Assignment 1 Secure Distributed Chat System

Internet Architecture and Protocols (CS60008)

GROUP 12

February 11, 2026

1 Introduction

This project implements a secure, multi-threaded, and distributed chat application using Python. The system is designed to support horizontal scaling by leveraging **Redis** for state management and **Pub/Sub** messaging. The application includes real-time chat rooms, private messaging, a publish-subscribe notification model, and end-to-end encryption using **TLS**. The entire stack is containerized using **Docker** to simulate a multi-server environment.

2 System Architecture

The system follows a **Stateless Server Architecture**. Individual server instances (containers) do not hold global state locally. Instead, they rely on a centralized Redis instance to store user sessions, room memberships, and subscriptions. This allows clients connected to different server instances (e.g., Port 8001 and 8002) to communicate seamlessly.

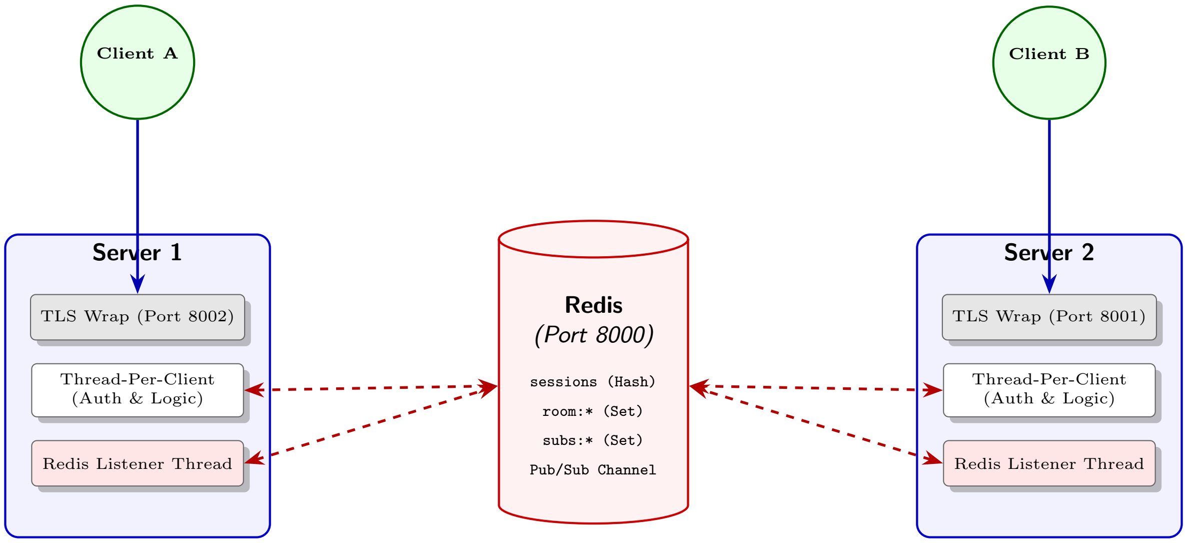

Figure 1: System Architecture Diagram showing Docker containers, TLS termination, and Redis State Management.

3 Design Choices

3.1 Concurrency Model: Thread-Per-Client

We replaced the asynchronous `asyncio` model with a **Multi-Threaded Blocking I/O** model.

- **Implementation:** A dedicated `threading.Thread` is spawned for every active client connection running the `handle_client` function.
- **Rationale:** This simplifies the control flow (Authentication → Command Loop → Disconnect) and allows for straightforward blocking socket operations ('sock.recv'). It ensures that computationally intensive tasks (like `bcrypt` hashing) for one user do not block the event loop for others.

3.2 Distributed Communication: Redis Pub/Sub

To support Problem 6 (Multiple Servers), we implemented a **Global Pub/Sub** architecture.

- **Single Channel Topology:** Instead of creating a Redis channel for every chat room (which consumes file descriptors), we use a single global channel `global_chat_events`.
- **Smart Filtering:** Each message contains a JSON payload with metadata (Room ID, Sender, Type). Server instances receive all messages but filter them locally using the `local_room2socks` map.
- **Echo Prevention:** We implemented an `exclude_sender=True` flag. When a server publishes a message, Redis broadcasts it back to the origin server. This flag ensures the origin server does not echo the message back to the sender, preventing duplicate text.

3.3 Consistency: Duplicate Login Policy

We selected **Policy 1: Reject Duplicate Login**.

- **Mechanism:** Before establishing a session, the server checks `r.hexists("sessions", username)`. If the key exists, the new connection is immediately terminated.
- **Trade-off:** This prioritizes the stability of the active session over the convenience of the new connection, preventing session hijacking.

4 Redis Schema Implementation

The Redis database is structured to support $O(1)$ lookups for critical operations.

Key Type	**Key Pattern**	**Purpose**
Hash	`sessions`	Maps `username` → `"active"`. Tracks online status.
Hash	`user:room`	Maps `username` → `room_name`. Tracks location.
Set	`room:<name>`	Stores list of usernames currently in a room.
Set	`subs:<user>`	Stores list of subscribers for a specific user.

Table 1: Redis Data Structures

5 Thread Safety Concurrency

Since the system uses multiple threads sharing local memory (sockets), thread safety is critical.

- **Local State Locks:** We utilize a global `threading.Lock()` to protect access to the shared dictionaries: `local_sock2user`, `local_user2sock`, and `local_room2socks`. This prevents race conditions where a client might disconnect (removing themselves from the dict) exactly when the Redis listener thread tries to send them a message.
- **Daemon Threads:** The Redis listener thread is started as a **Daemon Thread**. This ensures that when the main server process receives a shutdown signal (SIGINT), the listener thread terminates automatically without hanging the process.

6 TLS Security Decisions

6.1 Transport Layer Security (TLS)

- **Self-Signed Certificates:** We used the `cryptography` library to generate a 2048-bit RSA key and a self-signed X.509 certificate.
- **Server Side:** The server wraps the raw TCP socket using `ssl.wrap_socket(server_side=True)`. This forces the TLS handshake immediately upon connection.
- **Client Side:** We explicitly disable hostname checking (`check_hostname = False`) and load the local certificate into the trust store (`load_verify_locations`). This allows the client to trust our self-signed "localhost" certificate, simulating a trusted CA environment.

6.2 Authentication

Passwords are never stored in plaintext. We utilize the **bcrypt** library with salt. The `users_db` is initialized with pre-hashed passwords. During login, the incoming password is hashed and compared against the stored hash, protecting against rainbow table attacks.

7 Test Results

7.1 Scenario 1: Distributed Chat (Problem 6)

Setup: Client A connected to Port 8001 (Server 1). Client B connected to Port 8002 (Server 2).
Action: Client A typed "Hello from Server 1".
Result: Client B received the message. This confirms the Redis Pub/Sub backend successfully routed messages across isolated Docker containers.

7.2 Scenario 2: Exclusive Subscriptions (Problem 5)

Setup: Client B ran `/subscribe a`.
Action: Client A typed `/subscribe_only Secret Message`.
Result: Client B received the message prefixed with `[Sub]`. Other users in the same room

did not receive the message. This confirms the Logic-based routing (Room vs. Direct) works correctly.

7.3 Scenario 3: Duplicate Login (Problem 3)

Setup: Client A logged in as user 'a'. A second terminal attempted to login as 'a'.
Result: The second terminal received "User already logged in" and was disconnected.