito do projeto é apresentar os conceitos do modelo OSI abordando mais especificamente as camadas de aplicação, transporte e rede. Para isso, é feita a implementação de um programa com *sockets* em C, para que os conceitos fiquem evidentes.

### 1.1. Modelagem do Sistema

O projeto possuía as seguintes especificações para implementação, que foram seguidas de forma que o projeto ficasse coerente com o pedido pelo professor.

1. Ser implementado exclusivamente na Linguagem C.
2. Utilizar *sockets*, sendo proibido o uso de bibliotecas que abstraíam a implementação de *sockets*, funções prontas etc.
3. Abordar os conceitos de uma arquitetura P2P pura.
4. Um menu principal contendo **adicionar contato**, **listar contatos**, **excluir contato**, **enviar uma mensagem**, **mensagem em grupo**, **sair**.

## 1.2. Funcionalidades

Dadas as especificações do projeto, é necessário entendê-las para implementá-las de maneira coerente e eficaz. Portanto, detalha-se agora a abordagem de cada funcionalidade apresentada na subseção anterior.

1. Adicionar contato: Abrir uma conexão e adicionar à lista de contatos.
2. Listar contatos: Exibição dos contatos conectados na aplicação no momento.
3. Excluir contato: Fechar conexão e retirar da lista de contatos.
4. Enviar mensagem: Envia uma mensagem para um contato conectado.
5. Mensagem em grupo: Envia uma mensagem em grupo para vários contatos conectados.
6. Sair: Fecha as conexões e o programa.

## 2. Fundamentação Teórica

Para executar o projeto com êxito além de ser eficiente, é necessário estudar os conceitos de Redes de Computadores referentes a ele. Nesta seção, discute-se um pouco de alguns tópicos necessários para a elaboração do sistema proposto.

### 2.1. Modelos de Referência

Para que a rede de computadores a nível global funcione, algumas modelagens padronizadas foram propostas. Dentre elas se destacam o modelo OSI (*Open System Interconnection*) e o TCP/IP (*Transmission Control Protocol/Internet Protocol*) além de um modelo híbrido tentando explorar o melhor dos dois modelos.

#### 2.1.1. Modelo OSI

O modelo OSI é um modelo conceitual que padroniza a forma de comunicação em telecomunicação e sistema de computadores sem mencionar ou se preocupar com sua estrutura interna ou tecnologia. Seu objetivo é atingir a interoperabilidade fazendo uso de protocolos padronizados. Este modelo subdivide a comunicação em 7 camadas abstratas (*layers*). Camada camada é servida pela camada inferior, e serve a superior. O modelo é composto pelas seguintes camadas como ilustra a Figura 1.

1. Camada Física: Define as especificações físicas e elétricas dos meios de comunicação (fios de cobre, fibra óptica etc), define os protocolos de comunicação entre dois dispositivos ligados diretamente e os modos de comunicação (simplex, half duplex, full duplex) e a topologia.
2. Camada de Enlace: Uma de suas maiores funções é detectar e potencialmente corrigir erros encontrados na camada física. Faz uso do MAC (Media Access Control) responsável por controlar como um dispositivo ganha acesso ao dados e LLC (Logical Link Control) responsável por verificar erros e sincronização de pacotes.
3. Camada de Rede: Responsável pela parte funcional da transferência de dados (datagramas) entre um nó e outro. Traduz o endereço físico utilizado nas camadas abaixo para endereço de máquina. A rede é o meio pelo qual os diversos nós com seus respectivos endereços possam conversar e trocar mensagens entre si.

4. Camada de Transporte: Esta camada é responsável pela transmissão de segmentos entre uma rede e outra. Ela controla a confiança de um enlace e controle de erro. Seus protocolos podem ser orientados a conexão ou não. Os orientados podem rastrear os segmentos e retransmití-los caso falhem em chegar ao destino.
5. Camada de Sessão: Cuida das conexões entre os computadores. Ele estabelece, gerencia e encerra tais conexões. Provê checkpoints, adiantamento da terminação e reinicia os processo.
6. Camada de Apresentação: Contextualiza as diversas camadas de aplicação, sendo que podem usar diferentes sintaxes e semânticas para se comunicar. Encapsula a informação adquirida e repassa para as camadas abaixo.
7. Camada de Aplicação: É a mais próxima ao usuário final, assim, é nesta camada que o usuário se comunica com a aplicação. Como essencialmente a aplicação está fora da modelagem OSI, é função desta camada identificar padrões de comunicação da aplicação e verificar recursos e sincronização para que a aplicação tenha um desempenho satisfatório.

