### **Relatório do Sistema de Chat**

Aluno:João Victor Torres Parente

Este documento detalha a arquitetura, o protocolo de comunicação e os aspectos de implementação do sistema de chat desenvolvido.

#### **1. Arquitetura**

O sistema é construído sobre uma arquitetura **Cliente-Servidor**. O servidor atua como uma entidade central que gerencia a lógica de negócio, a comunicação e a persistência de dados, enquanto múltiplos clientes se conectam a ele para interagir com outros usuários através de uma interface gráfica.

**1.1. Módulos do Servidor (server.py)**

O servidor é modularizado nas seguintes áreas funcionais:

* **Módulo de Rede e Concorrência**: Responsável por aceitar e gerenciar conexões de clientes.
  + start\_server(): Inicializa o socket do servidor, o vincula a um endereço (HOST, PORT) e entra em um loop infinito para aceitar novas conexões.
  + handle\_client(conn, addr): Função executada em uma thread separada para cada cliente, gerenciando todo o ciclo de vida da comunicação com aquele cliente.
* **Módulo de Persistência de Dados (SQLite)**: Gerencia o armazenamento e a recuperação de dados do banco de dados chat\_users.db.
  + init\_db(): Cria as tabelas users e offline\_messages se não existirem.
  + register\_user(), authenticate\_user(): Funções para registro e autenticação de usuários, interagindo com a tabela users.
  + store\_offline\_message(), get\_offline\_messages(): Funções para armazenar e recuperar mensagens para usuários que estão offline, utilizando a tabela offline\_messages.
  + get\_all\_users(): Recupera a lista de todos os usuários registrados para a funcionalidade de "lista de contatos".
* **Módulo de Gerenciamento de Estado**: Mantém o estado atual dos usuários conectados em memória.
  + online\_users: Um dicionário que mapeia nomes de usuário a seus respectivos objetos de socket, permitindo o encaminhamento de mensagens em tempo real.
  + online\_users\_lock: Um objeto threading.Lock para garantir o acesso seguro ao dicionário online\_users por múltiplas threads, prevenindo condições de corrida.

**1.2. Módulos do Cliente (client.py)**

O cliente é encapsulado em uma única classe, ChatClient, que pode ser dividida nos seguintes módulos lógicos:

* **Módulo de Interface Gráfica (GUI)**: Construído com tkinter, é responsável por toda a interação com o usuário.
  + create\_login\_gui(): Renderiza a tela inicial de login e registro.
  + create\_chat\_gui(): Renderiza a interface principal do chat (lista de contatos, janela de mensagens, campo de entrada) após o login bem-sucedido.
* **Módulo de Rede**: Gerencia a comunicação com o servidor.
  + connect\_to\_server(): Estabelece a conexão do socket com o servidor.
  + send\_data(): Serializa e envia dados em formato JSON para o servidor.
  + receive\_messages(): Roda em uma thread de fundo para escutar continuamente as mensagens do servidor sem bloquear a interface gráfica.
* **Módulo de Gerenciamento de Estado do Cliente**: Mantém o estado local do cliente.
  + self.username, self.is\_connected, self.recipient\_chat\_window: Variáveis que armazenam o nome de usuário logado, o status da conexão e o destinatário da conversa atual, respectivamente.

#### **2. Protocolo de Comunicação**

A comunicação é baseada na troca de mensagens no formato **JSON** sobre um socket TCP. Cada mensagem JSON é terminada por um caractere de nova linha (\n), que atua como um delimitador para o servidor e o cliente saberem quando uma mensagem completa foi recebida.

O campo principal em todas as mensagens é “type”, que define a natureza da solicitação ou resposta.

