

Comunicação, Replicação e Tolerância a Falhas em Sistemas Distribuídos

ATIVIDADE 3 - DESENVOLVIMENTO DE CHAT EM GRUPO VIA MULTICAST

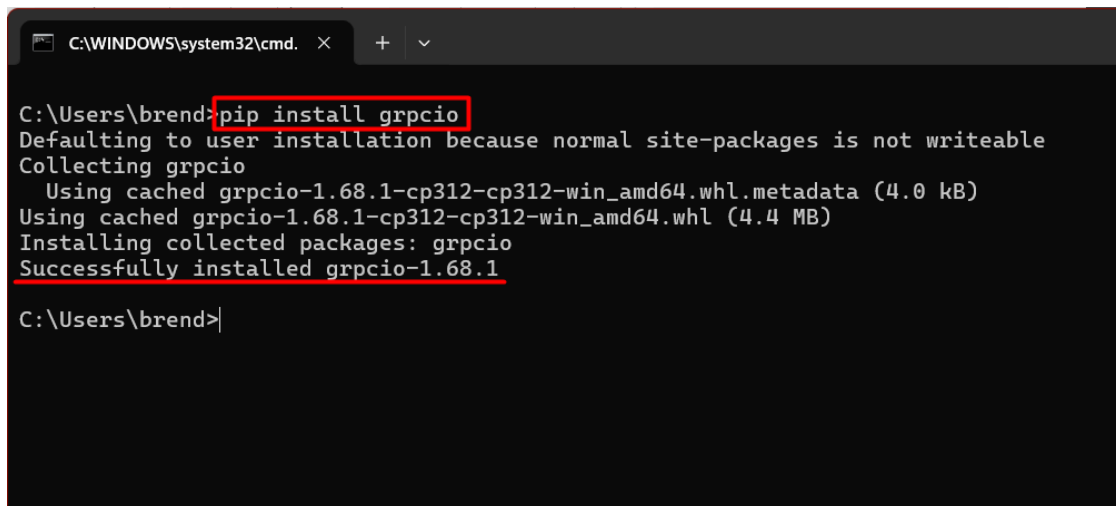
Brenda Martinez, Jaqueline Andrade, Rafael Magalhães
Análise e Desenvolvimento de Sistemas
março/2025

Sumário

Tecnologias.....	2
Sistema de Chat em Grupo Distribuído com Comunicação via Multicast.....	3
Como Executar o Sistema (Windows).....	3
Explicação do Código.....	3
chat_client.py.....	3
Requisitos do Projeto.....	4
Comunicação em Grupo com Multicast.....	4
Comunicação via multicast IP 224.1.1.1:5007.....	4
Demonstração.....	6
Replicação de Dados e Consistência Eventual.....	7
Gravação das Mensagens em arquivos de Réplica.....	7
Simulação: entrega fora de ordem (Delay Artificial).....	9
Processo de Reconciliação que sincroniza os dados entre réplicas.....	10
Demonstração.....	11
Controle de Concorrência com Exclusão Mútua Distribuída.....	12
Algoritmo de Ricart-Agrawala e a garantia que somente um nó por vez envia mensagens ao grupo.....	12
Exibição de mensagens de requisição e concessão do recurso.....	18
Demonstração.....	19
Tolerância a Falhas com Checkpoints e Rollback.....	19
Snapshots do estado do cliente (mensagens enviadas/recebidas).....	19
Restauração do estado salvo no último checkpoint, ao reiniciar após falha.....	21
Arquivos de checkpoint gerados.....	23
Demonstração.....	23

Tecnologias

- Essa documentação foi produzida utilizando o sistema operacional **WINDOWS**.
- Linguagem: Python 3
- Bibliotecas: socket, struct, threading, time, uuid, pickle, os, random, sys, argparse, collections.deque
- Instalando uma biblioteca python (Windows):
 - Necessário ter o python previamente instalado
 - Abrir o CMD e executar o comando “**pip install {nome da biblioteca}**”

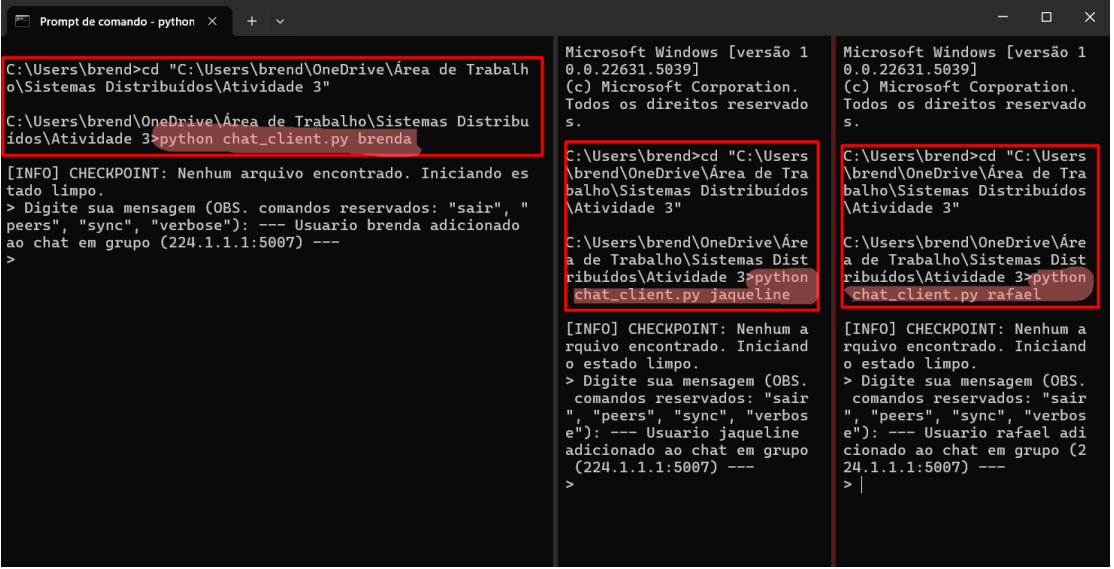


```
C:\WINDOWS\system32\cmd. x + v
C:\Users\brend>pip install grpcio
Defaulting to user installation because normal site-packages is not writeable
Collecting grpcio
  Using cached grpcio-1.68.1-cp312-cp312-win_amd64.whl.metadata (4.0 kB)
Using cached grpcio-1.68.1-cp312-cp312-win_amd64.whl (4.4 MB)
Installing collected packages: grpcio
Successfully installed grpcio-1.68.1
C:\Users\brend>|
```

Sistema de Chat em Grupo Distribuído com Comunicação via Multicast

COMO EXECUTAR O SISTEMA (WINDOWS)

1. Abra o CMD do Windows e entre na pasta onde está o arquivo "chat_client.py".
2. Execute o sistema utilizando o comando "python chat_client.py {nome_de_usuario}"
3. Repita os passos 1 e 2 para cada usuário dentro do Chat em Grupo.



The image displays three side-by-side screenshots of a Windows command prompt window. Each window shows the execution of the 'chat_client.py' script for a different user. The first window shows the user 'brenda' being added to the chat group. The second window shows the user 'jaqueline' being added. The third window shows the user 'rafael' being added. Each window also displays the directory path 'C:\Users\brend\OneDrive\Área de Trabalho\Sistemas Distribuídos\Atividade 3' and the command 'python chat_client.py {username}'.

```
C:\Users\brend>cd "C:\Users\brend\OneDrive\Área de Trabalho\Sistemas Distribuídos\Atividade 3"
C:\Users\brend\OneDrive\Área de Trabalho\Sistemas Distribuídos\Atividade 3>python chat_client.py brenda

[INFO] CHECKPOINT: Nenhum arquivo encontrado. Iniciando estado limpo.
> Digite sua mensagem (OBS. comandos reservados: "sair", "peers", "sync", "verbose"): --- Usuario brenda adicionado ao chat em grupo (224.1.1.1:5007) ---
>

Microsoft Windows [versão 10.0.22631.5039]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\brend>cd "C:\Users\brend\OneDrive\Área de Trabalho\Sistemas Distribuídos\Atividade 3"
C:\Users\brend\OneDrive\Área de Trabalho\Sistemas Distribuídos\Atividade 3>python chat_client.py jaqueline

[INFO] CHECKPOINT: Nenhum arquivo encontrado. Iniciando estado limpo.
> Digite sua mensagem (OBS. comandos reservados: "sair", "peers", "sync", "verbose"): --- Usuario jaqueline adicionado ao chat em grupo (224.1.1.1:5007) ---
>

Microsoft Windows [versão 10.0.22631.5039]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\brend>cd "C:\Users\brend\OneDrive\Área de Trabalho\Sistemas Distribuídos\Atividade 3"
C:\Users\brend\OneDrive\Área de Trabalho\Sistemas Distribuídos\Atividade 3>python chat_client.py rafael

[INFO] CHECKPOINT: Nenhum arquivo encontrado. Iniciando estado limpo.
> Digite sua mensagem (OBS. comandos reservados: "sair", "peers", "sync", "verbose"): --- Usuario rafael adicionado ao chat em grupo (224.1.1.1:5007) ---
>
```

EXPLICAÇÃO DO CÓDIGO

chat_client.py

O código do arquivo [chat_client.py](#) implementa um cliente para um chat em grupo distribuído, projetado para permitir a comunicação entre múltiplos usuários sem depender de um servidor central para o fluxo de mensagens. A base da comunicação é o protocolo UDP utilizando IP Multicast. Ele permite que múltiplos usuários, executando o script em diferentes terminais (também em máquinas diferentes na mesma rede local), comuniquem-se em um grupo compartilhado. Todos os outros clientes que estão "escutando" nesse mesmo endereço recebem as mensagens, criando uma dinâmica de grupo onde a informação é disseminada de forma eficiente para todos os participantes ativos na rede local.