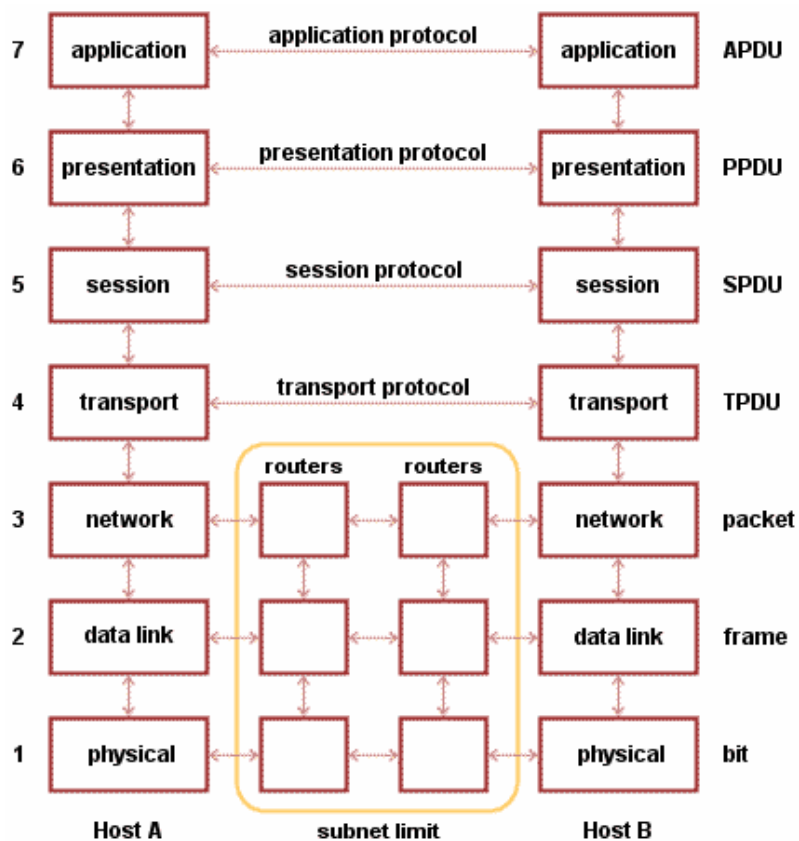


Figura 1. Modelo OSI de comunicação

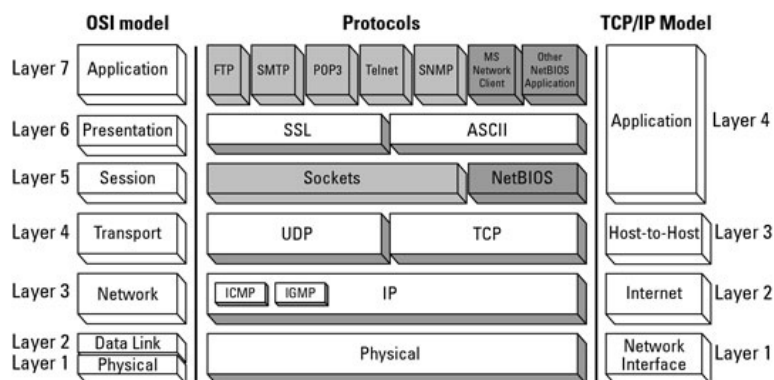
### 2.1.2. Modelo TCP/IP

Diferentemente do modelo OSI, o modelo TCP/IP não é um modelo conceitual e sim real. O modelo provê subdivisões da comunicação mais palpáveis para o mundo real sendo amplamente usado nos dias atuais. Usa-se apenas 4 camadas para dividir a comunicação.

Seu nome provem de um de seus mais importante protocolos, o TCP/IP, também chamado de **modelo internet**.

