Putting it all together, the main components are:

- ASCII title screen displayed on start.

- Contact management (add, list, delete).

- Encryption using Diffie-Hellman and AES.

- Peer-to-peer networking with sockets.

Now, considering threading. The listener needs to run in a separate thread so that the user can still input commands while waiting for messages. Similarly, when connected, receiving messages should be in a thread to not block sending.

Potential code outline:

1. Display ASCII title.

2. Load contacts from a JSON file.

3. Main menu with options: add contact, list contacts, start chat, quit.

4. When starting a chat, select a contact (which has IP and port), then connect to them as a client. Perform key exchange, then enter chat loop.

5. If the user chooses to listen, start a server socket, wait for connection. Once connected, perform key exchange, then chat.

For encryption steps during connection:

- When initiating a connection as a client:

- Generate DH parameters and keys.

- Send public key to server.

- Receive server's public key.

- Compute shared key.

- Derive AES key.

Similarly, when acting as a server:

- Receive client's public key.

- Send own public key.

- Compute shared key.

- Derive AES key.

///////////////////////////////////////////////////////

Main loop:

- Show ASCII title.

- Load contacts.

- Main menu: Add contact, List contacts, Chat, Quit.

- When selecting Chat, choose a contact, then decide to connect or wait.

- If connect, act as client: connect to contact's IP/port, perform key exchange, then enter chat loop.

- If wait, act as server: listen on a port, wait for connection, perform key exchange, then chat.

But how to handle both sending and receiving messages simultaneously. Use threads: one thread for sending messages, another for receiving. Or use async IO with selectors. But for simplicity, threads might be easier.

So in the chat loop:

- Start a receiver thread that listens for incoming messages, decrypts them, and displays them.

- The main thread handles user input, encrypts messages, and sends them.

Potential code structure:

When a connection is established (either as server or client):

1. Perform the ECDH key exchange.

2. Start receiver thread.

3. Enter loop to send messages.

VERSION 1.2

1. **Graceful Connection Handling**:
   * Added ping/pong mechanism to detect disconnections quickly
   * Implemented proper socket timeouts and error handling
   * Added length-prefixed message framing to handle messages reliably
   * Created proper cleanup on exit with stop events
2. **Connection Notification System**:
   * Added ConnectionHandler class to manage incoming connections
   * Implemented a persistent connection listener that runs in background
   * Added system to store pending connections with metadata
   * Created command system to accept specific connections
3. **Room for Future Features**:
   * Added structured message handling that can be extended
   * Implemented proper thread management
   * Created framework for handling different types of messages
4. **Connection Information**:
   * Added connection info exchange before accepting connections
   * Implemented metadata sharing (username, timestamp)
   * Created system to review connection details before accepting

Key improvements include:

* Reliable connection detection through ping/pong messages
* Proper thread cleanup and resource management
* Structured message handling with length prefixing
* Connection metadata exchange
* Command-based connection acceptance
* Better error handling and user feedback
* Clean shutdown mechanisms