

# Trabalho 1 - Parte 1

## Aplicação de 'bate-papo distribuído'

*Abraham Banafo Ampah - 117074396*

*Cristian Diamantaras Vilela - 118109047*

*Gilberto Lopes Inácio Filho - 115173699*

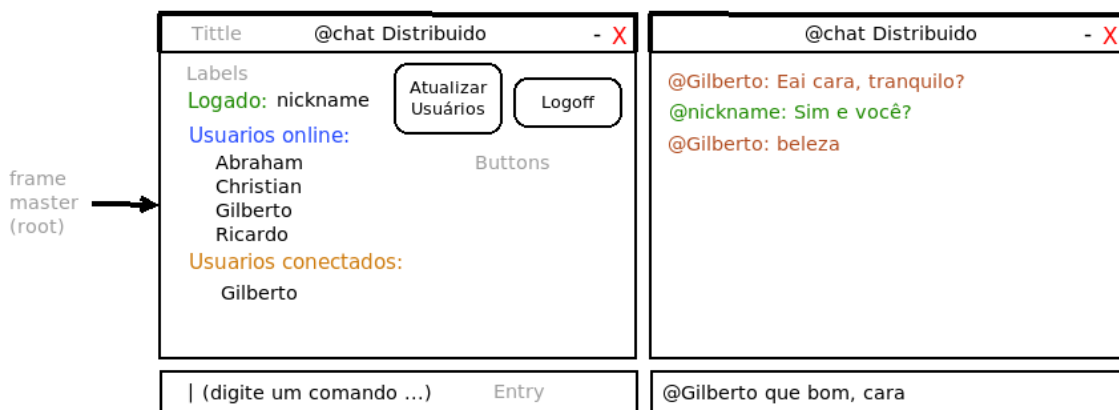
*Ricardo Kaê Bloise - 116039521*

### 1. Atividade 1 - Interface do usuário

De início, só haverão duas entidades (classes) no nosso projeto, sendo elas **Servidor Central** e **Usuario**. Desse jeito, iremos colocar as funcionalidades da interface na classe **Usuario**.

Contudo, o desejo do grupo é que a interface com usuário final seja projetada pela biblioteca gráfica **Tkinter** do Python, onde abaixo, mostra-se um esboço de como ela seria desenvolvida (e organizada). Desse jeito, mais um módulo (classe) relativo a GUI seria adicionado ao projeto e as funcionalidades da interface passariam ser responsabilidade da GUI (ou de forma mesclada com a classe **Usuario**).

Esboço da GUI do chat Distribuído  
Widgets (elementos de janela do projeto)



Nesse projeto, a GUI teria os seguintes componentes de janela (widgets):

- Um **frame principal** para acomodar todos os outros elementos
- **Labels** para imprimir todas as mensagens que chegam a aplicação.  
(!) O uso de cores seria feito para distinguir as mensagens de cada usuário.
- **Botões** para fazer logoff no Servidor Central e Atualizar a lista de usuários online
- Um campo **Entry**, para permitir o usuário final dar comandos à aplicação e serem impressos no frame principal (ou num subframe contido nele)  
(!) Em cinza claro, na imagem acima, mostram-se tais elementos.

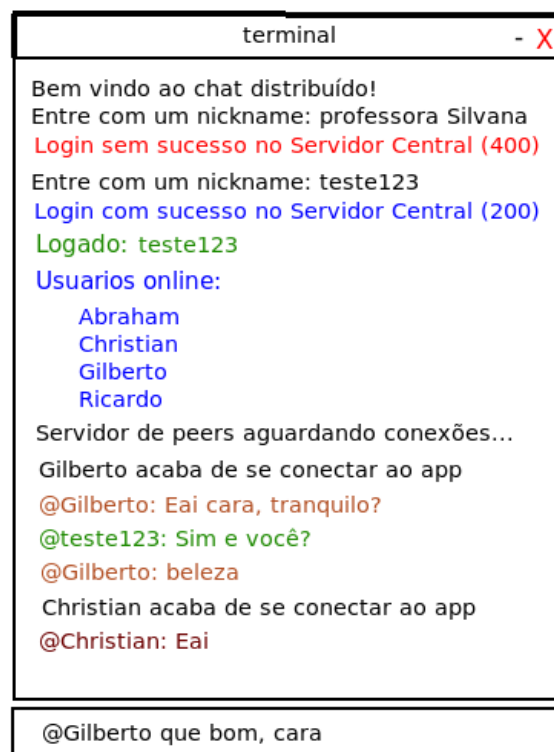
A vantagem desse projeto é permitir usuários conversarem em janelas separadas, como numa aplicação de chat convencional, o que pode ser facilmente implementado com **Tkinter**, bastando criar uma nova instância da classe **Tk** (interna à biblioteca), que cria um novo processo gráfico no SO, toda vez que um usuário se conectar. Tal processo teria basicamente o mesmo shape da janela principal e seria usado para imprimir as mensagens recebidas e enviadas desse/para usuário (*peer*), somente nessa janela.