Para gerenciar o acesso concorrente ao "direito de falar" e evitar que as mensagens de diferentes clientes se sobreponham de maneira caótica, o sistema implementa um algoritmo de exclusão mútua distribuída, especificamente o de **Ricart-Agrawala**. Antes de enviar uma mensagem, um cliente deve solicitar permissão aos seus pares na rede, enviando uma requisição com um número de sequência (baseado em timestamp lógico). Ele só prossegue com o envio após receber permissão (respostas) de todos os

outros participantes conhecidos, garantindo que apenas um cliente esteja efetivamente transmitindo sua mensagem de chat para o grupo por vez.

Visando a robustez e simulando cenários de sistemas distribuídos, o cliente adota uma estratégia de replicação local e aborda a consistência eventual. Cada mensagem recebida por um cliente é gravada em múltiplos arquivos de log locais (réplicas), oferecendo uma forma de redundância de dados naquele nó específico. Para simular condições de rede variáveis onde mensagens podem chegar fora de ordem, um pequeno delay aleatório artificial é introduzido antes que cada mensagem recebida seja processada e exibida. Há também um processo periódico que verifica a consistência *apenas entre* as réplicas locais de um mesmo cliente.

Finalmente, o cliente oferece tolerância a falhas básicas por meio de um mecanismo de checkpoint. Periodicamente, o estado essencial do cliente – incluindo o histórico de mensagens, os números de sequência usados na exclusão mútua e a lista de peers detectados – é salvo em um arquivo *.pkl*. Se o cliente for encerrado inesperadamente ou fechado e depois reiniciado com o *mesmo ID*, ele automaticamente tentará carregar o último checkpoint salvo, restaurando o histórico de mensagens e seu estado operacional. A interação do usuário ocorre via linha de comando, onde mensagens recebidas são exibidas e um prompt permite digitar novas mensagens ou comandos como *verbose*, *peers*, *sync* e *sair*.

Requisitos do Projeto

COMUNICAÇÃO EM GRUPO COM MULTICAST

Comunicação via multicast IP 224.1.1.1:5007

No início do código foram definidos o endereço IP e a porta, na forma de constantes, que serão utilizados para a comunicação via Multicast.

```
# --- Configuracoes ---
MULTICAST_GROUP = '224.1.1.1' # endereço IP multicast padrao
MULTICAST_PORT = 5007 # porta padrao para o grupo
REPLICATION_FACTOR = 3 # numero de arquivos de replica locais por cliente
REPLICATION_DIR = "replicas" # diretorio para guardar arquivos de replica
CHECKPOINT_DIR = "checkpoints" # diretorio para guardar arquivos de checkpoint
CHECKPOINT_INTERVAL_S = 60 # intervalo em segundos para salvar checkpoints
MESSAGE_DELAY_MS = 50 # delay maximo artificial (ms) para simular desordem
BUFFER_SIZE = 1024 # Tamanho do buffer para recebimento de mensagens
```

A função `_setup_sockets` prepara os sockets para envio e recebimento (de mensagens) no grupo multicast.

```

def _setup_sockets(self):
    # Configura os sockets UDP para envio e recebimento multicast
    try:
        # socket para envio
        self.sock_send = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
        # define o Time-To-Live (TTL) dos pacotes multicast
        self.sock_send.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 2)

        # socket para recebimento
        self.sock_recv = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
        # permite que outros sockets na mesma máquina se liguem a mesma porta necessário para vários clientes locais
        self.sock_recv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        # liga o socket a todas as interfaces de rede (') na porta definida
        self.sock_recv.bind('', MULTICAST_PORT)

        # adiciona o socket ao grupo multicast para que ele receba as mensagens enviadas para esse grupo
        mreq = struct.pack("4sl", socket.inet_aton(MULTICAST_GROUP), socket.INADDR_ANY)
        self.sock_recv.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)

    except OSError as e:
        # caso a porta já esteja em uso por outro processo
        self._log_error(f"Falha ao configurar sockets na porta {MULTICAST_PORT}: {e}. Verifique se já está em uso.")
        sys.exit(1)
    except Exception as e:
        # erros comuns
        self._log_error(f"Erro inesperado ao configurar sockets: {e}")
        sys.exit(1)

```

Foram criados sockets do tipo UDP (*SOCK_DGRAM*) para comunicação através de IPv4 (*AF_INET*). Criamos um para envio (*sock_send*) e outro para recebimento (*sock_recv*). O TTL em multicast controla quantos saltos de roteador o pacote pode dar. O valor 2 definido no código, permite que ele atravessasse roteadores (se configurados para permitir tráfego multicast).

O trecho de código

`self.sock_recv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)` permite que múltiplos processos na mesma máquina se liguem (*bind*) à mesma porta (5007). Sem isso, apenas a primeira instância do cliente conseguiria usar a porta 5007 para receber, e as outras falhariam.

Após, é preparada uma estrutura de dados (*mreq*) para a chamada de sistema que adiciona o socket a um grupo multicast. A função *socket.inet_aton(MULTICAST_GROUP)* converte o IP '224.1.1.1' para seu formato binário de 4 bytes. Já a função *socket.INADDR_ANY* indica que queremos receber pacotes destinados a esse grupo em qualquer interface de rede. Por fim, na linha `sock_recv.setsockopt(..., socket.IP_ADD_MEMBERSHIP, mreq)` o cliente é adicionado no grupo multicast 224.1.1.1.

```

def _send_multicast(self, message):
    # envia uma mensagem em formato string para o grupo multicast
    try:
        # DEBUG self._log_verbose(f"Enviando: {message}")
        self.sock_send.sendto(message.encode('utf-8'), (MULTICAST_GROUP, MULTICAST_PORT))
    except Exception as e:
        # não para a execução caso haja um erro
        self._log_error(f"Falha ao enviar mensagem multicast: {e}")

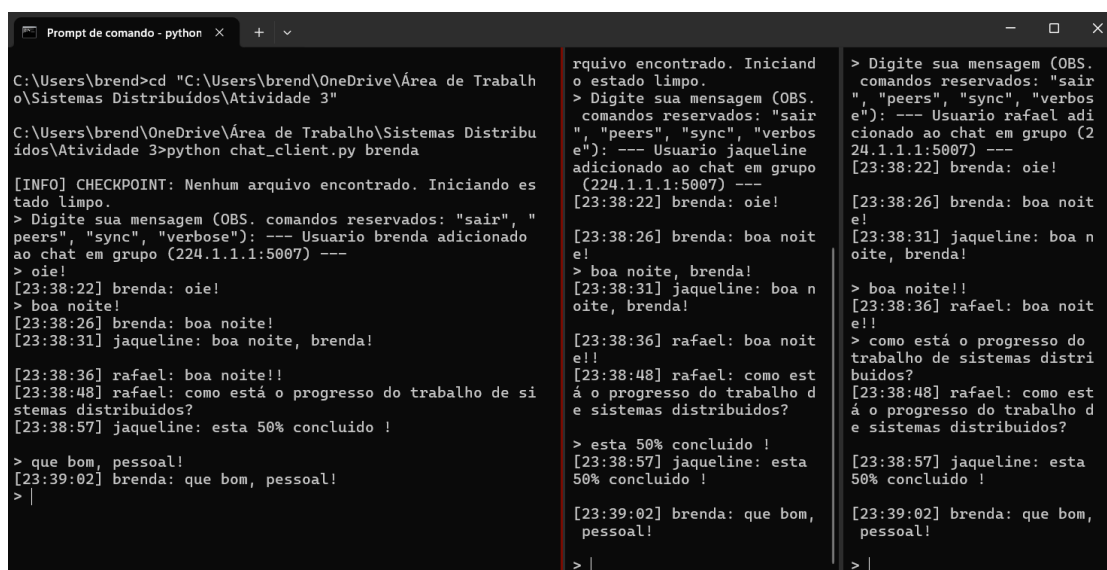
```

A função `_send_multicast(self, message)` é responsável pelo envio de mensagens via UDP. Nela, são especificados o IP e porta (definidos no início do código) destinos da mensagem.

```
def _receive_messages(self):
    # loop principal da thread que escuta e processa mensagens recebidas
    self._log_verbose("Thread de recebimento iniciada.")
    while self.running:
        try:
            # espera receber dados
            data, addr = self.sock_recv.recvfrom(BUFFER_SIZE)
            message = data.decode('utf-8')
            # DEBUG self._log_verbose(f"Recebido raw: {message} de {addr}")
            self._process_incoming_message(message)
        except socket.timeout:
            # se o socket tiver timeout apenas continua
            continue
        except OSError as e:
            # ocorre se o socket for fechado enquanto recvfrom espera, ex: ao sair
            if self.running: # so reporta erro se nao estivermos parando intencionalmente
                self._log_error(f"Problema no socket de recebimento: {e}")
                break # sai do loop
        except Exception as e:
            self._log_error(f"Falha ao processar mensagem recebida: {e}")
            # continua tentando receber outras mensagens
    self._log_verbose("Thread de recebimento terminada.")
```

A função `_receive_messages(self)` roda em uma thread separada, constantemente esperando por mensagens que chegam no grupo. Quando um pacote enviado para 224.1.1.1:5007 chega, o sistema operacional o entrega a este socket.