Como pode-se ver na Figura 2, é possível comparar o modelo OSI apresentado com o modelo TCP/IP. A distinção entre eles é clara, além de suas semelhanças. Um ponto importante desses modelos é que seu funcionamento ocorre mediante os protocolos vistos no centro da imagem. Eles são os responsáveis pela padronização da informação, possibilitando a comunicação.

1. Camada Física: É usada para mover pacotes entre duas camadas de *internet* de dois *hosts* no mesmo enlace. O processo de enviar e receber pacotes no enlace pode ser feito tanto por *software* como por *firmware* em *chip* dedicado (*hardware*). Para que o processo seja bem sucedido, é necessário empacotar o dado colocando um cabeçalho para então transmitir o quadro (*frame*) pelo meio.
2. Camada de Internet: É responsável por enviar os pacotes por múltiplas redes. Esta camada tem as funções principais: fornecer e identificar endereços, e efetuar o roteamento de pacotes.
3. Camada de Transporte: Também conhecida como *link* ou *host-to-host*. Estabelece o canal usado pela aplicação para troca de informações. Estabelece conexão processo-a-processo, assim, é independente da estrutura sobre a qual os dados do usuário foram construídos. Também é sua função zelar pela comunicação confiável, sendo responsável por identificar erros, corrigi-los e fazer o controle de fluxo e de congestionamento.
4. Camada de Aplicação: Esta camada é a camada que implementa o HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol) e DHCP (Dynamic Host Configuration Protocol), por exemplo. Sua função é empacotar os dados para que as camadas inferiores os tratem. Ela é fundamental, pois é ela quem faz a ligação entre o usuário e a rede, bem como a contextualização de semânticas e sintaxes.



**Figura 2. Modelo TCP/IP em comparação com o Modelo OSI evidenciando o uso de protocolos para comunicação efetiva.**

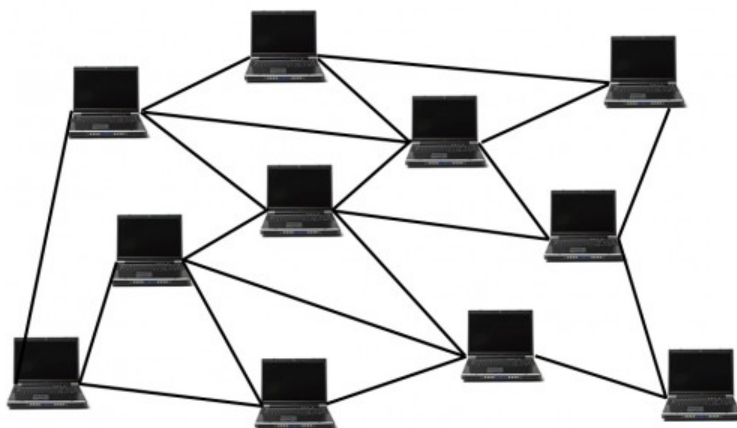
## 2.2. P2P

**Peer-to-Peer** refere-se a uma arquitetura distribuída de aplicação que subdivide tarefas para usuários da rede. *Peers* são os usuários da rede e têm os mesmos privilégios e potencialmente partições iguais das tarefas. São os nós da rede. Sua arquitetura é estruturada

sobre a ideia de que cada nós da rede é simultaneamente cliente e servidor para os outros nós da rede. Diferentemente da rede distribuída, não há um servidor central na arquitetura P2P, o que trás vantagens quando um nó da rede cai, mas desvantagens por ter que tratar todas as informações de forma completamente distribuída.

Utilizando a arquitetura P2P é possível tratar cada host como um indivíduo da rede que troca informações. A rede possui as seguintes características:

1. Iterativa: Os nós trocam informações entre si sem necessidade de supervisão. A troca de dados encerra ou pausa quando não há mais informações para serem trocadas.
2. Assíncrono: Os nós não precisam trocar informações simultaneamente. Não é necessário esperar uma resposta para propagar as informações.
3. Distribuído: Cada nó se comunica com seus vizinhos (nós diretamente conectados).



**Figura 3. Modelo P2P**

### 2.3. TCP vs UDP

*Transmission Control Protocol* (TCP) e *User Datagram Protocol* são alguns dos protocolos mais utilizados para transmissão de dados na camada de transporte.