A alternativa, caso a interface com o **Tkinter** não seja possível, é utilizar o próprio terminal, criando uma aplicação de janela única, onde todas as mensagens da aplicação aparecem na mesma tela para o usuário final.

E como forma de evitar bagunça e possibilitar que as informações recebidas sejam mais bem visíveis, o grupo usará um parâmetro *cor*, a cada categoria de mensagem que chega a aplicação, são elas:

- mensagens vinda do Servidor Central
- mensagens vinda dos *peers*
- mensagens gerais de respostas aos comandos de entrada à aplicação.

A seguir, apresenta-se uma figura de como seria a aplicação de janela única, no terminal.



```
terminal - X
Bem vindo ao chat distribuído!
Entre com um nickname: professora Silvana
Login sem sucesso no Servidor Central (400)
Entre com um nickname: teste123
Login com sucesso no Servidor Central (200)
Logado: teste123
Usuarios online:
  Abraham
  Christian
  Gilberto
  Ricardo
Servidor de peers aguardando conexões...
Gilberto acaba de se conectar ao app
@Gilberto: Eai cara, tranquilo?
@teste123: Sim e você?
@Gilberto: beleza
Christian acaba de se conectar ao app
@Christian: Eai
@Gilberto que bom, cara
```

Tanto na GUI, como na interface em linha de comando (CLI), o usuário final interage através de comandos, que são divididos em 3 categorias e descritos abaixo:

(Suporte à aplicação, Servidor Central, Comunicação entre os *peers*)

### Comandos de suporte à aplicação

- @menu: Exibe este mesmo menu de comandos.
- @exit: Encerra a aplicação de maneira elegante. Se houver conexões ativas, todas são encerradas.

### Comandos de comunicação com Servidor Central

- @login: Envia uma requisição de login para o **Servidor Central** com um *nickname* a ser definido pela aplicação.
- @logoff: Envia uma requisição de logoff para o **Servidor Central**, permitindo o usuário realizar o login novamente sob outro *nickname*.
- @get\_lista: Requisita e recebe a lista de usuários online (resposta do Servidor Central).

### Comandos de comunicação entre os *peers*

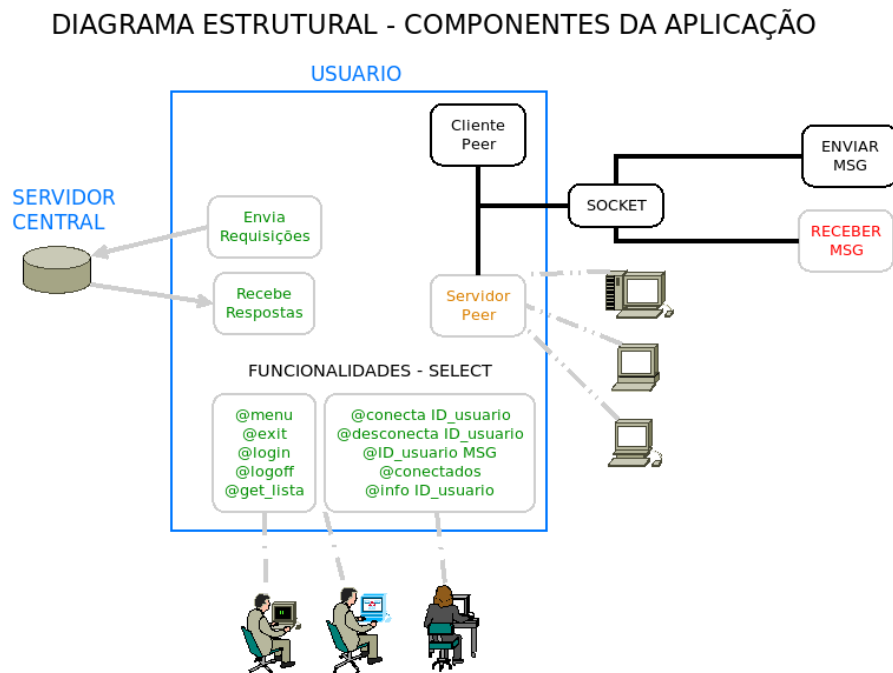
- @conecta id\_usuario: Envia uma requisição para conectar com um *peer* (de forma ativa)
- @desconecta id\_usuario: Encerra o socket de conexão com *peer*
- @id\_usuario **MENSAGEM**: Envia uma mensagem para um *peer*
- @conectados: Imprime a informação dos *peers* conectados à aplicação
- @info id\_usuario: Imprime informações de um *peer* específico conectado à aplicação. (seu ID, IP, porta, cor, socket)

