

Chat Mejorado en Python: Mensajes Privados y Sincronización con Lock

GÓMEZ EDGAR, Universidad Central de Venezuela, Venezuela

Se presenta una versión mejorada de un sistema de chat distribuido en Python basado en arquitectura cliente-servidor. La nueva implementación incorpora funcionalidades de mensajería privada entre usuarios y utiliza mecanismos de sincronización (Lock) para prevenir condiciones de carrera en el acceso concurrente a las estructuras de datos compartidas. El servidor maneja múltiples conexiones mediante sockets TCP y hilos, implementando tanto la retransmisión pública de mensajes como la entrega selectiva de mensajes privados. Se detallan las modificaciones en el protocolo de comunicación, la estructura de clases extendida y las técnicas de sincronización empleadas. El trabajo ejemplifica conceptos avanzados de sistemas distribuidos como concurrencia segura, comunicación selectiva y control de acceso a recursos compartidos.

CCS Concepts: • **Organización de sistemas informáticos** → **Sistemas Distribuidos**; *Cliente Servidor*; *Concurrencia*; *Sincronización*.

Additional Key Words and Phrases: Sistemas Distribuidos, Cliente Servidor, Concurrencia, Sincronización, Mensajes Privados

ACM Reference Format:

GÓMEZ EDGAR. 2026. Chat Mejorado en Python: Mensajes Privados y Sincronización con Lock. *Proc. ACM Hum.-Comput. Interact.* 1, 1, Article 1 (January 2026), 8 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

En este artículo presentamos una mejora significativa al sistema de chat básico previamente implementado. La nueva versión incorpora dos funcionalidades críticas: (1) la capacidad de enviar mensajes privados entre usuarios específicos, y (2) el uso de mecanismos de sincronización (Lock) para prevenir condiciones de carrera en el acceso concurrente a las estructuras de datos compartidas. Estas mejoras transforman el sistema de un chat público simple a una plataforma de comunicación más robusta y versátil, mientras que abordan problemas fundamentales de la programación concurrente en sistemas distribuidos.

1. Mejoras Implementadas

1.1. Mensajes Privados

Se ha extendido el protocolo de comunicación para permitir dos tipos de mensajes:

1. **Mensajes públicos:** Se envían a todos los usuarios conectados (funcionalidad original).
2. **Mensajes privados:** Se envían únicamente a un usuario específico, utilizando una sintaxis especial.

La sintaxis para mensajes privados es: `\w nombre_usuario_destino mensaje`

1.2. Sincronización con Lock

Dado que múltiples hilos acceden concurrentemente a las listas de clientes (`clients_sockets_list` y `clients_names_list`), se ha introducido un objeto Lock para sincronizar el acceso. Esto previene condiciones de carrera que podrían ocurrir cuando:

- Un nuevo cliente se conecta (añadiendo elementos a las listas).
- Un cliente se desconecta (removiendo elementos de las listas).
- Se busca un cliente para enviar un mensaje privado (consultando las listas).
- Se itera sobre las listas para enviar mensajes de broadcast.

Author's Contact Information: GÓMEZ EDGAR, Universidad Central de Venezuela, Caracas, Venezuela.

2026. ACM 2573-0142/2026/1-ART1
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

2. Implementación del Servidor Mejorado

2.1. Estructura de la Clase ChatServer Mejorada

Se ha modificado la clase ChatServer para incorporar las nuevas funcionalidades:

Listing 1. Constructor mejorado de ChatServer

```

1      class ChatServer:
2          def __init__(self, host_ip=socket.gethostname(), host_port=12345):
3              self.clients_sockets_list = []
4              self.clients_names_list = []
5              self.clients_lock = threading.Lock() # Nuevo: Lock para sincronización
6              self.server_socket = None
7
8              self.HOST_IP = host_ip
9              self.HOST_PORT = host_port
10             self.ENCODER = "utf-8"
11             self.BUFFER_SIZE = 1024
12

```

La principal adición es `self.clients_lock`, un objeto de sincronización que garantiza acceso exclusivo a las listas compartidas de usuarios y sockets de usuario.

2.2. Detección de Mensajes Privados

Se ha añadido un método para identificar mensajes privados:

Listing 2. Detección de mensajes privados

```

1      def is_private_message(self, message):
2          return message.startswith("\w")
3

```

Este método verifica si un mensaje comienza con la secuencia especial `\w` que indica un mensaje privado.

