SYSTEM DESIGN DOCUMENT

Night Guardian is a telemedicine system designed to enable real-time remote and secure monitoring of ECG and accelerometer (ACC) signals using BITalino biomedical device.

The platform consists of 3 independent desktop applications (Patient, Doctor, Administrator) communicating through a central **TCP sever** using a custom **JSON-based protocol** and encrypted using **public key and symmetric** encryption.

COMMUNICATION LAYER DESIGN

TCP VS UDP

The communication system behind *Night Guardian* was designed with one guiding principle: medical data must be delivered reliably, in order, and without corruption. The system uses TCP sockets rather than UDP.

The main objective of the system is to record and stream biomedical signals (ECG and ACC), allowing chronic patients with epilepsy to be monitored and doctors to track the evolution of their patients. These signals are medically relevant and, in some cases, may help identify life-threatening events. Therefore, medical data must be delivered reliably, in order, and without corruption.

This is why we chose to use **TCP/IP** rather than UDP. Because TCP provides ordered, reliable delivery, retransmission of lost packets, control error, built-in flow control and persistent connections. UDP, though faster and usually used to stream real-time data, lacks mechanisms for error control, to establish persistent connections and lost-packet handling. Features that we would have to manually implement.

With TCP, we obtain a compromise that is safe, robust and simple enough to implement and maintain.

USE OF DECORATORS

Once TCP was chosen, we decided to represent the messages using Strings as message units. Sending and receiving them through decorators such as:

- BufferReader
- PrintWriter instead of BufferWriter because it allows sending JsonObjects

This approach is simpler because:

1. Strings allow us to use human-readable protocols like JSON
2. Debugging is easier with string messages than having to manually convert between byte arrays and higher-level structures.
3. Strings remove the need to design binary framing like, for example, defining where one message ends and the next one begins. The readLine() function of the BufferedReader handles message boundaries.
4. Strings simplify JSON serialization/deserialization because Gson (the library used for the JSON messages) works directly with Strings.
5. Strings are safer for cross-platform and multi-language systems, which may be relevant for future updates.
6. The decorators ensure efficient transmission and correct encoding.

WHY JSON?

The system includes a custom communication protocol based on JSON messages. Each request includes:

- A “type” field: dictates how the message is routed. For example:
    - "LOGIN_REQUEST" → authentication module
    - "REQUEST_PATIENTS_FROM_DOCTOR" → medical records subsystem
    - "SAVE_COMMENTS_SIGNAL" → signal storage and annotation module
    - "STOP_CLIENT" → graceful shutdown of a client connection
- A “data” field: contains the metadata and the information shared between the client and server.

For example,

```
{
  "type": "REQUEST_PATIENTS_FROM_DOCTOR",
  "data": {
    "doctor_id": 1,
    "user_id": 12
```

We decided to use JSON-based protocol to provide a uniform human-readable communication language across the entire system. It provides many advantages:

- It is readable, transparent, and easy to log.
- It integrates natively with Java thanks to Gson.
- It can represent nested structures such as signals, reports, or patient profiles.
- It’s easier to debug during development
- It is extremely flexible: adding new fields or message types never breaks backward compatibility.
- JSON is widely supported and could later be consumed by web clients, mobile applications or third-party applications. Makes scalability easier.

This makes JSON ideal for our system that may grow, change, or one day integrate with mobile apps or web services.

Even large physiological recordings can be encoded as JSON by compressing them first (ZIP) and then encoding the compressed bytes in base64. This technique ensures binary safety while keeping messages compatible with text-based communication.

DESIGNING THE MULTITHREADED SERVER

On the server side, each client connection is handled by a dedicated **ClientHandler thread**. We chose this architecture because it is simple, reliable, and more than sufficient for the expected scale of the platform.

To manage the active handlers, the server uses a CopyOnWriteArrayList. This collection provides inherent **thread safety** while allowing us to iterate over the list without worrying about concurrent modifications, a detail that becomes crucial when shutting down multiple clients or broadcasting notifications.

The server also implements a shutdown protocol: when the administrator closes the server, each client receives a STOP_CLIENT message and shuts down cleanly before the server fully terminates. This avoids leaving zombie connections or half-open sockets behind.