O UDP não é confiável pois não garante a entrega do datagrama. Para que seja confiável é necessário uma implementação de segurança na aplicação tais como controle de fluxo, acknowledgments, timeouts e retransmissão. O TCP, por sua vez, é confiável pois todas essas técnicas de segurança, detecção e correção de erros já estão implementadas (recuperação de perdas ou de dados corrompidos, eliminação de pacotes duplicados, recuperação de conexão em caso de problemas na rede, controle de fluxo e de congestionamento, etc.).

### 2.4. Socket

Socket é um ponto final da comunicação de rede. É uma ferramenta normalmente oferecida pelo sistema operacional para permitir que o programador possa acessar a pilha da rede. Um socket pode ser construído a partir de um endereço IP e de uma porta. A porta é uma forma de diferenciar diversas aplicações em um mesmo IP. Por ela saem as

informações da comunicação (envio e recebimento). O IP é o endereço do host; é a partir dele que o host pode ser mapeado na rede. Na sua criação, é necessário escolher qual protocolo de rede utilizar (TCP ou UDP).

### 3. Detalhamento da Solução

Como visto na seção da **Fundamentação Teórica**, é necessário para a implementação do chat diferenciar conceitos ministrados durante o curso de Redes de Computadores e seleccionar qual utilizar para resolver o problema em específico. Assim, depois de analisar tanto o problema quanto a teoria por trás deles, é possível conceber a implementação do projeto.

#### 3.1. Dificuldades Encontradas

A priori, o programa seria implementado sobre o molde P2P puro de comunicação distribuída, de forma que cada computador fosse visualizado como um nó em um grafo. Cada nó possuiria um lado cliente e um lado servidor para receber e abrir conexões. A ideia era que permitir que nós não conectados diretamente pudessem se conectar, conforme explicado no tópico do Protótipo a seguir.

#### 3.2. Protótipo

A estrutura inicial para o protótipo era baseada em conexões diretas e indiretas, como observado pela Figura 4. Pensou-se em permitir que um nó não diretamente conectado (indiretamente conectado) pudesse se comunicar com outros sem abrir uma conexão direta. Entretanto, por dificuldades em lidar com sincronismo das *threads* e das primitivas de envio e recebimento dos *sockets*, decidiu-se continuar o projeto do chat apenas com conexões diretas.

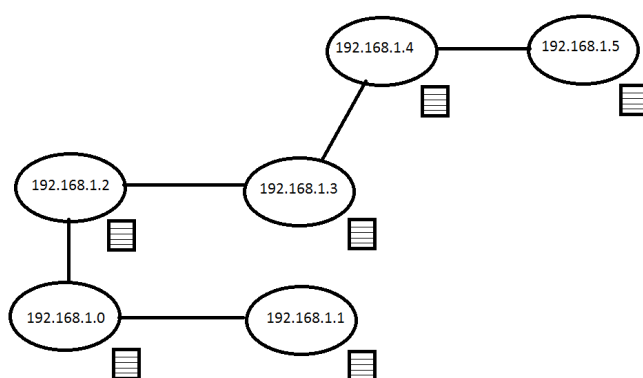


Figura 4. Estrutura de uma possível rede P2P

#### 3.3. Estrutura

O código foi subdividido em vários módulos de forma a facilitar a compreensão e a implementação. As funções marcadas como internas são utilizadas apenas dentro do próprio módulo para encapsular algumas operações.

### 3.3.1. Global

O módulo **Global** é um *header* que contém todas as bibliotecas e funções úteis para todos os outros módulos. Ele inclui as bibliotecas: *stdio*, *stdio\_ext*, *stdlib*, *unistd*, *pthread*, *string*, *errno*, *sys/socket*, e *arpa/inet*. Além disso, há a função *socket2ip* que recebe um *socket* e retorna o IP correspondente a ele.

### 3.3.2. Timer

O módulo **Timer** apenas disponibiliza um contador de tempo. É utilizado para controlar a frequência de execução das *threads* que não são bloqueantes, de forma evitar o excesso de processamento. As funções disponíveis são:

1. *timer\_start*: Inicializa a contagem do timer.
2. *timer\_stop*: Para a contagem do timer.
3. *timer\_timemsec*: Retorna o tempo entre o *start* e o *stop* em milissegundos (ms).
4. (interna) *timer\_timensec*: Retorna o tempo entre o *start* e o *stop* em nanossegundos (ns).