Demonstração



```
Prompt de comando - python
C:\Users\brend>cd "C:\Users\brend\OneDrive\Área de Trabalho\Sistemas Distribuídos\Atividade 3"
C:\Users\brend\OneDrive\Área de Trabalho\Sistemas Distribuídos\Atividade 3>python chat_client.py brenda
[INFO] CHECKPOINT: Nenhum arquivo encontrado. Iniciando estado limpo.
> Digite sua mensagem (OBS. comandos reservados: "sair", "peers", "sync", "verbose"): --- Usuario brenda adicionado ao chat em grupo (224.1.1.1:5007) ---
> oie!
[23:38:22] brenda: oie!
> boa noite!
[23:38:26] brenda: boa noite!
[23:38:31] jaqueline: boa noite, brenda!
[23:38:36] rafaël: boa noite!!
[23:38:48] rafaël: como está o progresso do trabalho de sistemas distribuídos?
[23:38:57] jaqueline: esta 50% concluído !
> que bom, pessoal!
[23:39:02] brenda: que bom, pessoal!
> |

rquivo encontrado. Iniciando o estado limpo.
> Digite sua mensagem (OBS. comandos reservados: "sair", "peers", "sync", "verbose"): --- Usuario jaqueline adicionado ao chat em grupo (224.1.1.1:5007) ---
[23:38:22] brenda: oie!
[23:38:26] brenda: boa noite!
[23:38:31] jaqueline: boa noite, brenda!
[23:38:36] rafaël: boa noite!!
[23:38:48] rafaël: como está o progresso do trabalho de sistemas distribuídos?
> esta 50% concluído !
[23:38:57] jaqueline: esta 50% concluído !
[23:39:02] brenda: que bom, pessoal!
> |

> Digite sua mensagem (OBS. comandos reservados: "sair", "peers", "sync", "verbose"): --- Usuario rafaël adicionado ao chat em grupo (224.1.1.1:5007) ---
[23:38:22] brenda: oie!
[23:38:26] brenda: boa noite!
[23:38:31] jaqueline: boa noite, brenda!
[23:38:36] rafaël: boa noite!!
[23:38:48] rafaël: como está o progresso do trabalho de sistemas distribuídos?
[23:38:57] jaqueline: esta 50% concluído !
[23:39:02] brenda: que bom, pessoal!
> |
```

A mensagem enviada no grupo, por qualquer um dos membros, é exibida para todos.

REPLICAÇÃO DE DADOS E CONSISTÊNCIA EVENTUAL

Gravação das Mensagens em arquivos de Réplica

Primeiro, criamos como constante o número de arquivos de réplica por cliente e qual será o diretório em que essas réplicas serão armazenadas.

```
# --- Configuracoes ---
MULTICAST_GROUP = '224.1.1.1' # endereço IP multicast padrao
MULTICAST_PORT = 5007 # porta padrao para o grupo
REPLICATION_FACTOR = 3 # numero de arquivos de replica locais por cliente
REPLICATION_DIR = "replicas" # diretorio para guardar arquivos de replica
CHECKPOINT_DIR = "checkpoints" # diretorio para guardar arquivos de checkpoint
CHECKPOINT_INTERVAL_S = 60 # intervalo em segundos para salvar checkpoints
MESSAGE_DELAY_MS = 50 # delay maximo artificial (ms) para simular desordem
BUFFER_SIZE = 1024 # Tamanho do buffer para recebimento de mensagens
```

Após, no método construtor `__init__`, definimos quantos arquivos de réplica teremos e onde eles ficarão, e então criamos a lista (utilizando uma list comprehension para gerar os nomes completos dos arquivos) com os nomes específicos desses arquivos para esta instância do cliente.

```
# --- Persistencia e Replicacao ---
self.checkpoint_file = os.path.join(CHECKPOINT_DIR, f"checkpoint_{self.client_id}.pkl")
self.replica_files = [
    os.path.join(REPLICATION_DIR, f"replica_{self.client_id}_{i}.log")
    for i in range(REPLICATION_FACTOR)
]
```

Quando uma mensagem de chat (**MSG**) é recebida e processada pela função `_handle_chat_message`, após ser exibida na tela e registrada no log de memória (`self.message_log`), a função de replicação é chamada:

```
def _handle_chat_message(self, sender_id, content):
    # Processa uma mensagem de chat: exibe, loga e replica
    timestamp = time.strftime('%H:%M:%S') # hora local formatada
    display_msg = f"[{timestamp}] {sender_id}: {content}"
    # log com timestamp float para ordenacao mais precisa
    log_entry = (sender_id, content, time.time())

    with self.mutex: # Protege acesso as listas de mensagens
        self.messages.append(display_msg) # adiciona a fila de exibicao
        self.message_log.append(log_entry) # adiciona ao log completo

    # --- Exibe a mensagem de chat (SEMPRE VISIVEL) ---
    # \r move para o inicio da linha, espaço extra limpa restos do prompt antigo
    print(f"\r{display_msg}{' ' * 20}")
    self._redisplay_prompt() # redesenha o prompt na linha seguinte

    # --- Replicacao Local (silenciosa por padrao) ---
    self._replicate_message(log_entry)
```


Nessa função, a `log_entry` (que contém remetente, conteúdo e o timestamp preciso de quando a mensagem foi processada localmente) é passada como argumento para `self._replicate_message`.

Por fim, a função de escrita `_replicate_message(self, log_entry)` efetivamente abre e escreve nos arquivos de log. A função recebe a `log_entry`, formata a `log_line` incluindo o timestamp numérico (ts), remetente e conteúdo, separados por | e terminando com `\n` (nova linha). Itera sobre a lista de nomes de arquivos que foi definida no `__init__` e abre cada um desses arquivos. O modo "a" (append) garante que a `log_line` será adicionada no final do arquivo, preservando o conteúdo anterior. Se o arquivo não existir, ele será criado.

```
def _replicate_message(self, log_entry):
    # Escreve a entrada do log em múltiplos arquivos de replica locais
    sender, content, ts = log_entry
    # Formato simples para o arquivo de log: timestamp|remetente|conteudo
    log_line = f"{ts}|{sender}|{content}\n"
    for replica_file in self.replica_files:
        try:
            # 'a' para append (adicionar ao fim do arquivo)
            with open(replica_file, "a", encoding='utf-8') as f:
                f.write(log_line)
        except Exception as e:
            # Reporta erro mas continua tentando outras replicas
            self._log_error(f"Falha ao escrever na replica {replica_file}: {e}")
```

Essa função é chamada toda vez que uma mensagem de chat é processada por `_handle_chat_message`.

Simulação: entrega fora de ordem (Delay Artificial)

Para realizar a simulação da entrega fora de ordem, definimos o limite máximo do delay a ser introduzido no início do arquivo. Esta linha define que o delay aleatório adicionado a cada mensagem terá um valor máximo de 50 milissegundos. Esse valor pode ser aumentado para tornar o efeito de desordem mais pronunciado e fácil de observar.

```
# --- Configuracoes ---
MULTICAST_GROUP = '224.1.1.1' # endereço IP multicast padrao
MULTICAST_PORT = 5007 # porta padrao para o grupo
REPLICATION_FACTOR = 3 # numero de arquivos de replica locais por cliente
REPLICATION_DIR = "replicas" # diretorio para guardar arquivos de replica
CHECKPOINT_DIR = "checkpoints" # diretorio para guardar arquivos de checkpoint
CHECKPOINT_INTERVAL_S = 60 # intervalo em segundos para salvar checkpoints
MESSAGE_DELAY_MS = 50 # delay maximo artificial (ms) para simular desordem
BUFFER_SIZE = 1024 # Tamanho do buffer para recebimento de mensagens
```

O local onde o delay é calculado e efetivamente aplicado é na função `_process_incoming_message(self, message)`, logo após uma mensagem do tipo **MSG** ser identificada, mas antes dela ser passada para a função `_handle_chat_message` (que a exibe e replica).

```
def _process_incoming_message(self, message):
    # Analisa o tipo da mensagem recebida e chama o handler apropriado
    try:
        parts = message.split('|', 2) # divide em TIPO | REMETENTE | DADOS
        if len(parts) < 2:
            self._log_verbose(f"Formato de mensagem invalido: {message}")
            return

        msg_type = parts[0]
        sender_id = parts[1]

        # --- Descoberta de Peers (implicita) ---
        if sender_id != self.client_id:
            with self.mutex: # protege acesso ao conjunto de peers
                if sender_id not in self.peers:
                    self.peers.add(sender_id)
                    self._log_verbose(f"Novo acesso detectado: {sender_id}")

        # --- Roteamento da Mensagem ---
        if msg_type == "MSG" and len(parts) == 3:
            # simula delay para consistencia eventual
            delay = random.randint(0, MESSAGE_DELAY_MS) / 1000.0
            time.sleep(delay)
            content = parts[2]
            self._handle_chat_message(sender_id, content)
        elif msg_type == "REQ" and len(parts) == 3:
            try:
                req_seq_num = int(parts[2])
                self._handle_request_cs(sender_id, req_seq_num)
            except ValueError:
```

```
        # --- Roteamento da Mensagem ---
        if msg_type == "MSG" and len(parts) == 3:
            # simula delay para consistencia eventual
            delay = random.randint(0, MESSAGE_DELAY_MS) / 1000.0
            time.sleep(delay)
            content = parts[2]
            self._handle_chat_message(sender_id, content)
        elif msg_type == "REQ" and len(parts) == 3:
            try:
                req_seq_num = int(parts[2])
                self._handle_request_cs(sender_id, req_seq_num)
            except ValueError:
                self._log_verbose(f"Numero de sequencia invalido em REQ: {parts[2]}")
        elif msg_type == "REP" and len(parts) == 3:
            target_id = parts[2]
            # so processa a resposta se for para mim
            if target_id == self.client_id:
                self._handle_reply_cs(sender_id)
```

O código do delay está posicionado dentro do `if msg_type == "MSG"`, isso garante que apenas as mensagens de chat sofram esse atraso simulado, e não as mensagens de controle do sistema (como REQ e REP da exclusão mútua), que idealmente devem ser processadas o mais rápido possível. Ele ocorre depois que a mensagem foi lida da rede mas antes de ser considerada "processada" pelo cliente.

