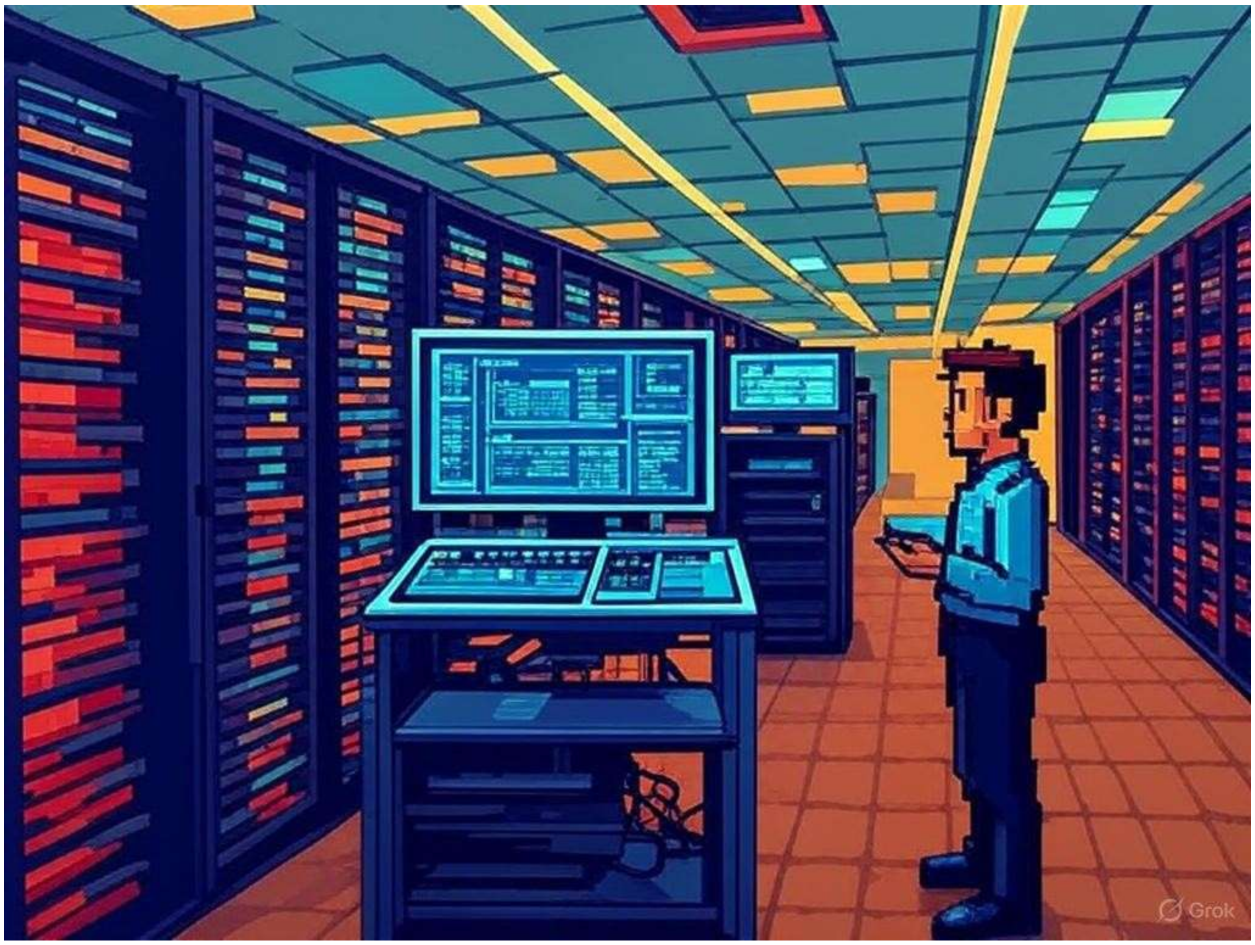
Distributed Logging System Report



Ahmed Essam Abdallah 192100088

Yousef Mahmoud Anis 192100029

Yousef Mohamed Gabr 192100069

Mohamed Hazem Saleh 192100040

Dr/ Cherry Aly

Executive Summary

This report presents an in-depth analysis of a developed Distributed Logging System, designed to tackle the challenge of aggregating log data from disparate software components operating across a network. Built upon fundamental distributed computing principles, the system features robust client-side log generation (including specialized Windows Event Log collection with a local fallback), a centralized Flask server for data ingestion and state management, and a web interface providing real-time visualization and interactive log handling. The system effectively demonstrates concepts such as concurrency, network communication, independent failure handling, and centralized state management in a distributed context. This document provides a detailed breakdown of each component's design, functionality, implementation details, and its role within the broader distributed architecture, validated by direct proof-of-concept code excerpts from the source files
(client.py, client_eventlog.py, server.py, view_logs.html, tickets.html, app.log, log_state.jso n).

System Overview

The Distributed Logging System centralizes logs from distributed clients, providing a unified platform for monitoring, analysis, and management. It supports general application logs, Windows Event Logs, and Linux system logs, with features like real-time streaming, log categorization (pinning, ticketing), and statistical visualization. Built using Python (Flask, Requests), HTML, JavaScript, and Chart.js, the system is designed for extensibility and ease of use.

- **Centralized Logging**: Aggregate logs from multiple clients for unified monitoring.
- **Real-Time Monitoring**: Provide live updates via Server-Sent Events (SSE).
- **Fault Tolerance**: Ensure reliability through retries, fallback mechanisms, and error handling.
- **User-Friendly Interface**: Offer intuitive web interfaces for log visualization and management.
- **Scalability**: Support multiple clients and concurrent log processing.
- **Cross-Platform Support**: Handle logs from Windows (via client_eventlog.py) and Linux (via client_linux.py) environments.

Core Components

1. **Client-Side Components**:

   - client.py: Simulates application logging, sending INFO, WARNING, and ERROR logs to the server.
   - client_eventlog.py: Monitors Windows Event Logs (Application, System) and forwards events to the server.
   - client_linux.py: Monitors Linux system logs (e.g., /var/log/syslog) and forwards them to the server.
2. **Server-Side Component**:
   - server.py: A Flask server that receives, stores, and serves logs, with REST APIs and SSE for real-time updates.
3. **Web Interfaces**:
   - view_logs.html: Displays all logs with real-time updates and a log level distribution chart.
   - tickets.html: Shows ticketed logs for focused issue tracking.