### 3.3.3. Client

O módulo **Client** é responsável por disponibilizar as primitivas de manipulação de conexões através dos *sockets* para um outro contato. As duas funções disponíveis são:

1. *client\_connect*: Cria o *socket* e tenta abrir uma conexão com o IP e a porta dados.
2. *client\_disconnect*: Termina a conexão e fecha o *socket* dado.

### 3.3.4. Server

O módulo **Server** é responsável por disponibilizar a porta de entrada para conexões providas de outros contatos. Ele foi implementado em forma de *struct*, onde ficam informações dele. O server executa como uma *thread* independente que recebe as conexões e armazena-as numa lista, que então é posteriormente buscada pelo próprio *mesenger*. As funções disponíveis são:

1. *server\_init*: Inicializa a *struct* do server.
2. *server\_destroy*: Destrói a *struct* do server (desaloca memória, etc.).
3. *server\_start*: Inicializa o server (*socket*, *bind*, *listen*) e a *thread* (*accept*).
4. *server\_stop*: Fecha o *socket* do server e finaliza a *thread*.
5. *server\_hasNewConnections*: Checa se existem novas conexões no server que estão pendentes.
6. *server\_getNewConnections*: Recebe em forma de lista de *sockets* das conexões que estão pendentes.
7. (interna) *server\_addNewConnection*: Manipula a lista de conexões pendentes e adiciona um novo *socket*.
8. (interna) *server\_run*: Função da *thread* propriamente dita do server.
9. (interna) *server\_shutdown*: Função que efetivamente fecha o *socket*.

### 3.3.5. Connection

O módulo **Connection** é a estrutura de dados que guarda as informações de uma conexão ativa (um contato). Ele foi implementado em forma de *struct*. Cada *connection* executa como uma *thread* independente que tem a função principal de monitorar o recebimento (recv) de mensagens naquela conexão, além de tratar essas mensagens quando elas chegam. É aqui também que todas as mensagens são guardadas até que o usuário faça a requisição da leitura dessas mensagens. As funções disponíveis são:

1. *connection\_new*: Constrói uma *struct connection* a partir dos parâmetros, e retorna.
2. *connection\_setUsername*: Altera o username de uma *connection*.
3. (interna) *connection\_pushMessage*: Guarda uma nova mensagem recebida na lista de mensagens da *connection*.
4. *connection\_popMessage*: Retorna a primeira mensagem da lista de mensagens recebidas (FIFO).
5. *connection\_hasMessages*: Retorna se há mensagens na lista de mensagens recebidas.