Processo de Reconciliação que sincroniza os dados entre réplicas

O processo de reconciliação neste código, é um processo que verifica a consistência localmente, ou seja, entre os múltiplos arquivos de réplica mantidos pelo mesmo cliente.

A função `_check_local_consistency(self)` é responsável por ler os arquivos de réplica do próprio cliente e compará-los.

```
def _check_local_consistency(self):
    # Verifica se os arquivos de replica locais contem o mesmo conjunto de mensagens
    # Loga detalhes apenas em modo verbose. Retorna True se consistente, False caso contrario.

    self._log_verbose("Verificando consistencia local das replicas...")
    replica_contents = [] # lista para guardar o conjunto de linhas de cada replica
    all_lines = set() # conjunto com todas as linhas unicas encontradas em todas as replicas
    consistent = True # assume consistencia inicial

    try:
        # le todas as replicas e popula os conjuntos
        for idx, replica_file in enumerate(self.replica_files):
            lines = set()
            if os.path.exists(replica_file):
                with open(replica_file, "r", encoding='utf-8') as f:
                    for line in f:
                        cleaned_line = line.strip()
                        if cleaned_line: # ignora linhas vazias
                            lines.add(cleaned_line)
                            all_lines.add(cleaned_line)
            replica_contents.append(lines)

        if not replica_contents:
            self._log_verbose("Nenhuma replica encontrada para verificacao.")
            return True # Considera consistente se nao ha arquivos

        # Compara cada replica com o conjunto agregado de todas as linhas
        for i, content_set in enumerate(replica_contents):
            if content_set != all_lines: # Se uma replica difere do total, ha inconsistencia
                consistent = False
                # Loga detalhes apenas se verbose
                missing = all_lines - content_set # Linhas que faltam nesta replica
                extra = content_set - all_lines # Linhas que so existem nesta replica (invalido/erro)
                if missing:
                    self._log_verbose(
                        f" Replica {i} ({os.path.basename(self.replica_files[i])}) faltam {len(missing)} linhas."
                    )
                if extra:
                    self._log_verbose(
                        f" Replica {i} ({os.path.basename(self.replica_files[i])}) tem {len(extra)} linhas extras ou
                    )
                # Numa implementacao real, aqui poderia haver logica para corrigir as replicas

        # Log final do resultado
        if consistent:
            self._log_verbose("Replicas locais parecem consistentes.")
        else:
            self._log_verbose("Inconsistencia detectada nas replicas locais.")
        return consistent

    except Exception as e:
        self._log_error(f"Falha ao verificar consistencia das replicas: {e}")
        return False # Retorna inconsistente em caso de erro
```

Essa função itera sobre `self.replica_files`, que contém os nomes dos arquivos de log do cliente específico, lê o conteúdo de cada um desses arquivos e armazena as linhas (mensagens) em sets. Ela também cria um conjunto `all_lines` que é a união de todas as linhas encontradas em todos os arquivos locais.

O ponto crucial é a comparação `if content_set != all_lines`: isso verifica se o conteúdo de um arquivo de réplica local específico (`content_set`) é diferente do conjunto total de todas as linhas encontradas localmente (`all_lines`). Se houver diferença, ela registra quais linhas estão faltando ou sobrando *naquele arquivo local específico*. A função retorna true se todas as réplicas locais forem idênticas e false caso contrário.

Essa função é executada periodicamente, através da função `_periodic_reconciler(self)`. A única função desta thread é chamar `_check_local_consistency()` a cada 120 segundos.

```
def _periodic_reconciler(self):
    # Thread que chama a verificação de consistência local periodicamente
    self._log_verbose("Thread Reconciliadora (local) iniciada.")
    while self.running:
        # Intervalo mais longo para não poluir logs verbose
        time.sleep(120) # Verifica a cada 2 minutos
        if self.running:
            with self.mutex: # Garante acesso seguro aos dados, se necessario
                # A verificação atual lê arquivos, talvez não precise do mutex aqui
                # Mas se fosse reconciliar/escrever, seria essencial
                self._check_local_consistency()
    self._log_verbose("Thread Reconciliadora (local) terminada.")
```

Demonstração

Alteração do valor da constante MESSAGE_DELAY_MS para melhor visualização da demonstração.

```
# --- Configuracoes ---
MULTICAST_GROUP = '224.1.1.1' # endereço IP multicast padrao
MULTICAST_PORT = 5007 # porta padrao para o grupo
REPLICATION_FACTOR = 3 # numero de arquivos de replica locais por cliente
REPLICATION_DIR = "replicas" # diretorio para guardar arquivos de replica
CHECKPOINT_DIR = "checkpoints" # diretorio para guardar arquivos de checkpoint
CHECKPOINT_INTERVAL_S = 60 # intervalo em segundos para salvar checkpoints
MESSAGE_DELAY_MS = 2000 # delay maximo artificial (ms) para simular desordem
BUFFER_SIZE = 1024 # Tamanho do buffer para recebimento de mensagens
```

Envio de diversas mensagens em um intervalo de tempo muito curto, no cliente 1 e no cliente 2.

```
Prompt de comando - python x + v
[23:38:26] brenda: boa noite!
[23:38:31] jaqueline: boa noite, brenda!
[23:38:36] rafael: boa noite!!
[23:38:48] rafael: como está o progresso do trabalho de sistemas distribuidos?
[23:38:57] jaqueline: esta 50% concluido !
[23:39:02] brenda: que bom, pessoal!

[INFO] CHECKPOINT: Estado restaurado de checkpoints\checkpoint_brenda.pkl (7 msgs).
> Digite sua mensagem (OBS. comandos reservados: "sair", "peers", "sync", "verbose"): --- Usuario brenda adicionado ao chat em grupo (224.1.1.1:5007) ---
> oi
[00:47:11] brenda: oi
> oiii
[00:47:12] brenda: oiii
> ie
[00:47:15] brenda: oiiiie
[00:47:16] jaqueline: oii
[00:47:18] jaqueline: oieo
[00:47:20] jaqueline: ieri
> teste 1
[00:48:04] brenda: teste 1
> > teste 2
[00:48:07] brenda: > teste 2
[00:48:11] jaqueline: > teste 3
> > teste 45
[00:48:12] brenda: > teste 45
>

comandos reservados: "sair", "peers", "sync", "verbose"): --- Usuario jaqueline adicionado ao chat em grupo (224.1.1.1:5007) ---
[00:47:12] brenda: oi
[00:47:13] brenda: oiii
[00:47:14] brenda: oiiiie
> oii
> oieo
[00:47:18] jaqueline: oii
> ieri
[00:47:19] jaqueline: oieo
[00:47:21] jaqueline: ieri
[00:48:04] brenda: teste 1
[00:48:06] brenda: > teste 2
> > teste 3
[00:48:10] jaqueline: > teste 3
[00:48:13] brenda: > teste 45
> |

estaurado de checkpoints\checkpoint_rafael.pkl (7 msgs).
> Digite sua mensagem (OBS. comandos reservados: "sair", "peers", "sync", "verbose"): --- Usuario rafael adicionado ao chat em grupo (224.1.1.1:5007) ---
[00:47:12] brenda: oi
[00:47:14] brenda: oiii
[00:47:16] brenda: oiiiie
[00:47:17] jaqueline: oii
[00:47:18] jaqueline: oieo
[00:47:20] jaqueline: ieri
[00:48:04] brenda: teste 1
[00:48:06] brenda: > teste 2
[00:48:11] jaqueline: > teste 3
[00:48:13] brenda: > teste 45
> |
```

Mesmo que a ordem de chegada e exibição tenha sido diferente, o sistema multicast entregou todas as mensagens para todos os três clientes. Com o tempo, e sem novas atualizações, todos os clientes atingiram um estado onde possuem o mesmo conjunto de informações. Isso é a consistência eventual: eles podem ter chegado a esse estado final por caminhos (ordens) diferentes, mas eventualmente convergem para o mesmo conjunto de dados.

CONTROLE DE CONCORRÊNCIA COM EXCLUSÃO MÚTUA DISTRIBUÍDA

Algoritmo de Ricart-Agrawala e a garantia que somente um nó por vez envia mensagens ao grupo

O Controle de Concorrência usando o algoritmo de Ricart-Agrawala está implementado em várias partes do código que trabalham juntas: variáveis de estado, funções para solicitar/liberar acesso e handlers para as mensagens de controle (REQ e REP).