System Architecture

The system follows a client-server architecture with distributed clients communicating with a centralized server over HTTP. The architecture is depicted as follows:

- **Clients**: Multiple instances of client.py, client_eventlog.py, and client_linux.py run on different machines, each with a unique CLIENT_ID. They send logs to the server via HTTP POST requests to the /log endpoint.
- **Server**: The Flask server listens on port 5000, appends logs to app.log with a UUID, and maintains state in log_state.json for actions like pinning, ticketing, and deletion.
- **Web Interface**: Users access logs at /view or /tickets, with real-time updates via SSE at /stream-logs.
- **Storage**: Logs are stored in app.log, and state (pinned, ticketed, deleted) is stored in log_state.json. Locks ensure thread-safe file access.

The system uses file-based storage for simplicity but is extensible to databases for scalability.

Concept of Distributed Computing

Distributed computing involves multiple nodes collaborating across a network to achieve a common goal. This system embodies distributed computing principles in the following ways:

1. **Decentralized Log Generation**:
   - Clients (client.py, client_eventlog.py, client_linux.py) operate independently on different machines, generating logs based on local events. This allows horizontal scaling by adding more clients.
2. **Centralized Aggregation**:
   - The server consolidates logs from distributed clients, enabling unified analysis.
3. **Asynchronous Communication**:
   - Clients send logs asynchronously via HTTP, decoupling client and server operations for fault tolerance.
4. **Scalability**:
   - The server handles concurrent requests using Flask's threaded mode and a queue for SSE.
5. **Fault Tolerance**:
   - Clients retry failed requests and use local fallback storage (e.g., client_eventlog.py). The server handles errors gracefully.
6. **Real-Time Propagation**: o SSE pushes logs to web clients instantly, supporting real-

   time monitoring.
7. **State Management**:
   - The server maintains log state independently, allowing distributed clients to interact with logs without direct file access.
8. **Communication via Network (Message Passing):** The primary interaction between clients and the server is through message passing over a network (HTTP POST requests). The web interface also communicates with the server via network requests (HTTP GET, POST, SSE).
9. **Client-Side State Management for Data Source Tracking:** client_linux.py maintains local state (last_known_inode, last_known_size) to track its position within the monitored file and detect changes like rotation or truncation. This is a form of client-side state management necessary for reliably processing data from a continuously updated local source in a distributed context, distinct from the server's central state management of log metadata.

Detailed Project Explanation

Client Implementation

client.py

Purpose: Simulates an application generating logs and sending them to the server.

Functionality:

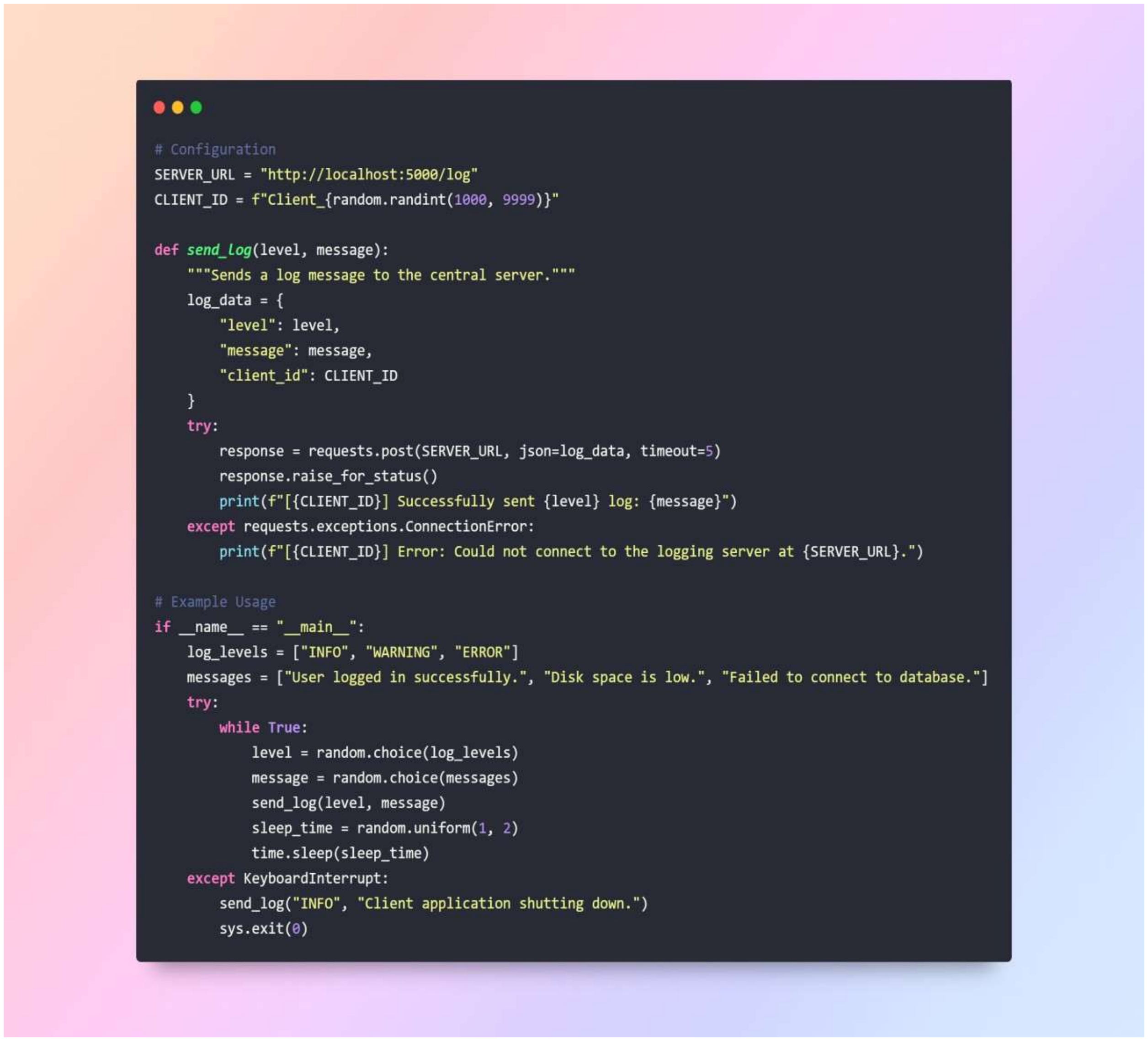
- Uses the requests library for making outbound HTTP POST requests, simplifying network communication.
- Employs random and time to simulate asynchronous, varied logging behavior.