**2.1. Formato das Mensagens (Cliente -> Servidor)**

| Tipo (type) | Campos Adicionais | Descrição do Fluxo |
| --- | --- | --- |
| REGISTER | username, password | Enviado quando o usuário clica em "Registrar". O servidor tenta criar o usuário no banco de dados e responde com SUCCESS ou ERROR. |
| LOGIN | username, password | Enviado ao tentar fazer login. O servidor valida as credenciais no banco de dados. Se bem-sucedido, responde com SUCCESS, envia mensagens offline e notifica outros usuários. Se falhar, envia ERROR. |
| GET\_CONTACTS | username | Enviado pelo cliente após o login para solicitar a lista de todos os usuários registrados. O servidor responde com um pacote CONTACT\_LIST. |
| MESSAGE | sender, recipient, timestamp, text | Contém uma mensagem de chat para outro usuário. O servidor recebe, verifica se o destinatário está online e encaminha a mensagem. Se offline, a armazena no banco de dados. |
| TYPING | sender, recipient, is\_typing | Enviado para notificar o servidor que o usuário está (true) ou parou (false) de digitar para um destinatário específico. |

Exportar para as Planilhas

**2.2. Formato das Mensagens (Servidor ->Cliente)**

| Tipo (type) | Campos Adicionais | Descrição do Fluxo |
| --- | --- | --- |
| SUCCESS | message, (opcional) username | Resposta a uma operação bem-sucedida (login, registro). Se for um login, inclui o username para confirmação. |
| ERROR | message | Resposta a uma operação que falhou (ex: usuário já existe, senha incorreta, usuário já logado). |
| INFO | message | Enviado para fornecer informações ao cliente, como a notificação de que uma mensagem foi armazenada para um usuário offline. |
| MESSAGE | sender, recipient, timestamp, text | Pacote encaminhado pelo servidor, contendo uma mensagem de outro usuário ou uma mensagem offline sendo entregue. |
| CONTACT\_LIST | contacts | Resposta à solicitação GET\_CONTACTS, contendo uma lista de todos os nomes de usuário registrados. |
| TYPING\_STATUS | user, is\_typing | Encaminhado a partir de um pacote TYPING, informa ao cliente se o interlocutor da conversa atual está digitando. |
| STATUS | user, status (online/offline) | Notificação enviada a todos os clientes online quando um usuário faz login ou logout, para que a interface possa refletir o status. |

#### **3. Implementação dos Requisitos Funcionais**

* **Cadastro e Login de Usuários**: Implementado através das rotas REGISTER e LOGIN. O servidor utiliza o módulo de persistência SQLite para validar e armazenar as credenciais dos usuários. O cliente fornece a GUI para a entrada desses dados.
* **Mensagens Privadas**: O tipo de pacote MESSAGE permite a comunicação 1-para-1. O servidor atua como um roteador de mensagens, encaminhando-as para o destinatário correto com base no dicionário online\_users.
* **Mensagens Offline**: Se um destinatário não estiver no dicionário online\_users, o servidor invoca store\_offline\_message para salvar a mensagem. Ao fazer login, o usuário recebe todas as suas mensagens pendentes através da função get\_offline\_messages.
* **Lista de Contatos e Status Online**: Após o login, o cliente solicita a lista de contatos (GET\_CONTACTS). O servidor responde com todos os usuários registrados (CONTACT\_LIST). O status online/offline é transmitido a todos os clientes conectados através do pacote STATUS sempre que um usuário entra ou sai.
* **Indicador de "Digitando"**: O cliente envia um evento TYPING quando o usuário começa a digitar e, após um timeout de inatividade, envia outro evento para indicar que parou. O servidor repassa essa informação como TYPING\_STATUS para o destinatário.

#### **4. Gerenciamento de Conexões e Concorrência**

* **Atendimento a Múltiplos Clientes**: O servidor é multithreaded. O laço principal em start\_server aceita conexões de forma bloqueante (server\_socket.accept()). Para cada nova conexão, ele instancia e inicia uma nova threading.Thread que executa a função handle\_client. Isso permite que o servidor lide com múltiplos clientes simultaneamente, já que cada cliente tem sua própria thread de execução dedicada, isolando sua comunicação do restante.
* **Tratamento de Falhas de Conexão**: A função handle\_client está envolta em um bloco try...except...finally. Se um cliente se desconectar abruptamente (fechando a janela ou por falha de rede), uma exceção como ConnectionResetError é levantada. O bloco finally garante que, independentemente de como a conexão termine, o usuário será removido da lista online\_users e seu socket será fechado, limpando os recursos e notificando outros usuários de seu status offline.
* **Comunicação Cliente-Servidor**: O cliente também utiliza threading para não congelar a GUI. A thread principal gerencia a interface tkinter, enquanto uma segunda thread (receive\_thread) é dedicada a ouvir as respostas do servidor. Para atualizar a GUI a partir dessa thread de rede de forma segura, o cliente utiliza o método self.master.after(0, lambda: ...). Este método agenda a execução de uma função no loop de eventos principal do tkinter, evitando conflitos de concorrência na manipulação dos widgets da interface.