CLIENT-SIDE THREADING AND THE USE OF A BLOCKINGQUEUE

One of the most important decisions in the client design was how to manage incoming messages. Initially, when only synchronous operations existed (such as login), reading directly from the socket worked well. But the moment we introduced the concept of the server being able to send a STOP_CLIENT request at any time, the model broke: the login method could end up competing with the message listener for the same input stream.

The solution was to introduce a dedicated **listener thread** whose only responsibility is to read incoming data from the socket. This thread never interacts with the UI directly; instead, it places all received messages into a BlockingQueue<JsonObject>.

This architectural shift solved two problems immediately:

1. The UI and background communication threads no longer interfere with one another**.** The login method, for example, simply waits for its corresponding message to appear in the queue.
2. The listener thread can handle asynchronous events, such as:
    - STOP_CLIENT
    - Connection interruptions
    - Unexpected server messages without disrupting the active UI workflow.

Blocking Queue is particularly well-suited for this because it automatically blocks the UI thread until a message arrives without consuming CPU or requiring manual synchronization.

HANDLING PHYSIOLOGICAL SIGNALS AND COMPRESSION DECISIONS

ECG and ACC readings are large, continuous streams of data. Transferring them efficiently required careful thought. We discussed two main options:

1. **Compress before sending to the server**
2. **Send raw and compress later for storage**

We opted to compress before transmission for several reasons:

- Reduces bandwidth usage, especially over hospital WiFi or home networks.
- Prevents transmission delays during peak network load.
- Makes the message payload significantly smaller.

To keep everything JSON-friendly, the compressed ZIP data is encoded in base64 and sent as a standard JSON field. This ensures compatibility with both Java's IO streams and future potential clients written in other languages.

SIGNAL-PROCESSING DESIGN DECISIONS

The proposed real-time ECG and accelerometer processing framework is designed for wearable, low-power biomedical devices, where efficiency, noise resilience, and adaptability are crucial. It uses an 8-second sliding window to ensure a continuous data stream while conserving memory, making it suitable for microcontroller-based systems. ECG samples and timestamps are stored in double-ended queues, enabling constant-time insertion and removal for real-time window management.

R-peak detection relies on a dynamic adaptive threshold instead of fixed thresholds or complex filters, addressing the high variability in ECG amplitude caused by different users, electrode placement, skin impedance, motion artifacts, and BITalino-style acquisition. This adaptive threshold increases during strong peaks to avoid false positives and decays slowly under noisy or low-amplitude conditions, offering a robust, low-cost solution.

Heart rate is calculated from recent RR-intervals, following clinical standards for accurate BPM estimation in noisy settings. The system compares the current heart rate with a baseline from earlier RR intervals, flagging increases above +15 BPM as significant. This straightforward approach provides physiologically relevant insights while fitting embedded devices. Accelerometer analysis is based on the variance of acceleration magnitude over a fixed window, which is an effective and inexpensive feature for classifying movement into calm, moderate, or abnormal states without complex frequency analysis. Thresholds are set initially and can be refined using real patient data.

Overall, the methods emphasize real-time performance, low computational requirements, portability, and robustness, making the system suitable for wearable biomedical monitoring and early event detection.

SECURITY AND ACCESS CONTROL DECISIONS

To maintain proper identity management and prevent unauthorized access, the platform does **not** allow users to self-register. Instead, it uses a token-based activation mechanism approach to safely share the token (secret key) using public key encryption for the posterior symmetric encryption of the communication over the network.

The hospital's IT department provides each user (doctor or patient) with:

- A corporate email (@nightguardian.com)
- A temporary password (temporarypass1)
- A single-use token

These credentials are only valid once, ensuring that **activation can only occur from a legitimate client device and only once per user**.

The main reason for adopting a **token-based mechanism** is that all patients and doctors are initially created by the administrator as inactive users, each with a temporary password and a single-use token stored in the database.

When the user accesses the platform for the first time, the administrator activates their accounts with the corresponding credentials (temporary password and single-use token) and the server sends its Public Key to the connected client.