Detailed Workflow/Logic:

1. Initializes a unique CLIENT_ID using a random integer (simple but effective for this PoC).
2. Defines a list of possible log levels and messages.
3. The send_log function formats the log data (level, message, client_id) into a Python dictionary.
4. This dictionary is sent as JSON (json=log_data) via an HTTP POST request to the configured SERVER_URL/log.
5. A timeout is set for the request to prevent indefinite hanging.
6. response.raise_for_status() checks if the server returned a successful status code (2xx).
7. try...except blocks catch common network errors (ConnectionError, Timeout, RequestException) and print informative messages instead of crashing.
8. The main execution block (if __name__ == "__main__":) enters an infinite loop.
9. Inside the loop, it randomly selects a level and message.
10. It calls send_log to send the generated log.

11. It then pauses for a random duration using time.sleep() before the next iteration, simulating variable load or activity.
12. A KeyboardInterrupt handler allows the user to stop the client gracefully, sending a final "shutting down" log message.

PoC Code:



```python
# Configuration
SERVER_URL = "http://localhost:5000/log"
CLIENT_ID = f"Client_{random.randint(1000, 9999)}"

def send_log(level, message):
    """Sends a log message to the central server."""
    log_data = {
        "level": level,
        "message": message,
        "client_id": CLIENT_ID
    }
    try:
        response = requests.post(SERVER_URL, json=log_data, timeout=5)
        response.raise_for_status()
        print(f"[{CLIENT_ID}] Successfully sent {level} log: {message}")
    except requests.exceptions.ConnectionError:
        print(f"[{CLIENT_ID}] Error: Could not connect to the logging server at {SERVER_URL}.")

# Example Usage
if __name__ == "__main__":
    log_levels = ["INFO", "WARNING", "ERROR"]
    messages = ["User logged in successfully.", "Disk space is low.", "Failed to connect to database."]
    try:
        while True:
            level = random.choice(log_levels)
            message = random.choice(messages)
            send_log(level, message)
            sleep_time = random.uniform(1, 2)
            time.sleep(sleep_time)
    except KeyboardInterrupt:
        send_log("INFO", "Client application shutting down.")
        sys.exit(0)
```

Analysis:

1. The send_log function encapsulates HTTP communication, ensuring reusability.
2. Error handling maintains client operation during network issues.
3. Random intervals simulate real-world logging patterns. **client_eventlog.py**

Purpose: Monitors Windows Event Logs (Application, System) and forwards events to the server.

Functionality:

1. Uses win32evtlog to read Windows Event Logs, requiring pywin32.
2. Maps Windows event types to log levels (e.g., EVENTLOG_ERROR_TYPE to ERROR).
3. Supports configurable logs to monitor and levels to forward.
4. Implements retry logic (3 attempts) and local fallback (logs/client_eventlog_fallback.log).
5. Tracks the last processed record number per log to avoid duplicates.
6. Polls logs every second, processing up to 1000 events per cycle.

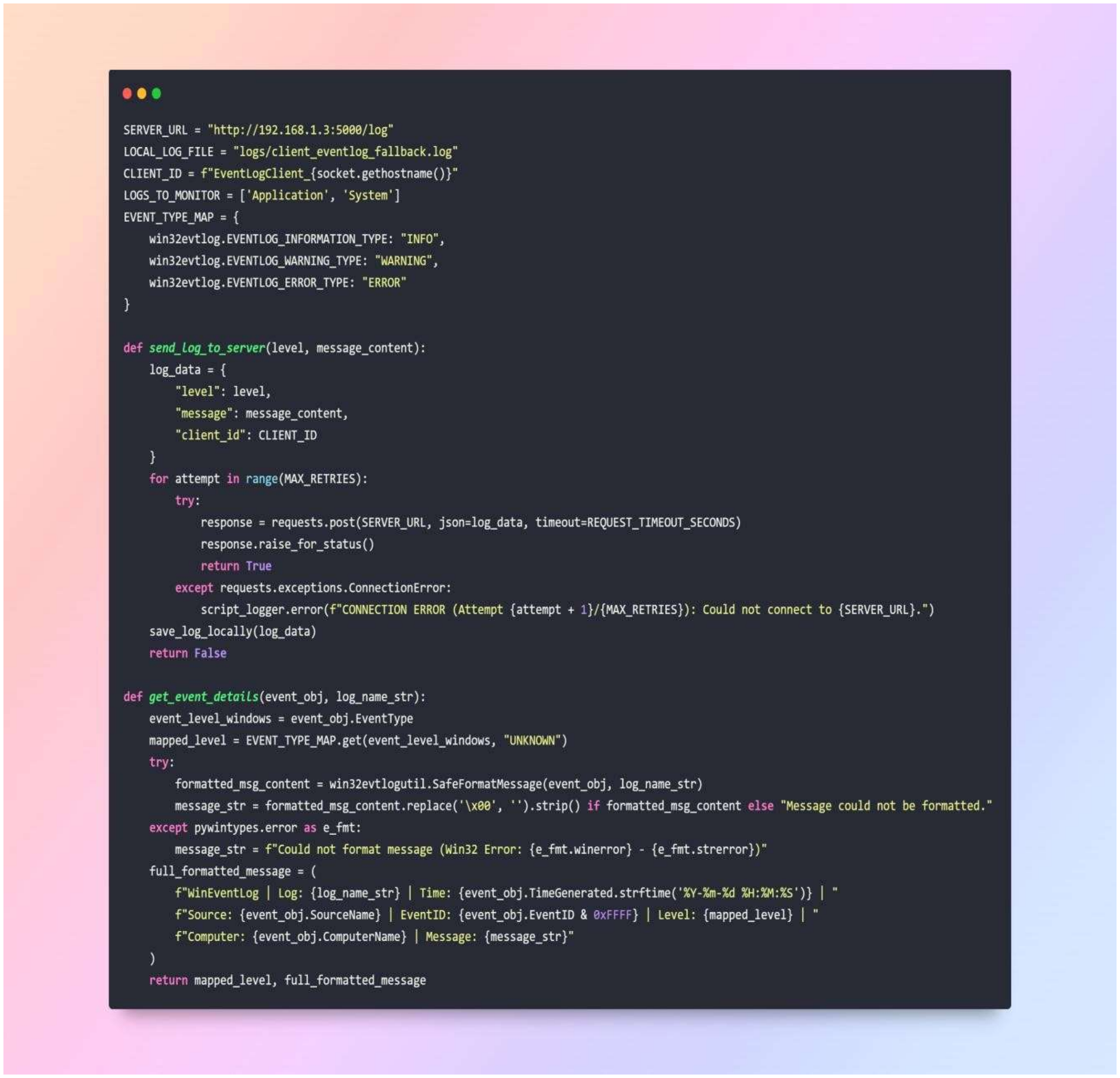
Detailed Workflow/Logic:

