Final Project Report: Collaborative Java Swing Piano Application

1. Abstract

This project delivers a networked piano application built in Java that enables two users to play and chat in real time, record the piano session to a file, and replay it with accurate timing. The implementation addresses four core topics from the course requirements—thread concurrency with synchronization, file I/O, networking via sockets, and GUI graphics—in distinct modules to ensure clarity, modularity, and maintainability.

2. Introduction

With the option to substitute the final exam, this project demonstrates mastery of advanced Java features. The application architecture comprises the following components:

- **Graphics**: Java Swing-based piano keyboard GUI with Metronome using graphical methods.
- **Networking (sockets)**: TCP socket communication for note events and text messages.
- **Thread concurrency**: Multi-threaded management of audio playback, recording, network I/O, metronome, and GUI events.
- **File IO**: File-based storage and retrieval of recorded sessions, real piano sample sound loading.

3. Implementation Details

3.1 Graphics

Java Swing/AWT Foundation Built the entire UI atop the standard Swing toolkit, using lightweight components (JFrame, JPanel, JButton, JSpinner, etc.) and relying on AWT for low-level drawing primitives. ##### Custom Painting via paintComponent Encapsulated all visual elements (housing, scale, pendulum) in a dedicated JPanel subclass that overrides paintComponent(Graphics), giving full control over every frame's rendering. ##### Vector-Based Drawing with Graphics2D Cast to Graphics2D to draw shapes (lines, polygons, ovals, rectangles) and text, ensuring resolution-independent, crisp graphics across display sizes. ##### Affine Transforms for Animation I also learned to use AffineTransform (translate + rotate) on the Graphics2D context to handle pendulum rotation about its pivot, rather than manually computing rotated coordinates.

3.2 Networking (sockets)

Client–Server Architecture Server: a dedicated ServerSocket listening on port 5190; accepts incoming Socket connections and spawns a handler thread for each client. Client: connects via new Socket(host, port), then wraps InputStream/OutputStream with DataInputStream/DataOutputStream for framed, UTF-8 chat and event messages. Communication: All messages are encoded in JSON in two kinds: "MUSIC" and "CHAT". The server relays each incoming event to all other clients to keep GUIs and audio playback in sync.

3.3 Thread Concurrency

Playback PlaybackManager: for each incoming noteOn event, spawns a new Thread (or submits a task to a fixed-size ExecutorService) that: 1. Opens the appropriate SourceDataLine. 2. Streams audio buffer until note-off or release. 3. Closes line. ##### Network Synchronization NetworkHandler: each socket connection uses its thread to read JSON messages and

enqueuing them on a thread-safe queue. ##### Local Synchronization Shared state (e.g. activePlaybackNotes) stored in ConcurrentHashMap and ConcurrentSkipListSet to avoid explicitly synchronized blocks. Timers and playback timestamps are tracked with AtomicLong to account for pause/resume delays safely across threads.

3.4 File IO

Loading & Saving Recordings The notes are saved in format: note,startTime,endTime, timbre, which is easy to code and modify outside. RecordingManager logs every note event (including velocity and timestamp) to a local file in JSON lines format, using BufferedWriter over FileWriter. On "Save", flushes buffer and closes stream; on "Load", reads file line by line with BufferedReader, reconstructs events, and replays them in order. ##### Piano Sample Files Piano samples (.wav) are loaded at startup and cached in memory for low-latency playback.

Soundplay Details: A thread is started for each note during the initialization process, and then the thread will listen for the signal to play the sound. For the electronic timbres, the frequency of each note was stored in advance, and the tone generator could make up certain kinds of waves(sine, square, etc.) with the corresponding frequency. Sine: Math.sin(phase) Square: Math.signum(Math.sin(phase)) Triangle: (2.0 / Math.PI) * Math.asin(Math.sin(phase)) Sawtooth:(2.0 * (phase / (2.0 * Math.PI))) - 1.0 For the real piano sound, an open-source sound pack TEDAgame's Piano Pack was used as the sound sample. The sound files will be loaded in advance and be played when the key is clicked. A metronome is also built into this App. A thread is responsible for playing beep sounds from the metronome, hence avoiding conflict with the piano keyboard. The chord is implemented by a map to indicate the pitch difference between different chords.

4. Conclusion and Future Work

This project satisfies the course requirement by integrating thread concurrency, file I/O, socket networking, and graphics in a cohesive Java application. Future enhancements could include:

- Adding JDBC support to store session metadata in a database.
- Enabling multi-room support and user authentication.
- Enhancing audio timbre options via a plugin architecture.

Report prepared by [Zhaodong Liu], submitted May 10, 2025.