No construtor, são inicializadas as variáveis de estado, responsáveis por manter o estado necessário para o algoritmo em cada cliente.

```

class chat_client:
    def __init__(self, client_id):
        self.client_id = client_id
        # --- Estado do Chat ---
        self.messages = deque(maxlen=200) # fila para exibir ultimas mensagens
        self.message_log = [] # log completo de mensagens para persistencia

        # --- Rede ---
        self.sock_send = None # socket para enviar mensagens multicast
        self.sock_rcv = None # socket para receber mensagens multicast

        # --- Estado para Ricart-Agrawala (Exclusao Mutua) ---
        self.sequence_number = 0 # numero proprio de sequencia para requisicoes
        self.highest_sequence_number = 0 # maior numero de sequencia visto na rede
        self.requesting_cs = False # se esta tentando obter acesso exclusivo
        self.outstanding_replies = set() # conjunto de peers dos quais espera resposta (REP)
        self.deferred_replies = set() # conjunto de peers cujas requisicoes (REQ) adiaram
        self.peers = set() # conjunto de ids de clientes detectados na rede
        self.mutex = threading.Lock() # lock para proteger acesso concorrente a dados compartilhados
        self.got_all_replies_event = threading.Event() # evento para sinalizar recebimento de todas as reps

```

Explicação das variáveis:

- *sequence_number* e *highest_sequence_number*: Funcionam como Timestamps Lógicos de Lamport. *sequence_number* marca a "hora lógica" da última requisição do cliente. *highest_sequence_number* rastreia a maior "hora lógica" vista em qualquer requisição na rede, usada para gerar o próximo *sequence_number*.
- *requesting_cs*: Indica se este cliente está no processo de tentar entrar na seção crítica (enviar mensagem).
- *outstanding_replies*: Guarda os IDs dos clientes dos quais este cliente ainda precisa receber uma mensagem REP (permissão) para poder entrar na seção crítica.
- *deferred_replies*: Guarda os IDs dos clientes que enviaram uma mensagem REQ para este cliente, mas cuja resposta REP foi adiada (porque este cliente tinha prioridade ou já estava na seção crítica).
- *peers*: Lista de outros clientes conhecidos na rede, para saber de quem esperar respostas.
- *mutex*: Um Lock de threading para garantir que apenas uma thread por vez modifique essas variáveis de estado compartilhadas, evitando race conditions.
- *got_all_replies_event*: Um objeto *threading.Event* que permite a uma thread (*start loop*) esperar (*wait()*) até que outra thread (*_handle_reply_cs*) sinalize (*set()*) que todas as permissões foram recebidas.

A função *_request_access(self)* é responsável pela solicitação do acesso exclusivo, para poder enviar uma mensagem ao grupo.

```

def _request_access(self):
    # Tenta obter acesso exclusivo para enviar uma mensagem
    with self.mutex: # Protege acesso as variaveis de estado de R-A
        self.requesting_cs = True
        # Incrementa o numero de sequencia (Lamport timestamp para R-A)
        self.sequence_number = self.highest_sequence_number + 1
        current_request_seq = self.sequence_number # Guarda o seq desta requisicao

        # Define de quem esperar respostas (todos os outros peers conhecidos)
        self.outstanding_replies = {p for p in self.peers if p != self.client_id}

        self.got_all_replies_event.clear() # Reseta o evento de espera

    # Se nao ha outros peers, considera acesso concedido imediatamente
    if not self.outstanding_replies:
        self._log_verbose("MUTEX: Nenhum outro usuario detectado. Acesso imediato.")
        # Nao precisa sinalizar evento pois a verificacao eh sincrona
        return True

    # Envia a requisicao para todos no grupo
    self._log_verbose(
        f"MUTEX: Solicitando acesso (Seq={current_request_seq}) para usuario: {self.outstanding_replies}")
    msg = f"REQ|{self.client_id}|{current_request_seq}"
    self._send_multicast(msg)

    # Espera pelo evento ser sinalizado (em _handle_reply_cs) com timeout
    self._log_verbose("MUTEX: Aguardando permissoes...")
    # Timeout para evitar bloqueio indefinido se um peer falhar sem responder
    got_it = self.got_all_replies_event.wait(timeout=15.0) # 15 segundos

```

```

    if not got_it:
        # Timeout ocorreu antes de receber todas as respostas
        self._log_verbose("MUTEX: Timeout ao esperar permissoes. Abortando envio.")
        with self.mutex:
            # Limpa o estado da requisicao para evitar problemas futuros
            self.requesting_cs = False
            # Libera peers que podem ter sido adiados durante a tentativa falha
            self._release_deferred()
        return False

    # Todas as permissoes recebidas
    self._log_verbose("MUTEX: Permissao concedida!")
    return True

```

Primeiramente, a função garante segurança ao modificar o estado do algoritmo Ricart-Agrawala usando um mutex. Dentro dessa proteção, ela marca que o cliente está agora tentando acesso, calcula o próximo número de sequência baseado no maior já visto, e determina quais outros usuários precisam enviar uma resposta.

Após preparar o estado da requisição, ela verifica se há outros clientes. Se não houver, a permissão é concedida imediatamente e a função retorna *True*. Caso contrário, uma mensagem REQ, contendo o ID e o número de sequência do cliente, é enviada via multicast para informar aos outros sobre a solicitação de acesso à seção crítica.

Por fim, a thread que chamou a função entra em estado de espera, aguardando um sinal (`got_all_replies_event.wait()`) que indica o recebimento de todas as respostas REP necessárias, ou até que um tempo limite (timeout) expire. Se todas as respostas chegarem a tempo, a função retorna `True`, concedendo permissão. Se ocorrer o timeout, o cliente desiste da tentativa (resetando `requesting_cs`), e a função retorna `False`.

Já a função `_handle_request_cs()` é chamada quando um cliente recebe uma mensagem REQ de outro cliente solicitando acesso. Esta função primeiro atualiza seu relógio lógico com o timestamp da requisição recebida, se for maior. Em seguida, ela aplica a regra de prioridade de Ricart-Agrawala: ela envia uma resposta REP imediatamente se não estiver tentando acessar a seção crítica ou se a requisição recebida tiver prioridade maior (menor timestamp ou menor ID em caso de empate). Caso contrário (se este cliente estiver tentando acesso e tiver prioridade maior), a resposta é adiada, e o ID do solicitante é guardado na lista `deferred_replies`.

```
def _handle_request_cs(self, sender_id, req_seq_num):
    # Processa uma requisição de acesso (REQ) recebida de outro peer
    with self.mutex: # Protege acesso as variáveis de estado R-A
        # Atualiza o maior numero de sequência visto (Timestamp de Lamport)
        self.highest_sequence_number = max(self.highest_sequence_number, req_seq_num)

        # Logica de decisao de Ricart-Agrawala:
        # Adiar a resposta (nao enviar REP agora) se:
        # 1. Eu ja estou na secao critica (nao implementado explicitamente, mas implicito por 'requesting_cs' e espera)
        # 2. Eu estou tentando entrar na secao critica ('requesting_cs == True') E
        #    a minha requisição tem prioridade maior que a do remetente.
        # Prioridade maior = (menor numero de sequencia) OU (mesmo numero de sequencia e menor ID de cliente)
        defer = False
        if self.requesting_cs:
            my_seq = self.sequence_number # Seq da minha requisição pendente
            # Compara minha prioridade com a do remetente
            if my_seq < req_seq_num or (my_seq == req_seq_num and self.client_id < sender_id):
                # Minha requisição tem prioridade, entao adio a resposta para ele
                defer = True

        if defer:
            self._log_verbose(f"MUTEX: Adiando REP para {sender_id} (Minha Req={my_seq} vs Req Dele={req_seq_num})")
            self.deferred_replies.add(sender_id) # Adiciona a lista de respostas a enviar depois
        else:
            # Envia a resposta (REP) imediatamente (nao estou pedindo ou ele tem prioridade)
            self._log_verbose(f"MUTEX: Enviando REP para {sender_id} (Nao peço ou ele tem prioridade)")
            reply_msg = f"REP|{self.client_id}|{sender_id}" # TARGET_ID e quem pediu
            self._send_multicast(reply_msg)
```

Chamada quando uma mensagem REP destinada a este cliente é recebida, a função `_handle_reply_cs` verifica se o cliente atual está realmente aguardando permissões e se a resposta veio de um peer esperado. Em caso afirmativo, o remetente é removido da lista de permissões pendentes. Quando essa lista fica vazia, significa que todas as permissões foram concedidas, então a função sinaliza o evento `got_all_replies_event.set()`, desbloqueando a thread principal que estava esperando para entrar na seção crítica.


```
def _handle_reply_cs(self, sender_id):
    # Processa uma resposta de permissao (REP) recebida.
    with self.mutex: # Protege acesso a outstanding_replies
        # Verifica se ainda estou esperando resposta e se veio de um peer esperado
        if self.requesting_cs and sender_id in self.outstanding_replies:
            self._log_verbose(f"MUTEX: Permissao (REP) recebida de {sender_id}")
            self.outstanding_replies.remove(sender_id)
            self._log_verbose(f"MUTEX: Permissoes restantes: {len(self.outstanding_replies)}")

        # Se nao ha mais respostas pendentes, sinaliza o evento para liberar a thread principal
        if not self.outstanding_replies:
            self.got_all_replies_event.set()
```

A liberação do acesso é feita após o cliente ter terminado a seção crítica (enviado sua mensagem), que chama `_release_access()`. Esta função marca que o cliente não está mais solicitando acesso e imediatamente chama `_release_deferred()`. Esta última verifica se há alguma resposta REP que foi adiada anteriormente; se houver, ela envia agora as mensagens REP para todos os clientes na lista `deferred_replies`, permitindo que eles prossigam com suas próprias requisições, e por fim limpa essa lista.

```
def _release_access(self):
    # Libera o acesso a secao critica e envia respostas adiadas
    with self.mutex: # Protege acesso a requesting_cs e deferred_replies
        self.requesting_cs = False # Marca que nao estou mais (tentando) na secao critica
        self._release_deferred() # Envia as respostas que foram adiadas

def _release_deferred(self):
    # Envia mensagens REP para todos os peers na lista de adiados.
    # Esta funcao deve ser chamada DENTRO de um bloco 'with self.mutex:'
    if self.deferred_replies:
        self._log_verbose(f"MUTEX: Liberando acesso. Enviando REPs adiados para: {self.deferred_replies}")
        # Itera sobre uma copia para poder modificar o conjunto original
        for target_id in list(self.deferred_replies):
            reply_msg = f"REP|{self.client_id}|{target_id}"
            self._send_multicast(reply_msg)
        self.deferred_replies.clear() # Limpa a lista de adiados
```

Por fim, a integração no fluxo de envio de mensagem é feita no start. No loop principal que lê o input do usuário, o envio da mensagem de chat é "envelopado" pelas chamadas de `_request_access` e `_release_access`.