1. Checks for the pywin32 dependency and exits if not found.
2. Configures Python's internal logging for the script's operation messages.
3. Initializes a CLIENT_ID based on the machine's hostname (socket.gethostname()).
4. Defines configuration parameters, including which Windows Logs to monitor (Application, System), how to map Windows event types to custom levels, and which custom levels to actually forward.
5. Initializes the last_record_numbers dictionary by querying the *oldest* available record number in each monitored log. This is a simple starting point; a more robust system might read the *latest* on startup or store the last read ID persistently.
6. The send_log_to_server function attempts to send the log data via HTTP POST, similar to client.py, but wraps it in a retry loop (MAX_RETRIES). Specific error messages and troubleshooting tips are included for connection errors. If all retries fail, it calls save_log_locally.
7. save_log_locally appends the log data (as a JSON string) to a configured local file (LOCAL_LOG_FILE). This acts as a buffer during server outages.
8. The get_event_details function reads properties from a win32evtlog event object, attempts to format the event message using win32evtlogutil.SafeFormatMessage, and constructs a standardized message string including log source, timestamp, ID, level, computer, and the formatted message. It handles potential errors during message formatting.
9. read_events_from_log opens a handle to a specific Windows Event Log. It reads events in chunks (win32evtlog.ReadEventLog with a read count) starting from

the start_rec_num marker, reading *forwards*. It handles the case where the log might have wrapped or been cleared (oldest record number increases). It tracks the highest record number read in the current batch and limits the total events processed per polling cycle.

10. The main loop (if __name__ == "__main__":) iterates through the configured LOGS_TO_MONITOR.
11. For each log, it calls read_events_from_log using the last_record_numbers entry for that log.
12. It iterates through the events returned, extracts details using get_event_details, checks if the mapped level should be forwarded, and if so, calls send_log_to_server. A small SEND_DELAY_SECONDS can be introduced between sends to avoid overwhelming the server or network if a large batch of events is read.
13. After processing all events from a specific log source in a cycle, it updates the last_record_numbers for that log with the highest record number seen, ensuring the next cycle starts from where it left off.
14. The loop pauses for POLL_INTERVAL_SECONDS before checking the logs again.
15. KeyboardInterrupt and a general Exception handler in the main loop provide graceful shutdown and catch unexpected errors, logging them critically.

PoC Code:



```python
SERVER_URL = "http://192.168.1.3:5000/log"
LOCAL_LOG_FILE = "logs/client_eventlog_fallback.log"
CLIENT_ID = f"EventLogClient_{socket.gethostname()}"
LOGS_TO_MONITOR = ['Application', 'System']
EVENT_TYPE_MAP = {
    win32evtlog.EVENTLOG_INFORMATION_TYPE: "INFO",
    win32evtlog.EVENTLOG_WARNING_TYPE: "WARNING",
    win32evtlog.EVENTLOG_ERROR_TYPE: "ERROR"
}

def send_log_to_server(level, message_content):
    log_data = {
        "level": level,
        "message": message_content,
        "client_id": CLIENT_ID
    }
    for attempt in range(MAX_RETRIES):
        try:
            response = requests.post(SERVER_URL, json=log_data, timeout=REQUEST_TIMEOUT_SECONDS)
            response.raise_for_status()
            return True
        except requests.exceptions.ConnectionError:
            script_logger.error(f"CONNECTION ERROR (Attempt {attempt + 1}/{MAX_RETRIES}): Could not connect to {SERVER_URL}.")
    save_log_locally(log_data)
    return False

def get_event_details(event_obj, log_name_str):
    event_level_windows = event_obj.EventType
    mapped_level = EVENT_TYPE_MAP.get(event_level_windows, "UNKNOWN")
    try:
        formatted_msg_content = win32evtlogutil.SafeFormatMessage(event_obj, log_name_str)
        message_str = formatted_msg_content.replace('\x00', '').strip() if formatted_msg_content else "Message could not be formatted."
    except pywintypes.error as e_fmt:
        message_str = f"Could not format message (Win32 Error: {e_fmt.winerror} - {e_fmt.strerror})"
    full_formatted_message = (
        f"WinEventLog | Log: {log_name_str} | Time: {event_obj.TimeGenerated.strftime('%Y-%m-%d %H:%M:%S')} | "
        f"Source: {event_obj.SourceName} | EventID: {event_obj.EventID & 0xFFFF} | Level: {mapped_level} | "
        f"Computer: {event_obj.ComputerName} | Message: {message_str}"
    )
    return mapped_level, full_formatted_message
```

Analysis:

1) Robust error handling addresses Windows API issues (e.g., access denied).
2) Local fallback ensures no data loss during server outages.
3) Detailed log formatting aids debugging. **client_linux.py**

Purpose: Monitors Linux system logs (e.g., /var/log/syslog or /var/log/messages) and forwards them to the server.