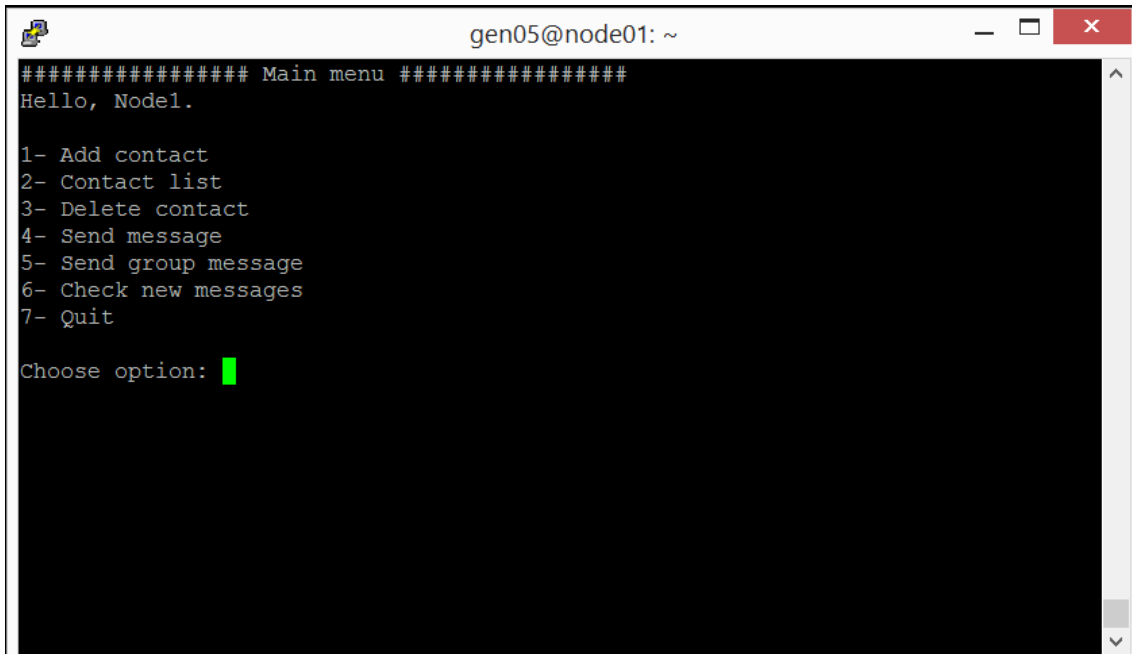
### 3.3.6. Messenger

O módulo **messenger** é onde fica a implementação do *messenger* em si, suportado pelos outros módulos. Também implementado como uma *struct*, ele guarda as informações do usuário atual e uma lista das conexões ativas (*structs connection*). Quando o *messenger* é iniciado pela *main*, ele inicia um *server* e cria uma *thread* para si mesmo. Essa *thread* é utilizada para checar as novas conexões que vem do *server*, para então adicioná-las na lista de conexões do *messenger*. Em seguida, o menu é acionado e inicia-se as iterações com o usuário (o menu executa na *thread* da própria *main*). As funções disponíveis são:

1. *messenger\_init*: Inicializa a *struct* do *messenger*.
2. *messenger\_destroy*: Destrói a *struct* do *messenger* (desaloca memória, etc.).
3. *messenger\_start*: Inicializa o *messenger*. Chamada bloqueante que roda as iterações com o usuário até que ele decida fechar o programa. É chamada pela *main* para abrir o *messenger*.
4. (interna) *messenger\_stop*: Chamada quando o usuário quer fechar o programa; finaliza as *threads* das conexões e do *messenger* e finaliza o *server*.
5. (interna) *messenger\_run*: Função da *thread* do *messenger*. Responsável por carregar as conexões de entrada do *server* para o *messenger*, além de criar as *threads* das *connections*.
6. (interna) *messenger\_conn\_run*: Função da *thread* de uma *connection*. Trata o recv de cada conexão, além de finalizar automaticamente a conexão caso haja desconexão.
7. (interna) *messenger\_stopConn*: Finaliza a *thread* e encerra a conexão de uma *connection* específica.
8. (interna) *messenger\_menu*: Executa o menu principal. A Figura 5 mostra o menu principal.
9. (interna) *messenger\_menu\_\**: Funções específicas das funções do menu.
10. (interna) *messenger\_conn\_connected2*: Checa se está conectado a um contato.



11. (interna) *messenger\_conn\_getConnByIP*: Retorna uma *connection* dado o IP, da lista de conexões ativas.
12. (interna) *messenger\_conn\_getConnByPos*: Retorna uma *connection* dada a posição, da lista de conexões ativas.
13. (interna) *messenger\_conn\_getConnPosByIP*: Retorna a posição de uma *connection* na lista de conexões ativas, dado o IP.
14. (interna) *messenger\_conn\_add*: Adiciona uma *connection* na lista de conexões ativas.
15. (interna) *messenger\_conn\_remove*: Remove uma *connection* na lista de conexões ativas.
16. (interna) *messenger\_msg\_encode*: Encapsula o formato das mensagens trocadas entre os *messengers*.

A screenshot of a terminal window titled 'gen05@node01: ~'. The terminal displays a main menu with the following text: '##### Main menu #####', 'Hello, Node1.', a list of seven options (1- Add contact, 2- Contact list, 3- Delete contact, 4- Send message, 5- Send group message, 6- Check new messages, 7- Quit), and a prompt 'Choose option: ' followed by a green cursor. The terminal has a black background and white text. The window has standard Linux window controls (minimize, maximize, close) in the top right corner.

```
gen05@node01: ~
##### Main menu #####
Hello, Node1.

1- Add contact
2- Contact list
3- Delete contact
4- Send message
5- Send group message
6- Check new messages
7- Quit

Choose option: █
```

**Figura 5. Menu Principal**

### 3.4. Threads

Listando as *threads* do programa, teremos:

1. Main: *Thread* da *main*; essencialmente exibe o menu e faz as interações com o usuário.
2. messenger: *Thread* de ligação entre o server e o *messenger*; recupera conexões de entrada e adiciona os contatos no *messenger*.
3. Server: *Thread* de entrada para novas conexões; manipula o *socket* do server e guarda as conexões em lista.
4. Conexões: *Threads* das conexões, individuais para cada uma delas; manipula o *socket* das conexões para receber e tratar mensagens.