O trecho abaixo está contido na função `start()`, e captura as entradas realizadas pelo usuário, tratando também os possíveis erros.

```

# --- Loop de Input do Usuario ---
try:
    while self.running:
        # Exibe o prompt apropriado (detalhado na primeira vez, curto depois)
        self._redisplay_prompt()
        try:
            # Espera pelo input do usuario
            user_input = input()
        except EOFError: # Captura Ctrl+D em terminais Linux/macOS
            user_input = "sair" # Trata como o comando sair
            print("sair") # Ecoa para clareza, pois input() nao retorna nada com EOF

        # Verifica se o cliente foi parado enquanto esperava input (raro, mas possivel)
        if not self.running: break

        # Processa o input
        cmd = user_input.lower().strip() # Converte para minusculas e remove espacos extras

        if cmd == "sair":
            self.stop() # Inicia o processo de parada
            break # Sai do loop de input
        elif cmd == "sync":
            # Comando para forçar verificacao de consistencia local
            self._log_info("Forçando verificacao de consistencia local...")
            # A funcao _check_local_consistency loga detalhes apenas se verbose
            with self.mutex: # Adquire lock se a funcao de checagem precisar
                self._check_local_consistency()
            # Continua para o proximo prompt sem enviar mensagem
            continue

```

```

        elif cmd == "verbose":
            # Comando para alternar o modo verbose
            self.verbose_mode = not self.verbose_mode # Inverte o valor da flag
            status = "ATIVADO" if self.verbose_mode else "DESATIVADO"
            self._log_info(f"Modo Detalhado (verbose) {status}.")
            continue # Continua para o proximo prompt

# Se nao for um comando conhecido e nao for vazio, trata como mensagem de chat
elif user_input:
    self._log_verbose("Tentando enviar mensagem...")
    # 1. Tenta obter acesso exclusivo usando Ricart-Agrawala
    if self._request_access():
        # 2. Se obteve acesso, envia a mensagem
        self._log_verbose("Permissao OK. Enviando...")
        msg_content = user_input # Mantem a capitalizacao original do usuario
        message = f"MSG[{self.client_id}] {msg_content}"
        self._send_multicast(message)

        # Nota: A mensagem enviada sera recebida pelo proprio cliente
        # atraves do loop de recebimento multicast e sera processada
        # por _handle_chat_message, aparecendo na tela e sendo replicada.

        # 3. Libera o acesso para outros peers
        self._release_access()
    else:
        # Nao obteve acesso (provavelmente devido a timeout)
        self._log_info("Nao foi possivel obter permissao para enviar agora. Tente novamente.")

# Se o input foi vazio (usuario apenas pressionou Enter), o loop continua
# e o prompt sera redesenhado.

```

Antes que uma mensagem de chat (MSG) do usuário possa ser enviada com `_send_multicast`, o código primeiro executa `_request_access()`. Somente se esta função retornar `True` (indicando que a permissão distribuída foi obtida com sucesso), a mensagem de chat é enviada. Logo após o envio, `_release_access()` é chamado para permitir que outros clientes possam obter acesso. Isso garante a exclusão mútua para a ação de enviar uma mensagem de chat.

Exibição de mensagens de requisição e concessão do recurso

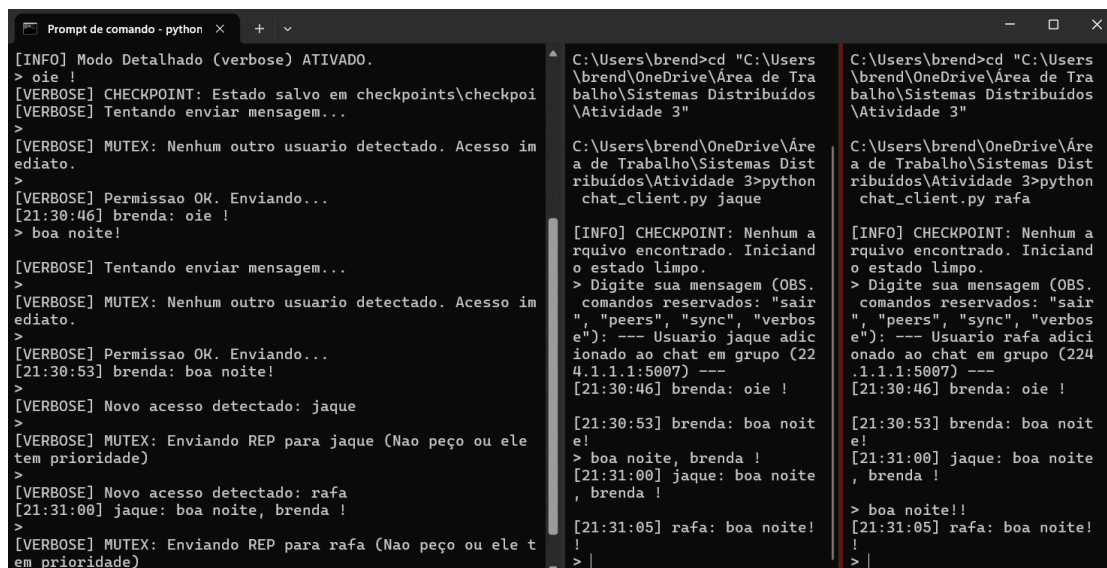
A fim de melhor visualização do processo e obter os logs de controle de acesso, implementamos em nosso algoritmo o modo “*verbose*”. As mensagens de sistema, incluindo as do Ricart-Agrawala, só aparecem se este modo estiver ativo.

A função responsável pela exibição das mensagens é a `_log_verbose()`. Ela faz uma verificação se a variável `verbose_mode` tem o valor `True`, caso sim, exibe a mensagem.

```
def _log_verbose(self, message):
    # Imprime a mensagem apenas se o modo verbose estiver ativo
    if self.verbose_mode:
        print(f"\n[VERBOSE] {message}") # imprime em nova linha
        self._redisplay_prompt() # redesenha o prompt
```

As mensagens específicas sobre o estado do algoritmo Ricart-Agrawala são exibidas nos seguintes pontos, mas apenas se o modo verbose estiver ativo: ao iniciar uma requisição (`_request_access`); ao lidar com uma requisição recebida (`_handle_request_cs`); ao receber uma resposta/concessão (`_handle_reply_cs`); ao finalizar a requisição (`_request_access`) e ao liberar o acesso (`_release_access` e `_release_deferred`).

Demonstração



```
Prompt de comando - python x + v
[INFO] Modo Detalhado (verbose) ATIVADO.
> oi e !
[VERBOSE] CHECKPOINT: Estado salvo em checkpoints\checkpoi
[VERBOSE] Tentando enviar mensagem...
>
[VERBOSE] MUTEX: Nenhum outro usuario detectado. Acesso im
ediato.
>
[VERBOSE] Permissao OK. Enviando...
[21:30:46] brenda: oi e !
> boa noite!

[VERBOSE] Tentando enviar mensagem...
>
[VERBOSE] MUTEX: Nenhum outro usuario detectado. Acesso im
ediato.
>
[VERBOSE] Permissao OK. Enviando...
[21:30:53] brenda: boa noite!
>
[VERBOSE] Novo acesso detectado: jaque
[VERBOSE] MUTEX: Enviando REP para jaque (Nao peço ou ele
tem prioridade)
>
[VERBOSE] Novo acesso detectado: rafa
[21:31:00] jaque: boa noite, brenda !
>
[VERBOSE] MUTEX: Enviando REP para rafa (Nao peço ou ele t
em prioridade)

C:\Users\brend>cd "C:\Users
\brend\OneDrive\Área de Tra
balho\Sistemas Distribuídos
\Atividade 3"

C:\Users\brend\OneDrive\Áre
a de Trabalho\Sistemas Dist
ribuídos\Atividade 3>python
chat_client.py jaque

[INFO] CHECKPOINT: Nenhum a
rquivo encontrado. Iniciand
o estado limpo.
> Digite sua mensagem (OBS.
comandos reservados: "sair
", "peers", "sync", "verbos
e"): --- Usuario jaque adici
onado ao chat em grupo (22
4.1.1.1:5007) ---
[21:30:46] brenda: oi e !

[21:30:53] brenda: boa noit
e!
> boa noite, brenda !
[21:31:00] jaque: boa noite
, brenda !

[21:31:05] rafa: boa noite!
!
> |

C:\Users\brend>cd "C:\Users
\brend\OneDrive\Área de Tra
balho\Sistemas Distribuídos
\Atividade 3"

C:\Users\brend\OneDrive\Áre
a de Trabalho\Sistemas Dist
ribuídos\Atividade 3>python
chat_client.py rafa

[INFO] CHECKPOINT: Nenhum a
rquivo encontrado. Iniciand
o estado limpo.
> Digite sua mensagem (OBS.
comandos reservados: "sair
", "peers", "sync", "verbos
e"): --- Usuario rafa adici
onado ao chat em grupo (224
.1.1.1:5007) ---
[21:30:46] brenda: oi e !

[21:30:53] brenda: boa noit
e!
[21:31:00] jaque: boa noite
, brenda !

> boa noite!!
[21:31:05] rafa: boa noite!
!
> |
```

Modo *verbose* ativado, demonstrando o funcionamento do envio de mensagens em multicast, utilizando o algoritmo de Ricart-Agrawala para controle de concorrência.