Functionality:

- Generates a unique CLIENT_ID based on the hostname (LinuxClient_<hostname>).
- Sends logs to http://192.168.1.3:5000/log with a 10-second timeout.
- Infers log levels (INFO, WARNING, ERROR) from keywords (e.g., “error” maps to ERROR).
- Polls the log file every 5 seconds, reading new lines since the last check.
- Handles log file rotation by tracking inode numbers and file size.
- Includes extensive error handling for file access, permissions, and network issues.
- Provides troubleshooting tips for connection failures (e.g., firewall, server status).

Key Technical Details:

- Uses standard Python libraries (os, stat) for file system interaction, specifically to check file metadata (inode and size).
- Employs basic string searching (keyword in lower_line) for inferring log levels based on common terms in the log line.
- Uses the requests library for sending logs via HTTP POST, similar to client.py.
- Implements a simple file-tailing logic based on tracking file size and inode number.

Detailed Workflow/Logic:

a. Initializes a CLIENT_ID based on the machine's hostname.
b. Defines configuration parameters, including the specific LOG_FILE_TO_MONITOR, the POLL_INTERVAL_SECONDS, and a simple LOG_LEVEL_MAP for keyword-based level inference.
c. Performs initial checks to ensure the configured log file exists and is readable, exiting with a fatal error if not.
d. Gets the initial file status (inode and size) of the monitored log file using os.stat and stores them in last_known_inode and last_known_size.
e. The main execution block (if __name__ == "__main__":) enters an infinite loop.
f. Inside the loop, it periodically checks the current status of the log file (current_inode, current_size) using check_file_status.
g. It checks for file disappearance or inability to stat, waiting and retrying if necessary.
h. It detects **log rotation or truncation** by comparing the current_inode and current_size to the last_known_inode and last_known_size:

i. If the current_inode is different from last_known_inode, or if the current_size is less than last_known_size, it assumes the file has been replaced or truncated. It closes any existing file handle, updates last_known_inode to the current inode, and resets last_known_size to 0, effectively preparing to read the *entire* new or truncated file from the beginning in the next step.

i. It detects **new content** if current_size > last_known_size.

j. If new content is detected:

- ✦ If a file handle is not already open (current_file is None), it opens the LOG_FILE_TO_MONITOR in read mode ('r') and seeks (current_file.seek()) to the last_known_size, positioning the file pointer at the end of the content that was read in the previous cycle. It ignores decoding errors (errors='ignore') which can happen with mixed encodings in log files.

- ✦ It reads all subsequent lines (current_file.readlines()) from the current position to the end of the file.

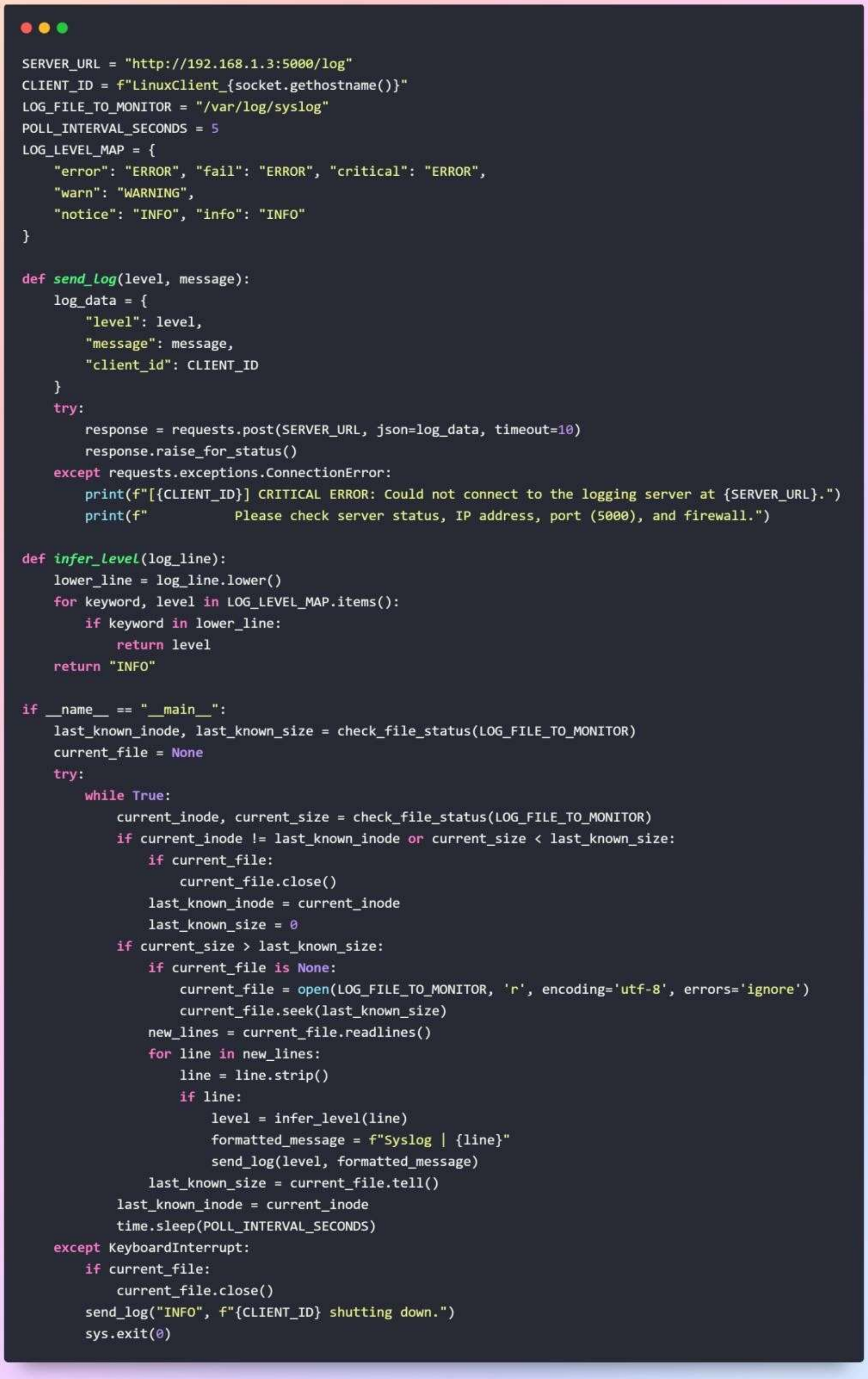
- ✦ For each new line read, it strips whitespace, infers a log level using infer_level (basic keyword matching), formats a message string (e.g., "Syslog | <original line>"), and calls send_log to send the log to the central server.