2.3. Envío de Mensajes Privados

El método `send_private_message()` maneja la entrega selectiva de mensajes:

Listing 3. Envío de mensajes privados

```

1      def send_private_message(self, from_client, to_client, message):
2          with self.clients_lock:
3              if to_client in self.clients_names_list:
4                  to_index = self.clients_names_list.index(to_client)
5                  to_socket = self.clients_sockets_list[to_index]
6              else:
7                  to_index = None
8                  to_socket = None
9
10             if from_client in self.clients_names_list:
11                 from_index = self.clients_names_list.index(from_client)
12                 from_socket = self.clients_sockets_list[from_index]
13             else:
14                 from_index = None

```

```

15         from_socket = None
16
17     if to_socket:
18         private_message = f"priv({from_client}): {message}"
19         try:
20             to_socket.send(private_message.encode(self.ENCODER))
21             print(f"Private message from '{from_client}' to '{to_client}': {message}")
22         except Exception as ex:
23             print(f"Socket error: {ex}")
24     elif from_socket:
25         error_message = f"User '{to_client}' not found."
26         try:
27             from_socket.send(error_message.encode(self.ENCODER))
28             print(f"Private message error: User '{to_client}' not found for
29 '{from_client}'")
30         except Exception as ex:
31             print(f"Socket error: {ex}")

```

Características importantes:

- Utiliza `with self.clients_lock:` para acceso seguro a las listas.
- Busca el socket del destinatario en las listas de clientes.
- Envía el mensaje solo al destinatario con formato especial `priv(remite): mensaje`.
- Maneja el caso donde el destinatario no existe, notificando al remitente.

2.4. Broadcast Mejorado con Sincronización

El método `broadcast_message()` ha sido actualizado para usar el lock:

Listing 4. Broadcast con sincronización

```

1  def broadcast_message(self, message):
2      with self.clients_lock:
3          sockets_snapshot = list(self.clients_sockets_list)
4
5      for client_socket in sockets_snapshot:
6          try:
7              client_socket.send(message.encode(self.ENCODER))
8          except Exception as ex:
9              print(f"Socket error: {ex}")
10             with self.clients_lock:
11                 if client_socket in self.clients_sockets_list:
12                     client_index = self.clients_sockets_list.index(client_socket)
13                     client_name = self.clients_names_list[client_index]
14                     self.clients_sockets_list.remove(client_socket)
15                     self.clients_names_list.remove(client_name)
16             client_socket.close()
17             self.broadcast_message(f"{client_name} has left the chat.\n")
18

```

Mejoras implementadas:

- Se crea una copia de la lista de sockets dentro del contexto del lock.
- Se itera sobre la copia en lugar de la lista original, evitando problemas si la lista cambia durante la iteración.
- El manejo de errores también utiliza el lock para modificar las listas de manera segura.

2.5. Recepción de Mensajes Mejorada

El método `recieve_message()` ahora distingue entre mensajes públicos y privados:

Listing 5. Recepción de mensajes mejorada

```

1  def recieve_message(self, client_socket):
2      while True:
3          try:
4              message = client_socket.recv(self.BUFFER_SIZE).decode(self.ENCODER)
5              if message == "QUIT":
6                  with self.clients_lock:
7                      client_index = self.clients_sockets_list.index(client_socket)
8                      client_name = self.clients_names_list[client_index]
9                      self.clients_sockets_list.remove(client_socket)
10                     self.clients_names_list.remove(client_name)
11                     client_socket.close()
12                     self.broadcast_message(f"{client_name} has left the chat.\n")
13             elif self.is_private_message(message):
14                 parts = message.split(' ', 2)
15                 if len(parts) >= 3:
16                     to_client = parts[1]
17                     private_message = parts[2]
18                     with self.clients_lock:
19                         client_index = self.clients_sockets_list.index(client_socket)
20                         from_client = self.clients_names_list[client_index]
21                     self.send_private_message(from_client, to_client, private_message)
22             else:
23                 with self.clients_lock:
24                     client_index = self.clients_sockets_list.index(client_socket)
25                     client_name = self.clients_names_list[client_index]
26                     full_message = f"{client_name}: {message}"
27                     print(full_message)
28                     self.broadcast_message(full_message)
29
30         except Exception as ex:
31             print(f"Socket error: {ex}")
32             with self.clients_lock:
33                 if client_socket in self.clients_sockets_list:
34                     client_index = self.clients_sockets_list.index(client_socket)
35                     client_name = self.clients_names_list[client_index]
36                     self.clients_sockets_list.remove(client_socket)
37                     self.clients_names_list.remove(client_name)
38                 else:
39                     client_name = "Unknown"
40             client_socket.close()
41             self.broadcast_message(f"{client_name} has left the chat.\n")

```

```

42         break
43

```

Flujo mejorado:

1. Si el mensaje es "QUIT", maneja la desconexión ordenada con sincronización.
2. Si es un mensaje privado (detectado por `is_private_message()`), extrae destinatario y mensaje, luego llama a `send_private_message()`.
3. Si es un mensaje público normal, lo retransmite a todos.
4. Todas las operaciones sobre las listas utilizan el lock para sincronización.

