Remote Logging System

1st Given Horea TINEGHE
Faculty of Information Systems and Cyber Security
Military Technical Academy "Ferdinand I"
Bucuresti, Romania
horeatng@gmail.com

2nd Given Andreea PETRAR
Faculty of Information Systems and Cyber Security
Military Technical Academy "Ferdinand I"
Bucuresti, Romania
andreeapetrar9@gmail.com

***Abstract*—This paper presents the design and implementation of a Remote Logging System aimed at monitoring distributed computing devices and applications through centralized log collection. The project, titled Automatic Logger, introduces a client–server architecture in which multiple remote clients send real-time status messages to a main server for aggregation, filtering, and visualization. The system ensures fault-tolerant communication, supports log-level classification (INFO, WARN, ERROR), and maintains a searchable historical archive covering up to 48 hours of operation. The implementation demonstrates fundamental operating system concepts such as multithreading, synchronization, socket-based IPC (Inter-Process Communication), and file management, all integrated into a graphical monitoring interface resembling Wireshark. The project provides a practical foundation for networked event tracking, debugging, and system supervision within heterogeneous environments.**

***Index Terms*—Remote logging, multithreading, socket communication, client-server architecture, concurrent systems, monitoring**

I. Introduction

The Remote Logging System (RLS) represents a class of software applications that enable centralized monitoring of distributed systems by collecting, storing, and analyzing log messages generated by multiple remote devices or applications. Such systems are widely used in large-scale networks, where direct manual supervision of every node is impractical. By transmitting logs over TCP/IP, RLS architectures facilitate fault diagnosis, performance evaluation, and real-time event visualization.

In our Remote Logging System project, the server component is designed to receive and store concurrent log streams from multiple clients using a multithreaded C++ architecture. It includes an intuitive graphical user interface (GUI) built in Python, which displays incoming logs in a structured, color-coded table — similar in appearance to Wireshark's packet visualization. The interface allows filtering by log level, text-based searching across historical data, and automatic updates via timed refresh or live socket feed. Historical logs are maintained for up to 48 hours, with periodic cleanup to ensure optimal performance.

On the client side, lightweight Windows-based modules written in C++ are responsible for collecting local application events and transmitting them to the central server. Each client formats messages containing metadata such as timestamp, hostname, application identifier, log level, and descriptive content. The use of standardized message structures ensures consistency and ease of parsing at the server end, supporting future extensions for data analytics or alerting. To achieve these capabilities, several Operating System principles are applied throughout the implementation:

Multithreading: separate threads manage each client connection concurrently, improving responsiveness and scalability.

Synchronization mechanisms: mutexes protect shared resources (e.g., log files, in-memory buffers).

File I/O management: log persistence and automated cleanup ensure stable long-term operation.

Socket communication: facilitates inter-process data exchange between remote clients and the central server.

Memory and resource control: efficient allocation, cleanup, and thread management minimize overhead.

Together, these elements result in a robust and extensible Remote Logging System capable of assisting administrators in monitoring multiple systems simultaneously, identifying errors quickly, and maintaining an organized historical record of all events.

II. Related Work

A. RWELS: A Remote Web Event Logging System Author links open overlay pane

Event logs serve as a critical data source for identifying usability issues within websites. This paper introduces a web-based client-server application, the Remote Web Event Logging System (RWELS), designed to log user-interface events generated in Microsoft Internet Explorer during web interactions.

The system is engineered to capture events without interfering with the user experience, requiring no additional actions from the user to enable logging. RWELS is highly configurable and supports user-centric data collection; it allows usability analysts to specify both the set of events to be captured and the particular pages to be monitored for a given user. Once captured, event logs are dispatched via HTTP to a central server, where they are stored as text files. The system ensures that users are uniquely identified and that all logs are accurately associated with their respective user sessions.