Obs.: Todos os comandos são precedidos por um símbolo específico, que no caso desse projeto, designa-se o @.

Assim, quando o usuário der algum comando reconhecido como entrada, a aplicação (classe **Usuario**) internamente terá um handler para processar tal comando e imprimir na tela, a resposta a esse comando na cor apropriada.

## 2. Atividade 5 - Projeto de Modularização e Implementação

Internamente, a aplicação será modularizada (e implementada) segundo à arquitetura orientada a objetos com 2 classes: **Usuário** e **Servidor Central**. Nesse sentido, a aplicação se organiza como mostra a figura abaixo.



Na figura, os componentes escritos em **verde** representam aqueles que executarão sob ação de uma **thread** (**principal** do programa). Assim, tanto as requisições e respostas do servidor central quanto as funcionalidades da Interface, que foram descritas mais acima, executam sob ação de uma mesma thread, sendo o módulo *select* do Python, aquele responsável por garantir a “pseudo concorrência” entre as partes e garantir que a aplicação consiga multiplexar todos esses recursos (funcionalidades).

Os componentes escritos em **laranja** e **vermelho** representam outras threads designadas a cada tarefa. Uma **thread** diferente (laranja) é responsável por colocar a aplicação em modo passivo, para **aceitar conexões** entre os *peers* e outra thread (vermelha) é responsável por **executar o receive do socket** correspondente a cada usuário (*peer*) conectado e poder então imprimir a mensagem de distintos usuários simultaneamente.

Mais abaixo, descreve-se em linhas gerais, as funcionalidades previstas para cada classe.

## 2.1. Classe Usuário

A classe **Usuário** terá 3 funcionalidades principais:

1. Ela permitirá o interfaceamento com usuário final, permitindo que esse digite comandos que serão **reconhecidos e processados** (handled) pelo programa.  
(!) A funcionalidade de tais comandos já foi descrita na atividade 1 - Interface do programa.
2. Ela permitirá a comunicação com o **Servidor Central**, fazendo com que o usuário final possa **enviar requisições e receber** (imprimir) suas respostas.  
  
(!) Sendo o protocolo de requisições e respostas com Servidor Central, aquele determinado em aula, com requisições em formato JSON com operações de login, logoff e get\_lista e respostas de sucesso ou insucesso, em cada caso.
3. E por fim, permitirá a comunicação entre os *peers* conectados (online no **Servidor Central**). Desse jeito, tanto os *peers* podem se comunicar à aplicação desse projeto, pelo componente **Servidor Peer**, que fica em modo de escuta (passivo) em uma thread diferente (laranja) da principal (verde), aceitando conexões. Quanto o usuário final pode decidir se comunicar com os *peers* ativamente (segundo comando @conecta id\_usuario), sendo um Cliente no Servidor de pares deles. (Cliente Peer)

Em ambos os casos, um socket TCP terá que ser criado para estabelecer essa conexão e no caso do recebimento de mensagens, terá de ser feita uma nova thread (vermelha), para **executar o recv (receive)** desse socket em questão (criado por usuário *peer* conectado) e assim ter a possibilidade de receber mensagens de diferentes usuários simultaneamente, conferindo concorrência à aplicação.

## 2.1. Classe Servidor Central

O servidor central terá, além da sua base para estabelecer/desfazer conexões e gerenciar os atendimentos (semelhante ao Lab3), o método de retornar a lista de usuários online (que deverá ser chamado explicitamente pelo usuário, segundo comando @get\_lista).