#### **5. Desafios e Aprendizados**

* **Persistência de Dados:** Uma dificuldade inicial foi garantir que os novos registros de usuários fossem imediatamente persistidos para autenticação. Observou-se que, após registrar um novo usuário, a tentativa de login subsequente falhava, causando instabilidade no cliente. A causa era a falta do comando conn.commit() após a instrução INSERT na função register\_user. Sem o commit, a transação não era gravada no banco de dados, e o novo usuário não era encontrado na validação. A solução foi garantir que toda operação de escrita no banco fosse seguida por um commit, solidificando o conceito de transações atômicas.
* **Concorrência na GUI**: A principal dificuldade foi integrar a comunicação de rede (que é bloqueante e/ou demorada) com a interface gráfica tkinter (que precisa ser responsiva). Tentar receber dados na thread principal congelaria a aplicação.

**-Solução**: A implementação de uma thread separada (receive\_messages) no cliente foi a solução. O aprendizado fundamental foi a necessidade de isolar operações de I/O de longa duração da thread da GUI.

* **Desafio 2: Atualização Segura da GUI**: Uma vez que a comunicação estava em outra thread, surgiu o problema de como atualizar os componentes da GUI (como exibir uma nova mensagem) de forma segura. tkinter não é thread-safe.

**-Solução**: O uso de master.after() foi o aprendizado chave. Este mecanismo permitiu "postar" as atualizações da GUI da thread de rede para a fila de eventos da thread principal, garantindo a execução segura.

* **Desafio 3: Gerenciamento de Estado Compartilhado no Servidor**: Garantir que o dicionário online\_users não fosse corrompido por acessos simultâneos de múltiplas threads de clientes.

**-Solução**: A aplicação de um threading.Lock (online\_users\_lock) para proteger todas as leituras e escritas no dicionário compartilhado foi a solução. Isso solidificou o conceito de seções críticas e a importância de mecanismos de sincronização em programação concorrente.

* **Aprendizados Gerais**:

**-Redes**: O projeto tornou prático o conceito de sockets TCP, a natureza de stream dos dados e a necessidade de um protocolo de aplicação (delimitadores de mensagem como \n) para estruturar a comunicação.

**-Persistência**: O uso de SQlite demonstrou uma forma simples e eficaz de adicionar persistência a uma aplicação, com aprendizados sobre a criação de esquemas, chaves primárias para garantir unicidade (username) e o ciclo de vida de uma conexão com o banco de dados.

#### **6. Limitações e Possíveis Bugs**

* **Segurança**: A limitação mais crítica é que **as senhas são armazenadas e transmitidas em texto plano**. Em um ambiente real, isso é uma falha de segurança grave. O ideal seria usar hashing com salt (ex: Argon2, bcrypt) para armazenar as senhas.
* **Falta de Atualização Visual do Status**: A função update\_contact\_status no cliente apenas imprime o novo status no console (print(f"Usuário {user} está agora {status}")). Não há uma implementação para atualizar visualmente a contact\_listbox (ex: mudando a cor do nome do contato ou adicionando "(Online)" ao lado).
* **Perda de Histórico de Chat**: Ao selecionar um novo contato na lista, o histórico da conversa anterior é completamente apagado da tela, pois o código executa self.chat\_display.delete('1.0', tk.END). Não há persistência do histórico de chat no lado do cliente.
* **Não há Chat em Grupo**: O protocolo foi projetado estritamente para mensagens 1-para-1, com um campo recipient único. Não há suporte para salas de chat ou mensagens em grupo.
* **Escalabilidade do Servidor**: O modelo de uma thread por cliente não escala bem para um grande número de usuários simultâneos devido ao consumo de memória e sobrecarga de troca de contexto do sistema operacional. Para um sistema em larga escala, arquiteturas baseadas em I/O assíncrono (como asyncio) seriam mais adequadas.
* **Ausência de Feedback de Entrega**: O remetente recebe uma confirmação de que a mensagem foi armazenada para um usuário offline, mas não há um mecanismo para notificar quando a mensagem for efetivamente entregue e lida pelo destinatário.