B. Android-based remote logging and control

The system utilizes four specific sensors—moisture content, temperature, humidity, and ultrasonic—to acquire real-time data from the irrigation site for processing by a central

controller. Internet connectivity is established via a Wi-Fi device, linking the site to either a local or cloud-based server. A dedicated mobile application displays these four sensor measurements through a user-friendly graphical interface, providing real-time updates as data is received from the server.

Operationally, the system automatically manages irrigation pumps based on soil moisture threshold values. Its remote logging capability enables users to monitor farmland status—including atmospheric conditions and reservoir water levels—from any location with internet access. Furthermore, the architecture grants users the flexibility to manually override and trigger pumps remotely via their mobile devices. Experimental results indicate that the local server architecture demonstrates greater stability compared to the cloud-based alternative.

C. Compact Eddy

The architecture of a process-based model is presented, designed to simulate the daily growth and development of an olive agroecosystem. This model accounts for both olive tree and grass cover dynamics, specifically focusing on their competition for water resources. Central to the system is a phenological sub-model that simulates the sequence of vegetative and reproductive stages to determine biomass allocation, with final yield calculated as a fraction of total accumulated biomass at the end of the growing season.To ensure broad applicability, the model underwent calibration and validation across diverse climates, soil types, planting densities, and management practices. Initial calibration was conducted using daily CO_2 flux measurements from a three-year eddy covariance experiment. Subsequently, the model's effectiveness was tested across multiple sites in the Tuscany region, focusing on key plant and grass cover processes.

III. System Architecture and Design

A. Overview

The system follows a classic client-server architecture where multiple workstations (clients) report information about active processes to a centralized server. The architecture consists of three main components: the client application running on each monitored workstation, the network communication layer ensuring reliable data transmission, and the server application collecting and storing received information.

B. Architectural Pattern

The project adopts a modular architecture based on Separation of Concerns. Each module is responsible for a specific aspect of system functionality and exposes a clear interface to other modules.

At the client level, the architecture consists of four main modules: Process Collector interacts with the operating system through Windows APIs to enumerate active processes and extract information. Network Client manages TCP connection to the server and implements message framing protocol. Log Types defines data structures and implements their conversion to JSON format. Client Config centralizes all configurable system parameters.

C. Client Architecture

The client is structured around a main loop that orchestrates interactions between modules. At startup, the client initializes Winsock library and establishes server connection. After authentication handshake, it enters the main monitoring loop.

Each loop iteration consists of several steps: collecting information about active processes, filtering to keep only applications with graphical interface, serializing each process to JSON format, validating message sizes, individual transmission to server with delay between messages, and cleaning CPU statistics cache for terminated processes.

Process Collector is the most complex module, implementing operating system query logic. It creates a snapshot of all processes using CreateToolhelp32Snapshot, enumerates each process with Process32NextW, and determines if each has visible windows using EnumWindows. For processes passing filtering, the module opens a handle with OpenProcess and collects detailed statistics about memory through GetProcessMemoryInfo, execution times through GetProcessTimes, and user information through OpenProcessToken.

CPU usage calculation requires maintaining a cache with previous measurements for each process. At each measurement, the module compares current kernel and user times with previous values to determine how much CPU the process consumed in the elapsed interval.

D. Communication Protocols

TCP offers a continuous byte stream without message delimiters. To transmit discrete messages, we implement a length-prefix framing protocol. Each message consists of two parts: a four-byte header containing the payload length as an unsigned 32-bit integer in big-endian network byte order, followed by the actual payload of specified length.

The sender converts length from host byte order to network byte order using htonl function before transmission. The receiver first reads the four header bytes, converts length from network byte order to host byte order with ntohl, then reads exactly that many bytes for the payload.

Each message payload is a JSON object encoded in UTF-8. All messages contain a type field identifying message type. HELLO message sent by client at connection contains type with value HELLO, clientname with workstation hostname, and version with protocol version. WELCOME message sent by server contains type WELCOME, servername, and timestamp.