### 3.5. Mensagens

A troca de mensagens foi padronizada em: 1 *byte* para o tipo de mensagem, utilizado individualizar os campos de cada tipo de mensagem; restante para os dados específicos do tipo de mensagem. Para este projeto, definiram-se as seguintes codificações de mensagens, pelo tipo de mensagem:

1. Nome de usuário (código 0): carrega o nome do usuário remetente, repassando-o para o destinatário. Sempre é enviada após o estabelecimento de uma conexão.
2. Resposta ao nome de usuário (código 1): ao receber o nome de usuário de uma conexão, responde também com seu nome de usuário, possibilitando a troca dos nomes.
3. Mensagem (código 2): carrega uma mensagem do chat. Utilizada tanto para o envio de mensagens normais quanto em mensagens em grupo.

### 3.6. Desafios

Durante a execução do projeto, lidamos com desafios como: manipulação das várias *threads* e do sincronismo entre elas; *deadlocks*; manipulação correta dos *sockets* para escolher entre primitivas bloqueantes ou não bloqueantes; erros antes desconhecidos da rede. Esses motivos tiveram grande peso na decisão de implementar um projeto simples e funcional, ao invés da ideia inicial de conexões indiretas. Talvez com a utilização de linguagens de mais alto nível (Java ou C++) seja possível efetivar o conceito com menos problemas ou erros.

## 4. Ambientes e Ferramentas Utilizadas

### 4.1. Ambiente

O ambiente utilizado para teste do programa foi o cluster do LaSDPC - ICMC, onde várias máquinas virtuais foram disponibilizadas o aprendizado dos conceitos de Redes e de programação para este fim fosse possível. O acesso foi feito via SSH.

### 4.2. Ferramentas

Para o desenvolvimento do programa proposto, utilizou-se da interface de desenvolvimento QtCreator na linguagem C. Usou-se também bibliotecas como *pthread* e *sockets*, além das bibliotecas padrões da linguagem C como *stdio.h*, *stdlib.h*, *string.h*, etc. Não foi utilizada nenhuma biblioteca com funcionalidades prontas, conforme especificação.

## 5. Conclusão

O objetivo do projeto é complementar e praticar os conceitos apresentados em aula na implementação de uma aplicação de *chat* que utiliza esses conceitos, além de trabalhar com um ambiente de testes (o cluster) distribuído. Ao final do projeto, os alunos puderam com êxito praticar os conceitos de Redes e aprender a programar com base em *sockets*, o que mostra que o objetivo do projeto foi atingido.

## 6. Referências

Rüdiger Schollmeier, A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications, Proceedings of the First International Conference on Peer-to-Peer Computing, IEEE (2002).

[https://en.wikipedia.org/wiki/OSI\\_model](https://en.wikipedia.org/wiki/OSI_model), acessado em Junho de 2015.

RFC 1122, Requirements for Internet Hosts – Communication Layers, R. Braden (ed.), October 1989.

RFC 1123, Requirements for Internet Hosts – Application and Support, R. Braden (ed.), October 1989.

Comer, Douglas E. (2006). Internetworking with TCP/IP: Principles, Protocols, and Architecture 1 (5th ed.). Prentice Hall. ISBN 0-13-187671-6.

[https://en.wikipedia.org/wiki/User\\_Datagram\\_Protocol](https://en.wikipedia.org/wiki/User_Datagram_Protocol), acessado em Junho de 2015.

Cisco Networking Academy Program, CCNA 1 and 2 Companion Guide Revised Third Edition, P.480, ISBN 1-58713-150-1.

<https://books.google.com.br/books?id=ptSC4LpwGA0C&printsec=frontcover> - UNIX Network Programming: The sockets networking API, acessado em Junho de 2015.