- ✦ After reading the new lines, it updates last_known_size to the current file position using current_file.tell(). The file handle (current_file) is kept open across loop iterations for efficiency, only being closed and reopened if rotation/truncation is detected.

5. After processing new content (or if there was none), it updates last_known_inode (necessary even if the inode didn't change in this cycle, to be ready for the next check) and pauses for POLL_INTERVAL_SECONDS.

6. KeyboardInterrupt and a general Exception handler provide graceful shutdown (closing the file handle) and catch critical errors, logging them and sending a final error message to the server.

PoC Code:



```python
SERVER_URL = "http://192.168.1.3:5000/log"
CLIENT_ID = f"LinuxClient_{socket.gethostname()}"
LOG_FILE_TO_MONITOR = "/var/log/syslog"
POLL_INTERVAL_SECONDS = 5
LOG_LEVEL_MAP = {
    "error": "ERROR", "fail": "ERROR", "critical": "ERROR",
    "warn": "WARNING",
    "notice": "INFO", "info": "INFO"
}

def send_log(level, message):
    log_data = {
        "level": level,
        "message": message,
        "client_id": CLIENT_ID
    }
    try:
        response = requests.post(SERVER_URL, json=log_data, timeout=10)
        response.raise_for_status()
    except requests.exceptions.ConnectionError:
        print(f"[{CLIENT_ID}] CRITICAL ERROR: Could not connect to the logging server at {SERVER_URL}.")
        print(f"           Please check server status, IP address, port (5000), and firewall.")

def infer_level(log_line):
    lower_line = log_line.lower()
    for keyword, level in LOG_LEVEL_MAP.items():
        if keyword in lower_line:
            return level
    return "INFO"

if __name__ == "__main__":
    last_known_inode, last_known_size = check_file_status(LOG_FILE_TO_MONITOR)
    current_file = None
    try:
        while True:
            current_inode, current_size = check_file_status(LOG_FILE_TO_MONITOR)
            if current_inode != last_known_inode or current_size < last_known_size:
                if current_file:
                    current_file.close()
                last_known_inode = current_inode
                last_known_size = 0
            if current_size > last_known_size:
                if current_file is None:
                    current_file = open(LOG_FILE_TO_MONITOR, 'r', encoding='utf-8', errors='ignore')
                    current_file.seek(last_known_size)
                new_lines = current_file.readlines()
                for line in new_lines:
                    line = line.strip()
                    if line:
                        level = infer_level(line)
                        formatted_message = f"Syslog | {line}"
                        send_log(level, formatted_message)
                last_known_size = current_file.tell()
            last_known_inode = current_inode
            time.sleep(POLL_INTERVAL_SECONDS)
    except KeyboardInterrupt:
        if current_file:
            current_file.close()
        send_log("INFO", f"{CLIENT_ID} shutting down.")
        sys.exit(0)
```

Analysis: 1) The inode and size tracking handles log rotation, a common Linux feature, ensuring no logs are missed.

2) The infer_level function maps log content to levels, making it adaptable to various log formats.
3) Detailed troubleshooting messages assist users in resolving connection issues, enhancing usability.

Server Implementation

server.py

Purpose: Receives, stores, and serves logs, providing APIs and web interfaces for management.

- **Key Technical Details:**
  - Uses the Flask micro web framework to handle HTTP requests and serve web pages.
  - Employs threading.Lock objects (log_lock, state_lock) to prevent race conditions when multiple threads access shared resources (app.log, log_state.json). This is necessary because app.run(threaded=True) enables basic concurrency.
  - Uses a queue.Queue (log_stream_queue) as a buffer for implementing Server-Sent Events (SSE), decoupling log reception from real-time streaming.
  - Leverages the uuid module to generate unique identifiers for each log entry upon reception.
  - Uses the re module (regular expressions) to parse log lines read from app.log.
  - Uses stream_with_context with Flask Responses for efficient SSE streaming.

Functionality:

- **REST API**:
  - /log (POST): Receives logs, appends to app.log, and queues for SSE. o

    /api/logs/<uuid>/pin, /ticket, /delete (POST): Modifies log state. o

    /api/log_stats (GET): Returns log level statistics. o /api/logs/new (GET):

    Fetches logs since a given UUID.

- **SSE**: /stream-logs streams logs in real-time using a queue.
- **Web Routes**: o /view: Displays all logs with real-time updates.
    - /tickets: Shows ticketed logs.
- **Storage**:
    - Logs in app.log with format: timestamp - [level] - [client_id] - [ID:uuid] - message.
    - State in log_state.json for pinned, ticketed, and deleted status.

Detailed Workflow/Logic:

Initializes the Flask app, file paths, locks, and the SSE queue.

Ensures the log_state.json file exists, creating an empty JSON object if not.

load_state reads log_state.json into a Python dictionary under state_lock, handling FileNotFoundError and json.JSONDecodeError.

save_state writes a Python dictionary to log_state.json under state_lock.

read_logs_with_state reads app.log line by line under log_lock. For each line, it uses LOG_LINE_REGEX to extract fields (timestamp, level, client_id, uuid, message). It then looks up the extracted uuid in the dictionary loaded by load_state to get the log's state (pinned, ticketed, deleted). It filters logs based on filter_deleted and filter_tickets parameters. Finally, it sorts the resulting list of log dictionaries by timestamp (newest first).

The /log (POST) endpoint is the primary intake for clients. It receives JSON, validates the message, assigns a default level if invalid, generates a fresh uuid.uuid4(), formats the complete log line including the UUID, and appends it to app.log under log_lock. It then creates a dictionary representation of the log and attempts to put it onto log_stream_queue using put_nowait (non-blocking, drops if queue is full). It returns a success JSON response with the assigned UUID.

The /stream-logs (GET) endpoint serves the SSE connection. It returns a Flask Response using stream_with_context and a generator function (generate_log_messages). This generator blocks on log_stream_queue.get(timeout=1), waiting for new logs. When a log is available, it's formatted into the SSE data: ...\n\n format and yielded. The timeout allows the generator to periodically yield a keep-alive comment (: keepalive\n\n) to prevent proxies/firewalls from closing the connection even when no new logs are arriving.