Network communication is inherently unreliable and must handle multiple error scenarios. If send function returns error or sends fewer bytes than requested, connection is considered broken. Similarly, if recv returns zero bytes, connection was closed by remote party. For fatal errors, client marks connection as closed and initiates reconnection.

Reconnection uses a simple strategy with fixed two-second delay between attempts. After reconnection, handshake is re-executed to reinitiate session with server. Configured socket timeouts prevent indefinite blocking.

The server must accept simultaneous connections from multiple clients, requiring a multi-threaded or asynchronous model.

In the thread-per-client model, the main thread executes a loop that accepts new connections. For each accepted client, a dedicated thread is created that handles all interactions with that client until disconnection. This model provides simplicity and isolation but has scalability limitations.

The server maintains data structures to track connected clients and accumulated logs. The ClientConnection structure represents an active client with fields for socket descriptor, client hostname, IP address, connection timestamp, last activity timestamp, and message count. The LogEntry structure represents a single process measurement with fields corresponding to ProcessInfo transmitted by clients.

IV. Implementation

Both the client and the server are implemented using low-level system programming. The client uses C++ (C++11 standard) with Windows API using Visual Studio 2022 as development environment, while the server is implemented in pure C (C99 standard) using POSIX APIs for Linux/Unix systems. This cross-platform approach demonstrates interoperability between different operating systems through standardized network protocols.

Key libraries used for the client include windows.h for basic Windows functions, winsock2.h and ws2tcpip.h for network communication, psapi.h for Process Status API, and tlhelp32.h for Tool Help Library. The project links against ws2_32.lib for Winsock, psapi.lib for process information, and user32.lib for window enumeration.

The server is developed for POSIX-compliant systems (Ubuntu 24+) using GCC compiler. Key libraries include sys/socket.h and netinet/in.h for BSD socket operations, pthread.h for POSIX threads, termios.h for terminal control, and dirent.h for directory operations. The server links against the pthread library for multi-threading support.

A. Project Structure

1) Client Structure (Windows): The client is structured around a main loop that orchestrates interactions between modules. At startup, the client initializes Winsock library and establishes server connection. After authentication handshake, it enters the main monitoring loop.

The client consists of four main modules: Process Collector interacts with the operating system through Windows APIs to enumerate active processes and extract information. Network Client manages TCP connection to the server and implements message framing protocol. Log Types defines data structures and implements their conversion to JSON format. Client Config centralizes all configurable system parameters.

2) Server Structure (Linux/POSIX): The server implementation follows a modular architecture with clear separation of concerns. The project consists of the following source files:

main.c serves as the application entry point, containing the main function, global variable definitions, main menu display logic, and server lifecycle management. It initializes all shared data structures and coordinates thread creation and cleanup.

retea.c (network module) implements the TCP server using BSD sockets API. It contains three thread functions: `thread_server` for accepting incoming connections, `thread_gestionare_client` for handling individual client communication, and `thread_refresh_automat` for periodic display updates.

parser_json.c provides a lightweight custom JSON parser specifically designed for the process log format. It implements extraction functions for strings, integers, longs, and doubles, as well as higher-level functions for parsing individual process entries and process snapshot arrays.

afisare.c (display module) handles all terminal rendering using ANSI escape codes. It implements screen clearing, color-coded log display, filter evaluation, header rendering, and the command menu interface.

export.c provides CSV export functionality, generating timestamped files with all log entries that pass current filter criteria.

terminal.c manages terminal mode switching between canonical (line-buffered) and raw (character-by-character) modes using the termios API, enabling non-blocking keyboard input during server operation.

utilitare.c (utilities module) contains helper functions for string manipulation, time formatting, case conversion, and case-insensitive text searching.

vizualizare_loguri.c implements the historical log viewer, allowing users to load and analyze previously exported CSV files without running the server.

Header files define data structures (`structuri_date.h`), color codes and configuration constants (`culori_si_configurari.h`), and function prototypes for each module.