TOLERÂNCIA A FALHAS COM CHECKPOINTS E ROLLBACK

Snapshots do estado do cliente (mensagens enviadas/recebidas)

A cada 60 segundos, ou conforme configurado, o cliente salva seu estado atual em um arquivo binário usando o módulo pickle. Isso é essencial para registrar todas as mensagens enviadas e recebidas, garantindo que as informações estejam sempre atualizadas.

O código que gerencia os snapshots do estado do cliente está na função *save_checkpoint*. Inicialmente, fizemos a definição das constantes e variáveis, que, respectivamente, representam os parâmetros de configuração do checkpoint (diretório e frequência) e quais dados serão salvos no snapshot, esses últimos estão definidos no *__init__*.

```
# --- Configuracoes ---
MULTICAST_GROUP = '224.1.1.1' # endereço IP multicast padrao
MULTICAST_PORT = 5007 # porta padrao para o grupo
REPLICATION_FACTOR = 3 # numero de arquivos de replica locais por cliente
REPLICATION_DIR = "replicas" # diretorio para guardar arquivos de replica
CHECKPOINT_DIR = "checkpoints" # diretorio para guardar arquivos de checkpoint
CHECKPOINT_INTERVAL_S = 60 # intervalo em segundos para salvar checkpoints
MESSAGE_DELAY_MS = 50 # delay maximo artificial (ms) para simular desordem
BUFFER_SIZE = 1024 # Tamanho do buffer para recebimento de mensagens
```

O estado salvo inclui detalhes como o histórico de mensagens, que registra quem enviou cada mensagem e quando, além do número de sequência atual do algoritmo Ricart-Agrawala e a lista de peers conhecidos. Essas informações permitem que todos os clientes se mantenham sincronizados. Os dados são armazenados em um arquivo no diretório *checkpoints* (constante *CHECKPOINT_DIR*), com o formato checkpoint <client_id>.pkl.

```

def _save_checkpoint(self):
    # Salva o estado relevante do cliente em um arquivo pickle.
    state_to_save = {} # Dicionario para guardar o estado
    with self.mutex: # Garante que o estado nao mude durante a copia
        state_to_save = {
            'client_id': self.client_id,
            'message_log': list(self.message_log), # Salva o log completo de mensagens
            'sequence_number': self.sequence_number, # ultimo numero de sequencia usado/visto
            'highest_sequence_number': self.highest_sequence_number,
            'peers': list(self.peers) # Lista de peers conhecidos
            # Nao salva estados volateis como outstanding_replies, deferred_replies, requesting_cs
        }
    try:
        # 'wb' = Write Bytes (necessario para pickle)
        with open(self.checkpoint_file, 'wb') as f:
            pickle.dump(state_to_save, f) # Serializa e salva o dicionario
            self._log_verbose(f"CHECKPOINT: Estado salvo em {self.checkpoint_file}")
    except Exception as e:
        self._log_error(f"Falha ao salvar checkpoint: {e}")

```

Para evitar problemas durante a gravação, a serialização é protegida por um mutex. Uma thread dedicada, chamada *_periodic_checkpoint*, cuida desse processo em segundo plano, garantindo que os snapshots sejam criados continuamente e sem interrupções, o que contribui para a estabilidade do sistema. A thread de checkpoint automático executa *_save_checkpoint* periodicamente a cada 60 segundos, conforme definido na constante *CHECKPOINT_INTERVAL_S*.

```

def _periodic_checkpoint(self):
    # Thread que chama a funcao de salvar checkpoint periodicamente.
    self._log_verbose("Thread de Checkpoint iniciada.")
    while self.running:
        # Espera o intervalo definido
        time.sleep(CHECKPOINT_INTERVAL_S)
        # Verifica se ainda estamos rodando antes de salvar
        # (evita salvar apos o comando 'sair' ser dado mas antes da thread parar)
        if self.running:
            self._save_checkpoint()
    self._log_verbose("Thread de Checkpoint terminada.")

```

Para evitar perda de mensagens, quando o usuário escolhe sair, ou o programa é interrompido de forma controlada, dentro da função *stop()* é chamada a função *_save_checkpoint*.

```
def stop(self):
    # Inicia o processo de parada limpa do cliente.
    # Evita chamadas multiplas se stop() ja estiver em execucao
    if not self.running:
        return

    print(f"\nEncerrando cliente {self.client_id}...")
    self.running = False # Sinaliza para as threads pararem seus loops

    # Tenta um ultimo checkpoint antes de sair
    self._log_verbose("Salvando checkpoint final...")
    self._save_checkpoint()
```

Restauração do estado salvo no último checkpoint, ao reiniciar após falha

A restauração do estado a partir do último checkpoint salvo acontece automaticamente quando o cliente é iniciado, e a verificação é realizada logo na função `__init__`.

```
# --- Persistencia e Replicacao ---
self.checkpoint_file = os.path.join(CHECKPOINT_DIR, f"checkpoint_{self.client_id}.pkl")
self.replica_files = [
    os.path.join(REPLICATION_DIR, f"replica_{self.client_id}_{i}.log")
    for i in range(REPLICATION_FACTOR)
]

# --- Controle de Exibicao e Threads (...) ---

# --- Inicializacao ---
self._setup_directories() # cria os diretorios de replicas e checkpoints
self._load_checkpoint() # carrega o estado anterior de um checkpoint
self._setup_sockets() # configura os sockets de envio e recebimento
self.peers.add(self.client_id) # adiciona a si mesmo a lista de peers
```

Na função `_load_checkpoint`, responsável pela restauração do histórico de mensagens, primeiro é realizada uma verificação se existe um arquivo de checkpoint correspondente ao seu ID. Se encontrar esse arquivo, ele começa o processo de recuperação. Se tudo estiver certo, o cliente restaura o histórico de mensagens, os números de sequência e a lista de peers. Ele também reconstrói a fila de exibição, formatando as mensagens com timestamps que são mais fáceis de entender.

```
def _load_checkpoint(self):
    # Carrega o estado do cliente do arquivo de checkpoint, se existir e for valido.
    if os.path.exists(self.checkpoint_file):
        try:
            # 'rb' = Read Bytes
            with open(self.checkpoint_file, 'rb') as f:
                state = pickle.load(f) # Carrega e deserializa o dicionario

            # Validação básica: pertence a este cliente?
            if state.get('client_id') == self.client_id:
                # Restaura o estado
                self.message_log = state.get('message_log', [])
                self.sequence_number = state.get('sequence_number', 0)
                self.highest_sequence_number = state.get('highest_sequence_number', 0)
                loaded_peers = state.get('peers', [])
                self.peers = set(loaded_peers)
                self.peers.add(self.client_id) # Garante que o próprio ID esta presente

                # Preenche a deque de exibição com base no log restaurado
                self.messages.clear()
                num_restored = 0
                for sender, content, ts in self.message_log:
                    timestamp_str = time.strftime('%H:%M:%S', time.localtime(ts))
                    self.messages.append(f"[{timestamp_str}] {sender}: {content}")
                    num_restored += 1
```

```
        # exibe mensagens restauradas
        if num_restored > 0:
            # itera sobre uma copia da deque para evitar problemas de modificacao
            for msg in list(self.messages):
                print(msg) # imprime cada mensagem restaurada

            self._log_info(f"CHECKPOINT: Estado restaurado de {self.checkpoint_file} ({num_restored} msgs).")
            # Loga detalhes adicionais apenas se verbose
            self._log_verbose(
                f" -> Seq Num: {self.sequence_number}, Highest Seen: {self.highest_sequence_number}")
            self._log_verbose(f" -> Peers conhecidos: {self.peers}")
        else:
            # Arquivo existe mas e de outro ID, informa e ignora
            self._log_info(
                f"CHECKPOINT: Arquivo encontrado pertence a outro ID ({state.get('client_id')}). Ignorando.")

    except Exception as e:
        # Erro ao ler ou deserializar o arquivo
        self._log_error(f"Falha ao carregar checkpoint ({self.checkpoint_file}): {e}. Iniciando estado limpo.")
        self.message_log = [] # Garante estado limpo em caso de falha
        self.peers = {self.client_id}
    else:
        # Nenhum arquivo de checkpoint encontrado
        self._log_info("CHECKPOINT: Nenhum arquivo encontrado. Iniciando estado limpo.")
```

Porém, se o arquivo estiver corrompido ou for de outro cliente, o sistema começa do zero. Isso garante que, pelo menos, uma recuperação básica aconteça, evitando problemas maiores e permitindo que o cliente volte a funcionar de forma segura.