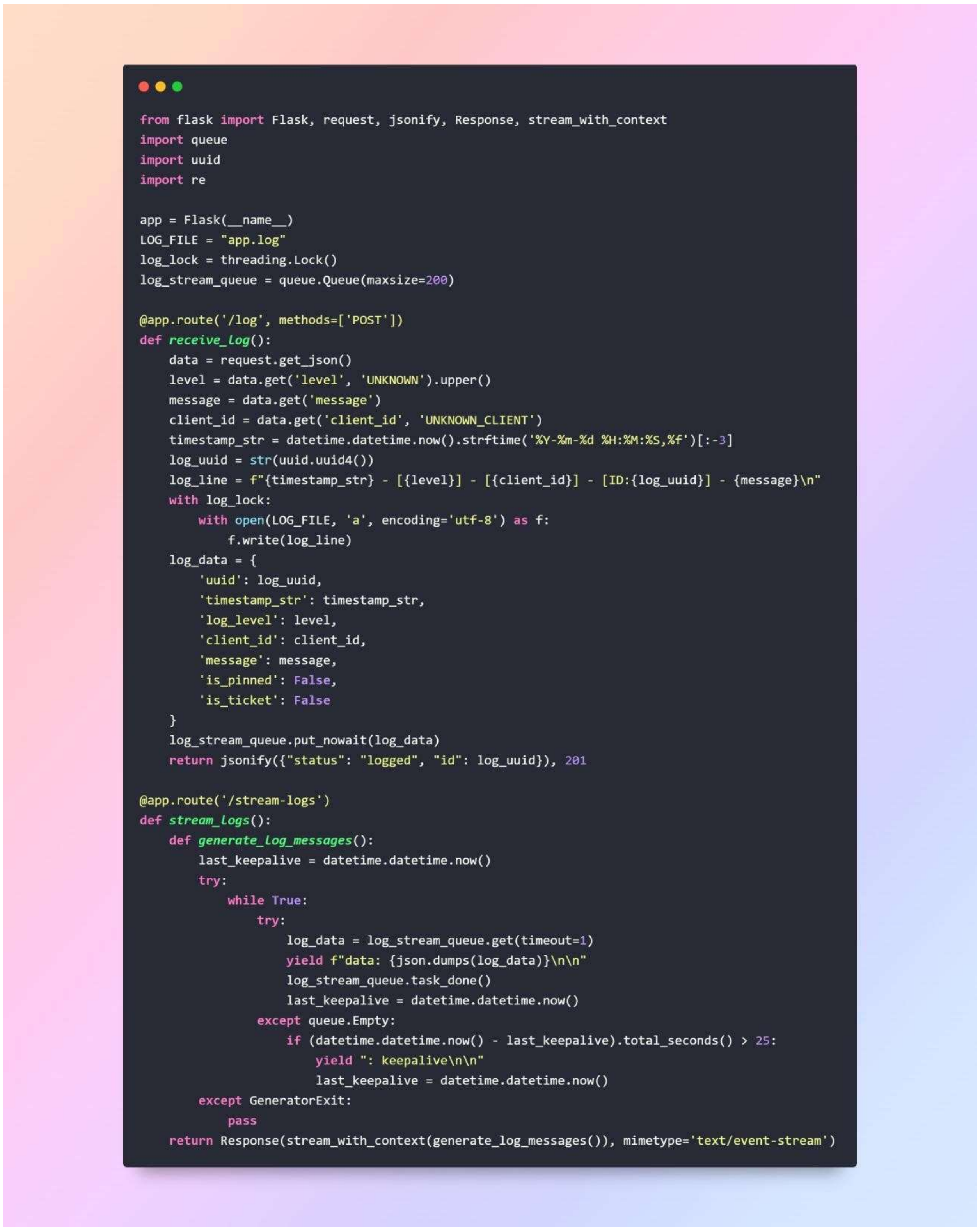
API endpoints like /api/logs/<log_uuid>/ticket receive a UUID in the URL. They load the state using load_state(), modify the specific log entry's state (ticketed flag) in the dictionary, and save the updated state using save_state(). They return a JSON response indicating the success and the new state. Deleting simply sets the deleted flag to True.

Web view endpoints (/view, /tickets) call read_logs_with_state with appropriate filters, potentially limit the number of logs for the initial load, and then render the corresponding HTML templates using render_template, passing the list of log dictionaries.

The /api/log_stats endpoint calls read_logs_with_state(filter_deleted=True) to get all active logs, iterates through them to count occurrences of each log level, and returns the counts as a JSON dictionary.

The main execution block starts the Flask development server on 0.0.0.0:5000 with threading enabled for basic concurrency.

PoC Code:



```python
from flask import Flask, request, jsonify, Response, stream_with_context
import queue
import uuid
import re

app = Flask(__name__)
LOG_FILE = "app.log"
log_lock = threading.Lock()
log_stream_queue = queue.Queue(maxsize=200)

@app.route('/log', methods=['POST'])
def receive_log():
    data = request.get_json()
    level = data.get('level', 'UNKNOWN').upper()
    message = data.get('message')
    client_id = data.get('client_id', 'UNKNOWN_CLIENT')
    timestamp_str = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S,%f')[:-3]
    log_uuid = str(uuid.uuid4())
    log_line = f"{timestamp_str} - [{level}] - [{client_id}] - [ID:{log_uuid}] - {message}\n"
    with log_lock:
        with open(LOG_FILE, 'a', encoding='utf-8') as f:
            f.write(log_line)
    log_data = {
        'uuid': log_uuid,
        'timestamp_str': timestamp_str,
        'log_level': level,
        'client_id': client_id,
        'message': message,
        'is_pinned': False,
        'is_ticket': False
    }
    log_stream_queue.put_nowait(log_data)
    return jsonify({"status": "logged", "id": log_uuid}), 201

@app.route('/stream-logs')
def stream_logs():
    def generate_log_messages():
        last_keepalive = datetime.datetime.now()
        try:
            while True:
                try:
                    log_data = log_stream_queue.get(timeout=1)
                    yield f"data: {json.dumps(log_data)}\n\n"
                    log_stream_queue.task_done()
                    last_keepalive = datetime.datetime.now()
                except queue.Empty:
                    if (datetime.datetime.now() - last_keepalive).total_seconds() > 25:
                        yield ": keepalive\n\n"
                        last_keepalive = datetime.datetime.now()
        except GeneratorExit:
            pass
    return Response(stream_with_context(generate_log_messages()), mimetype='text/event-stream')
```

Analysis:

1) The /log endpoint ensures robust log ingestion with validation.