B. Key Implementation Details

1) Client-Side Process Collection: Process Collector is the most complex client module, implementing operating system query logic. It creates a snapshot of all processes using CreateToolhelp32Snapshot, enumerates each process with Process32NextW, and determines if each has visible windows using EnumWindows. For processes passing filtering, the module opens a handle with OpenProcess and collects detailed statistics about memory through GetProcessMemoryInfo, execution times through GetProcessTimes, and user information through OpenProcessToken.

CPU usage calculation requires maintaining a cache with previous measurements for each process. At each measurement, the module compares current kernel and user times with previous values to determine how much CPU the process consumed in the elapsed interval.

2) Server-Side Multi-threaded Architecture: The server employs a thread-per-client model implemented with POSIX threads. The main server thread executes an accept loop with a one-second timeout configured via `SO_RCVTIMEO` socket option, allowing periodic checks of the shutdown

flag. For each accepted connection, a new thread is spawned using `pthread_create` and immediately detached with `pthread_detach` to enable independent execution and automatic resource cleanup upon termination.

```
pthread_t thread_id;
pthread_create(&thread_id, NULL,
    thread_gestionare_client, info);
pthread_detach(thread_id);
```

3) Thread Synchronization: Two POSIX mutexes protect shared data structures from concurrent access. The `g_mutex_loguri` mutex guards the log entry array and count, acquired during log insertion, display rendering, clearing, and CSV export operations. The `g_mutex_clienti` mutex protects the connected clients list, acquired when adding clients on connection, removing them on disconnection, and displaying the client list.

The server shutdown flag `g_server_ruleaza` is declared as `volatile sig_atomic_t` to ensure atomic access from signal handlers without requiring mutex protection, enabling safe communication between the SIGINT/SIGTERM handler and all running threads.

4) JSON Message Processing: The custom JSON parser handles incoming data by searching for key patterns and extracting values. For string extraction, it locates the key surrounded by quotes, skips the colon and whitespace, then copies characters until the closing quote while handling escape sequences. Numeric values are extracted using standard library functions (`atoi`, `atol`, `atof`).

Process snapshot arrays are parsed by tracking brace depth to identify complete JSON objects within the array:

```
int depth = 0;
while (*pos) {
    if (*pos == '{') depth++;
    else if (*pos == '}') {
        depth--;
        if (depth == 0) {
            // Complete object found
            parse_process(start, pos);
        }
    }
    pos++;
}
```

5) Terminal User Interface: The display module uses ANSI escape sequences for a colorful terminal interface. Screen clearing is performed with `\033[2J\033[H`, colors are applied using sequences like `\033[32m` for green foreground and `\033[41m` for red background, and formatting is reset with `\033[0m`.

The terminal is switched to raw mode for non-blocking input using termios configuration:

```
struct termios settings;
tcgetattr(STDIN_FILENO, &settings);
settings.c_lflag &= ~(ICANON | ECHO);
settings.c_cc[VMIN] = 0;
settings.c_cc[VTIME] = 1;
tcsetattr(STDIN_FILENO, TCSANOW, &settings);
```

This allows the main loop to poll for keyboard input without blocking, enabling simultaneous display updates and user interaction.

6) Log Filtering System: The filtering system evaluates each log entry against three criteria: log level (INFO, WARN, ERROR), process status (RUNNING, SLEEPING, STOPPED, ZOMBIE, CRASHED), and free-text search. Comparisons are performed case-insensitively by converting both operands to uppercase before comparison. Text search scans multiple fields including process name, username, message content, status, and hostname using a helper function that creates lowercase copies of both strings before calling `strstr`.

7) Buffer Management: The log buffer is implemented as a fixed-size circular array with capacity of 1000 entries (configurable via `MAX_LOGURI`). When the buffer reaches capacity, the oldest entry is removed using `memmove` to shift all entries, and the new entry is inserted at the end, maintaining FIFO ordering. This operation occurs atomically within the mutex lock.