2.6. Conexión de Clientes con Sincronización

El método `connect_client()` también ha sido actualizado para menjar concurrencia con los recursos compartidos:

Listing 6. Conexión de clientes con sincronización

```

1  def connect_client(self):
2      while True:
3          client_socket, client_address = self.server_socket.accept()
4          print(f"Connection established with {client_address}...")
5
6          name_flag = "NAME"
7          client_socket.send(name_flag.encode(self.ENCODER))
8          client_name = client_socket.recv(self.BUFFER_SIZE).decode(self.ENCODER)
9
10         with self.clients_lock: # Sincronización al añadir nuevo cliente
11             self.clients_sockets_list.append(client_socket)
12             self.clients_names_list.append(client_name)
13
14         print(f"Client '{client_name}' connected.")
15         welcome_message = f"Welcome to the chat, {client_name}!\n"
16         client_socket.send(welcome_message.encode(self.ENCODER))
17         self.broadcast_message(f"{client_name} has joined the chat.\n")
18
19         recieve_message_thread = threading.Thread(target=self.recieve_message,
20             args=(client_socket,))
21         recieve_message_thread.start()

```

3. Protocolo de Comunicación Extendido

El protocolo ahora soporta tres tipos de mensajes:

1. **Conexión inicial y handshake de nombre** (sin cambios):
 - Servidor → Cliente: "NAME"
 - Cliente → Servidor: nombre del usuario
2. **Mensajes públicos** (sin cambios):
 - Cliente → Servidor: mensaje de texto
 - Servidor → Todos: "nombre: mensaje"
3. **Mensajes privados** (nuevo):

- Cliente → Servidor: "\w nombre_usuario_destino mensaje"
- Servidor → Destinatario: "priv(nombre_remitente): mensaje"
- Servidor → Remitente (si destinatario no existe): "User 'nombre_destino' not found."

4. Desconexión (mejorada con sincronización):

- Cliente → Servidor: "QUIT"
- Servidor (con lock): Remueve cliente de listas
- Servidor → Todos: "nombre ha left the chat."

4. Problemas de Concurrency Resueltos

4.1. Condiciones de Carrera

En la versión original, múltiples hilos podían acceder simultáneamente a las listas de clientes, causando condiciones de carrera como:

- Un hilo iterando sobre la lista mientras otro añade/elimina elementos.
- Dos hilos intentando eliminar el mismo cliente simultáneamente.
- Un hilo buscando un cliente que otro hilo está eliminando.

4.2. Solución con Lock

El uso de `threading.Lock()` proporciona:

1. **Exclusión mutua:** Solo un hilo puede ejecutar el código protegido por el lock a la vez.
2. **Consistencia:** Las operaciones sobre las listas son atómicas desde la perspectiva de otros hilos.
3. **Prevención de corrupción de datos:** Evita que las listas queden en estado inconsistente.

4.3. Patrón "Snapshot"

En el método `broadcast_message()`, se utiliza el patrón de crear una copia (snapshot) de la lista dentro del contexto del lock:

Listing 7. Patrón snapshot para iteración segura

```

1  with self.clients_lock:
2      sockets_snapshot = list(self.clients_sockets_list)
3      # Iterar sobre la copia fuera del lock
4      for client_socket in sockets_snapshot:
5          # Enviar mensajes...
6  
```

Esto permite:

- Mantener el lock por el menor tiempo posible (solo para crear la copia).
- Evitar bloquear otras operaciones durante el envío de mensajes.
- Prevenir problemas si la lista cambia durante la iteración.

5. Ventajas de la Implementación Mejorada

- **Funcionalidad extendida:** Soporte para mensajes privados además de públicos.
- **Concurrency segura:** Uso de locks para prevenir condiciones de carrera.
- **Robustez mejorada:** Manejo más seguro de desconexiones concurrentes.
- **Escalabilidad:** La sincronización permite mayor número de clientes concurrentes sin corrupción de datos.
- **Modularidad:** Separación clara entre lógica de mensajes públicos y privados.