Arquivos de checkpoint gerados

Os checkpoints são arquivos binários gerados periodicamente contendo o estado completo do cliente. Cada cliente tem seu próprio arquivo no formato:

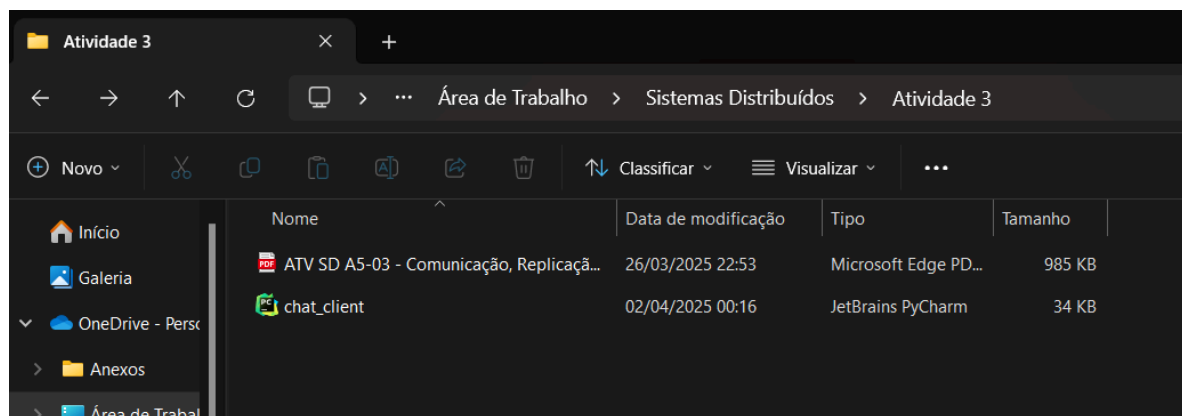
`checkpoints/checkpoint_<client_id>.pkl`

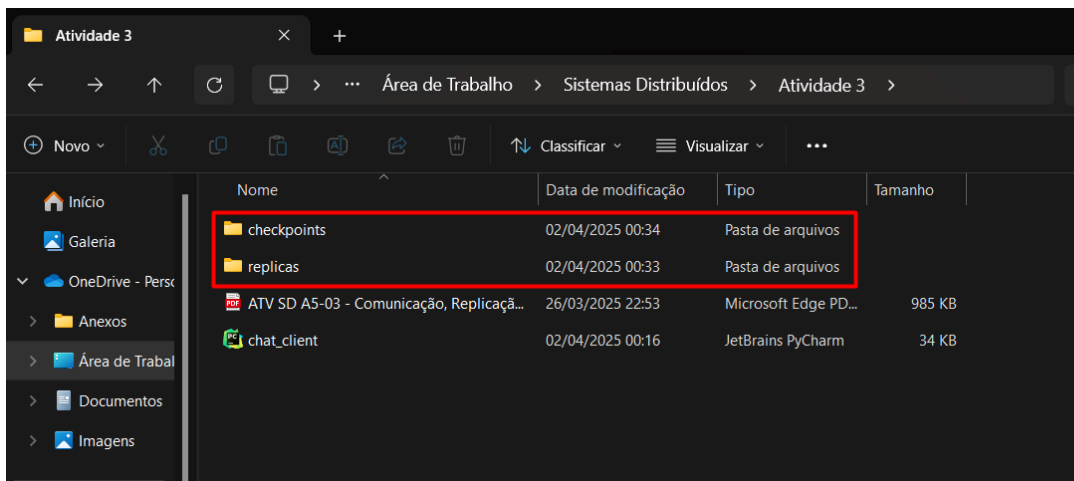
A geração dos arquivos de checkpoint é realizada principalmente pela função `_save_checkpoint`, que coleta o estado atual e o escreve no arquivo `.pkl` (pickle).

Os dados são compactados e armazenados em um arquivo binário no formato pickle, que mantém a estrutura das informações intacta. O arquivo é salvo na pasta "checkpoints" com um nome que inclui o ID do usuário, garantindo que cada cliente tenha seu próprio ponto de recuperação.

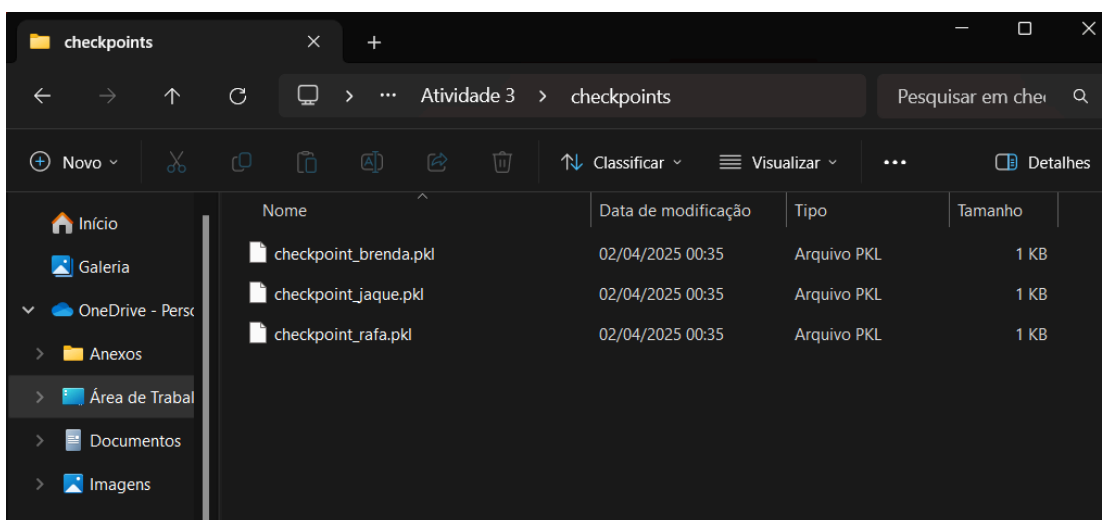
Demonstração

Pasta do sistema antes da primeira inicialização:

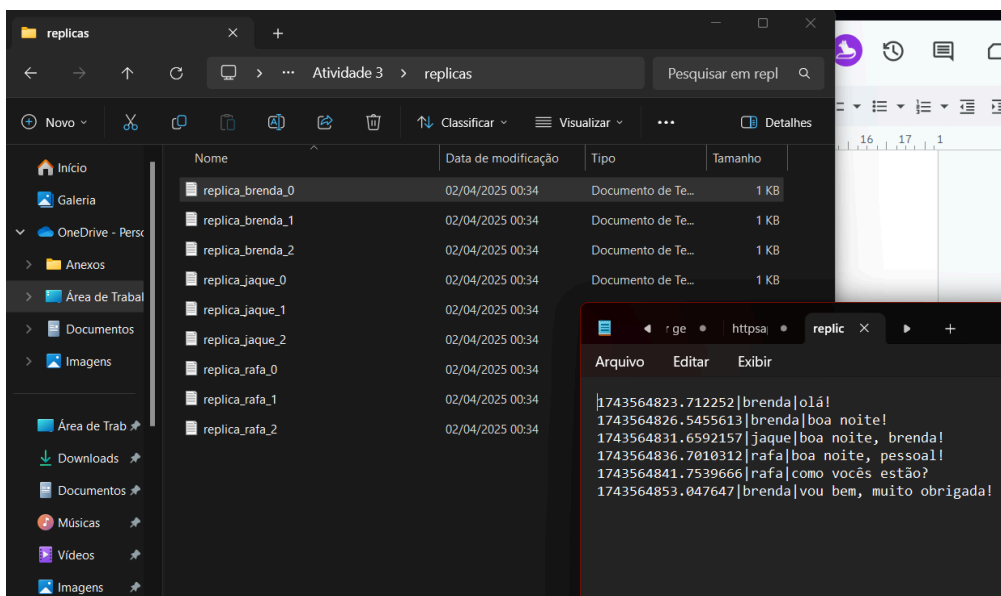




Arquivos de checkpoint:



Arquivos de Réplica:



Simulação de falha e restauração:

```
Prompt de Comando - pythor  X  +  v

[00:33:43] brenda: olá!
> boa noite!
[00:33:46] brenda: boa noite!
[00:33:51] jaque: boa noite, brenda!
[00:33:56] rafa: boa noite, pessoal!
[00:34:01] rafa: como vocês estão?
> vou bem, muito obrigada!
[00:34:13] brenda: vou bem, muito obrigada!

>
Saíndo ...

Encerrando cliente brenda...
Cliente brenda encerrado.

C:\Users\brend\OneDrive\Área de Trabalho\Sistemas Distribu
ídos\Atividade 3>python chat_client.py brenda
[00:33:43] brenda: olá!
[00:33:46] brenda: boa noite!
[00:33:51] jaque: boa noite, brenda!
[00:33:56] rafa: boa noite, pessoal!
[00:34:01] rafa: como vocês estão?
[00:34:13] brenda: vou bem, muito obrigada!

[INFO] CHECKPOINT: Estado restaurado de checkpoints\checkp
oint_brenda.pkl (6 msgs).
> Digite sua mensagem (OBS. comandos reservados: "sair", "
peers", "sync", "verbose"): --- Usuario brenda adicionado
ao chat em grupo (224.1.1.1:5007) ---
>
```