C. Configuration Parameters

Client configuration parameters are defined in client_config.hpp: CLIENT_BUFFER_SIZE set to 16384 bytes for maximum send/receive buffer size, MAX_FIELD_LENGTH set to 256 characters for text field limits, MAX_JSON_MESSAGE_SIZE set to 15360 bytes for JSON payload limit with safety margin, DELAY_BETWEEN_SENDS_MS set to 50 milliseconds for throttling between consecutive transmissions, CONNECTION_TIMEOUT_MS set to 5000 milliseconds for connection establishment, SEND_TIMEOUT_MS set to 3000 milliseconds for send operations, and RECEIVE_TIMEOUT_MS set to 5000 milliseconds for receive operations.

Server configuration parameters are defined in culori_si_configurari.h: SERVER_PORT set to 12345 for TCP listening port, MAX_CLIENTI set to 50 for maximum concurrent client connections, MAX_LOGURI set to 1000 for maximum stored log entries, DIMENSIUNE_BUFFER set to 8192 bytes for network receive buffer size, and LUNGIME_CAMP set to 64 characters for default string field length.

These parameters were selected based on empirical testing and represent a balance between performance, reliability, and resource consumption.

D. Error Handling and Robustness

The implementation includes comprehensive error handling at multiple levels on both client and server sides.

On the client side, socket operations check return values and handle errors appropriately. When send or recv fails, the connection is marked as broken and reconnection is initiated. The reconnection algorithm includes delay to allow transient

issues to resolve, closes existing socket, attempts new connection, and resends handshake after successful reconnection. Memory management uses RAII principles where resources are tied to object lifetime. The CPU cache is periodically cleaned to remove entries for terminated processes, preventing unbounded memory growth.

On the server side, socket operations use `perror` for diagnostic output. When `send` or `recv` fails, the connection is marked as broken and the client handler thread exits gracefully after removing the client from the connected list. The SIGPIPE signal is explicitly ignored using `signal(SIGPIPE, SIG_IGN)` to prevent server crashes when writing to disconnected clients. Accept timeouts (`EAGAIN`/`EWOULDBLOCK`) are treated as normal operation, allowing the accept loop to check the shutdown flag periodically.

Memory management on the server follows careful allocation and deallocation patterns. Client information structures are allocated with `malloc` and freed after data extraction in the handler thread. Client IP strings created with `strdup` are freed during disconnect cleanup. The `contine_text_insensitiv` function allocates temporary string copies for case conversion and properly frees them after comparison to prevent memory leaks.

Graceful shutdown is coordinated through the `g_server_ruleaza` flag. When SIGINT or SIGTERM is received, the signal handler sets this flag to zero, causing all thread loops to exit. The main thread then waits for the server and refresh threads using `pthread_join`, closes the server socket, frees all allocated client strings, restores terminal settings, and exits cleanly.

V. Testing and validation

The testing approach combines functional testing to verify correctness, performance testing to evaluate efficiency, and robustness testing to ensure reliable operation under adverse conditions.

Objective: Verify that all applications with visible windows are correctly detected.

Procedure: Start several known applications (Chrome, Notepad, Calculator), run client, examine console output to confirm all started applications appear in process list.

Results: Client successfully detects all applications with visible windows. Background processes and system services are correctly excluded. Process information is accurate based on Task Manager comparison.

All functional tests passed, confirming that the implementation correctly identifies processes, calculates CPU usage, and transmits data. Performance tests demonstrate minimal overhead suitable for continuous operation. Robustness tests confirm reliable operation under network failures and error conditions.

The testing validates that the system meets its objectives of providing accurate process monitoring with minimal performance impact and robust error recovery.

VI. Conclusions and validations

This project successfully implemented a distributed process monitoring system demonstrating key operating systems concepts. The client application leverages Windows APIs to enumerate processes, extract detailed metrics including CPU usage and memory consumption, and filter applications from background processes. The network communication layer implements a reliable protocol using TCP/IP sockets with JSON encoding. The system demonstrates robustness through automatic reconnection when connectivity is lost.