6. Ejecución del Chat Básico con mensajes privados y sincronización de recursos

Ejecutamos el servidor, donde podemos ver la IP del servidor y el puerto Fig 1.

```
PS C:\Users\egomez\Desktop\distributedPostgrade\Tarea_Chat_Basico_Privados> python .\chat_server.py
Server listening on 172.17.192.1:12345
```

Fig. 1

Iniciamos tres clientes "Luis" Fig 2, "Pedro" Fig 3 y "Ana" Fig 4.

```
PS C:\Users\egomez\Desktop\distributedPostgrade\Tarea_Chat_Basico_Privados> python .\chat_client.py
Enter server IP address: 172.17.192.1
Enter server port: 12345
Enter your name: Luis
Welcome to the chat, Luis!
Luis has joined the chat.
```

Fig. 2

```
PS C:\Users\egomez\Desktop\distributedPostgrade\Tarea_Chat_Basico_Privados> python .\chat_client.py
Enter server IP address: 172.17.192.1
Enter server port: 12345
Enter your name: Pedro
Welcome to the chat, Pedro!
Pedro has joined the chat.
```

Fig. 3

```
PS C:\Users\egomez\Desktop\distributedPostgrade\Tarea_Chat_Basico_Privados> python .\chat_client.py
Enter server IP address: 172.17.192.1
Enter server port: 12345
Enter your name: Ana
Welcome to the chat, Ana!
Ana has joined the chat.
```

Fig. 4

Se desarrolla una conversación donde "Luis" y "Ana" intercambian mensajes privados y también hay mensajes públicos, así como también "Pedro" intenta enviar un mensaje privado a un usuario inexistente.

Chat desde la perspectiva de "Luis", se envían mensajes privados entre él y "Ana" también se observan todos los mensajes públicos Fig 5.

```
PS C:\Users\egomez\Desktop\distributedPostgrade\Tarea_Chat_Basico_Privados> python .\chat_client.py
Enter server IP address: 172.17.192.1
Enter server port: 12345
Enter your name: Luis
Welcome to the chat, Luis!
Luis has joined the chat.

Pedro has joined the chat.

Ana has joined the chat.

\w Ana Hola Ana como estas?
pendiente a todos les voy a escribir a cada uno por privado
Luis: pendiente a todos les voy a escribir a cada uno por privado
priv(Ana): muy bien Luis recuperandome
\w Ana ok me alegra que estas mejor
priv(Ana): ni mama no te ha dicho para ir a cinecita
\w Ana si vamos este viernes
Pedro: ok gracias pendiente
Ana: creo que me desconectare pronto
[]
```

Fig. 5

Chat desde la perspectiva de "Ana", se observan los mensajes privados y públicos Fig 6.

```

PS C:\Users\egomez\Desktop\distributedPostgrade\Tarea_Chat_Basico_Privados> python .\chat_client.py
Enter server IP address: 172.17.192.1
Enter server port: 12345
Enter your name: Ana
Welcome to the chat, Ana
Ana has joined the chat.

priv(luis): Hola Ana como estas?
Luis: pendiente a todos les voy a escribir a cada uno por privado
Vw Luis muy bien Luis recuperandome
priv(luis): ok me alegra que estas mejor
Vw Luis mi mama no te ha dicho para ir a Cinecita
priv(luis): si vamos este viernes
Pedro: ok gracias pendiente
creo que me desconectare pronto
Ana: creo que me desconectare pronto

```

Fig. 6

Chat desde la perspectiva de "Pedro", no se observan los mensajes privados entre "Luis" y "Ana", se captura el error de enviar un mensaje privado a un usuario inexistente Fig 7.

```

PS C:\Users\egomez\Desktop\distributedPostgrade\Tarea_Chat_Basico_Privados> python .\chat_client.py
Enter server IP address: 172.17.192.1
Enter server port: 12345
Enter your name: Pedro
Welcome to the chat, Pedro
Pedro has joined the chat.

Ana has joined the chat.

Luis: pendiente a todos les voy a escribir a cada uno por privado
ok gracias pendiente
Pedro: ok gracias pendiente
Vw Gabriel hola... andgo
User "Gabriel" not found
Ana: creo que me desconectare pronto

```

Fig. 7

7. Link Repositorio GitHub

En el siguiente enlace puede encontrar el código completo de la implementación del chat básico con mensajes privados y sincronización de recursos en Python.

Repositorio: https://github.com/aurquiel/chatbasicopython/tree/private_message

8. Conclusión

La implementación mejorada del sistema de chat demuestra conceptos avanzados de sistemas distribuidos y programación concurrente. La adición de mensajes privados transforma el sistema de una herramienta de comunicación grupal simple a una plataforma más versátil, mientras que el uso de mecanismos de sincronización (Lock) aborda problemas fundamentales de la concurrencia.

Esta evolución ilustra cómo los sistemas distribuidos deben balancear funcionalidad con seguridad concurrente, y cómo las abstracciones de sincronización son esenciales para construir sistemas robustos y escalables. El trabajo sirve como ejemplo práctico de diseño e implementación de sistemas de comunicación en red que requieren manejo seguro de recursos compartidos en entornos concurrentes.