During this activation step, the client securely **generates its Public/Private key pair locally**, ensuring that the private key never leaves the device. The newly **generated client's public key is then sent to the server** and if the credentials and token are valid, the administrator associates them to the user's account inside the database.

This design provides two key advantages:

- **Security-by-design**: The admin never generates or handles user private keys, preventing a major security risk.
- **Trusted key binding**: Only a user who possesses the correct token can activate the account and register a valid public key.

Once the account is activated and the client has generated its key pair, it can freely log in with its temporary password and email credentials. When logging in, the **client sends a request** to the server to get the **token** (secret key) that will encrypt all the communication over the network. This guarantees that the server verifies that the request is sent by a formal client in the network.

Once the request is received by the server, it uses the newly registered Client's Public key to **encrypt** the secret token. Additionally, the server **signs** the secret token with its own private key. This **double public-key encryption** to the secret token ensures **confidentiality**, as well as **authenticity** to the communication. The server then sends the encrypted session token back to the client. This token is used to establish a **symmetric encryption key**, which will protect all subsequent communication between client and server over the network.

This design is intentional: although public-key (asymmetric) encryption is essential for secure identity binding and initial key exchange, it is not efficient for continuous communication, especially when transferring large biomedical datasets such as EEG or accelerometer recordings. Asymmetric algorithms like RSA are computationally expensive and significantly slower.

By contrast, **symmetric encryption** (AES) provides several advantages for high-volume data transmission:

- **Much higher performance**: symmetric algorithms are orders of magnitude faster, enabling real-time or near-real-time transfer of large signal files.
- **Lower CPU usage**: ideal for mobile devices and embedded clients, reducing battery consumption and latency.
- **Scalable for high throughput data:** supports continuous streaming and recording sessions without performance degradation.
- **Strong security guarantees**: once the symmetric key is securely exchanged, all data is protected with modern, robust cryptography.

In summary, **asymmetric encryption** is used only once to securely deliver the **symmetric session key (token).** From that point on, symmetric encryption becomes the primary method, providing the speed and efficiency required to handle large medical signal files while maintaining strong end-to-end confidentiality.

The Application also supports **secure password modification** before logging in via the *"Forgot Password?"* option in the Client's login window. As we have established a secure communication between client and server, we must encrypt the modification of a password. The client requests a password modification by clicking in the *'Forgot password'* button in the *Login* window. The client inserts its email address and its new password twice to verify its modification. These credentials are **encrypted using public-key encryption**: the client encrypts this *Change Password* request with the server's public key (ensuring **confidentiality**) and with its own Private key (ensuring **authentication** and **integrity**). The server receives the encrypted message and decrypts it using its own Private key and the client's public key (previously saved in the database). By doing this, the server obtains the new client's password and with the administrator's help, properly changes it in the database.

To make sure the client acknowledges this modification, it sends back a public-key encrypted message back to the client as explained formerly; and the client decrypts it the same way the server decrypted the *Change Password* request.

After the successful password change, the client can now freely log in using the new credentials. This triggers the token exchange process, establishing symmetric encryption for future communication, just as described in the initial login process.

ERROR HANDLING AND RECOVERY

In a distributed medical system, network interruptions are inevitable. The design gracefully handles these scenarios:

- If the server intentionally disconnects a client, the client receives STOP_CLIENT, cleans up its resources, and informs the UI.
- If the connection is lost unexpectedly, the client detects the read failure, closes itself safely, and prompts the user to reconnect.
- The UI always receives a callback (onServerDisconnected()) to guide the user through recovery.

This helps avoid frozen windows, half-initialized screens, or invisible failures.

FUTURE EXTENSIONS/ IMPROVEMENTS

Future improvements may include:

- Migrating from thread-per-client to a thread pool (Executors)
- Non-blocking I/O (NIO)
- Binary compression for ECG signal transmission (reducing bandwidth)
- Integration with cloud medical systems
- Incorporating patient emergency contacts and continuous (or last known) geolocation, enabling the system to automatically notify caregivers or emergency services when a seizure is suspected, rather than only alerting the administrator as in the current version.