From an academic perspective, the project illustrates Process Control Block access through system APIs, CPU scheduling and time accounting mechanisms, memory management concepts, and inter-process communication through network sockets. These concepts are demonstrated through working code that interacts with real operating system facilities.

From a practical perspective, the system provides a functional monitoring solution suitable for small to medium-sized environments. The lightweight architecture with minimal dependencies facilitates deployment, while the modular design allows future extensions.

A. Encountered Challenges

Several challenges were encountered during implementation. Correctly calculating CPU usage required understanding the relationship between kernel time, user time, and real time, as well as maintaining state between measurements. Handling partial sends and receives on sockets required careful attention to ensure complete data transmission. Filtering applications from background processes involved experimenting with various window properties to match Task Manager behavior. Managing the CPU cache required implementing cleanup logic to prevent memory leaks.

Protocol design required balancing simplicity with robustness. The length-prefix framing provides reliable message delimitation while remaining simple to implement. JSON serialization offers readability but requires careful character escaping to avoid parsing errors.

B. Limitations

The current implementation has limitations that constrain applicability. The client is specific to Windows, as it uses Windows-specific APIs, and the server is Linux-specific. Porting the client to Linux or macOS would require substantial rewrites using different system calls. The protocol lacks authentication and encryption, making it unsuitable for untrusted networks. The system does not handle network address changes gracefully. Server implementation is not included, requiring users to develop their own or use estimated architecture.

Performance limitations include that the client scans all processes each cycle even though most remain unchanged. An optimization would track process creation/termination events to update only changed processes. The CPU cache grows proportionally with monitored processes, though cleanup bounds this growth.

C. Future Enhancements

Several enhancements could improve capabilities and address limitations:

Cross-platform support would allow monitoring macOS and other OS workstations. This requires abstracting process enumeration behind platform-independent interface and implementing platform-specific backends.

Security enhancements including TLS encryption would make the system suitable for untrusted networks. Client authentication through certificates would prevent unauthorized connections.

Server implementation with database integration would provide long-term log retention and efficient querying. Web dashboard for visualization would facilitate trend identification and problem diagnosis.

Advanced features could include anomaly detection using machine learning to identify unusual process behavior, alerting mechanisms when thresholds are exceeded, and process correlation across time to identify patterns.

Performance optimizations could include differential updates sending only changed processes, hierarchical aggregation for scaling to thousands of nodes, and compression for reduced bandwidth usage.

D. Educational Value

This project provided valuable hands-on experience with operating systems programming. Working with process enumeration APIs illustrated how operating systems manage processes and expose information. Implementing CPU calculations demonstrated relationships between scheduling and time accounting. Socket programming illustrated inter-process communication mechanisms. Handling connection failures demonstrated distributed systems challenges.

The project reinforced software engineering practices including modular design, comprehensive error handling, and systematic testing. The experience is applicable to other systems programming tasks including performance optimization, system monitoring, security tools, and many other applications.

REFERENCES

[1] Microsoft Corporation, "Windows Sockets 2," Microsoft Docs, 2023. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/winsock/

[2] Rafael Accorsi , "On the Relationship of Privacy and Secure Remote Logging in Dynamic Systems", pp 329–339.

[3] M. Hauswirth et al., "Distributed Process Monitoring and Profiling," in Proceedings of the IEEE International Conference on Cluster Computing, 2004. controlled eddy covariance logging system".

[4] Olugbenga Kayode Ogidan, Kennedy Richmond Afia, "Smart irrigation system with an Android-based remote logging and control".

[5] Steven Robbins, "Remote logging in Java using Jeli: a facility to enhance development of accessible educational software".

[6] Ting-Fang Cheng, Jung-San Lee, Chin-Chen Chang , "Security enhancement of an IC-card-based remote login mechanism"