2) SSE with a queue manages backpressure, ensuring real-time updates without overwhelming the server.

3) Thread-safe file access prevents data corruption.

Web Interfaces

view_logs.html

Purpose: Displays all logs with real-time updates and a log level distribution chart.

Functionality:

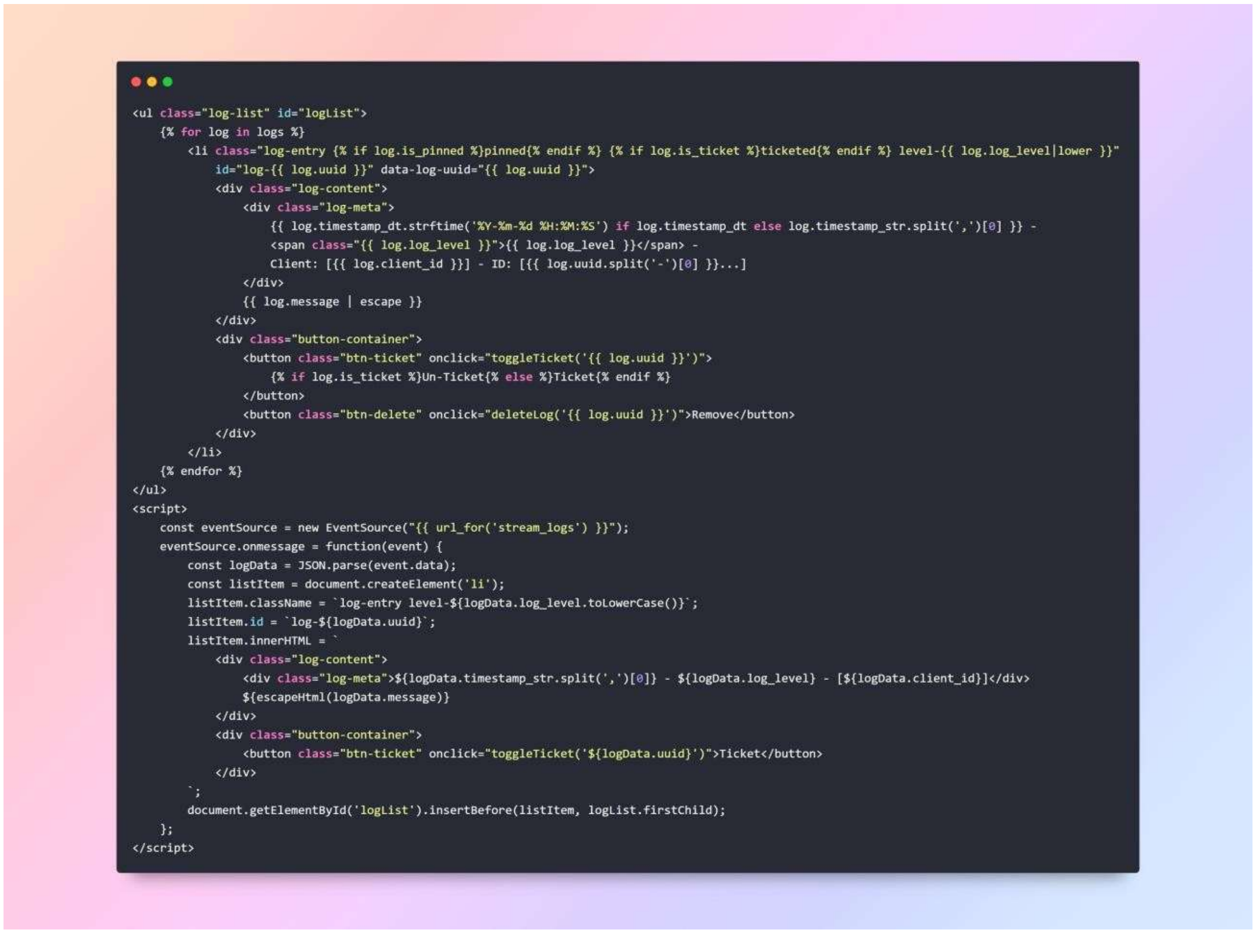
1) Lists logs with timestamp, level, client ID, shortened UUID, and message.

2) Supports ticketing and deletion with animated fade-out effects.

3) Uses SSE to prepend new logs in real-time.

4) Displays a bar chart (Chart.js) showing log counts by level.

PoC Code:



```
<ul class="log-list" id="logList">
    {% for log in logs %}
        <li class="log-entry {% if log.is_pinned %}pinned{% endif %} {% if log.is_ticket %}ticketed{% endif %} level-{{ log.log_level|lower }}"
            id="log-{{ log.uuid }}" data-log-uuid="{{ log.uuid }}">
            <div class="log-content">
                <div class="log-meta">
                    {{ log.timestamp_dt.strftime('%Y-%m-%d %H:%M:%S') if log.timestamp_dt else log.timestamp_str.split(',')[0] }} -
                    <span class="{{ log.log_level }}">{{ log.log_level }}</span> -
                    Client: [{{ log.client_id }}] - ID: [{{ log.uuid.split('-')[0] }}...]
                </div>
                {{ log.message | escape }}
            </div>
            <div class="button-container">
                <button class="btn-ticket" onclick="toggleTicket('{{ log.uuid }}')">
                    {% if log.is_ticket %}Un-Ticket{% else %}Ticket{% endif %}
                </button>
                <button class="btn-delete" onclick="deleteLog('{{ log.uuid }}')">Remove</button>
            </div>
        </li>
    {% endfor %}
</ul>
<script>
    const eventSource = new EventSource("{{ url_for('stream_logs') }}");
    eventSource.onmessage = function(event) {
        const logData = JSON.parse(event.data);
        const listItem = document.createElement('li');
        listItem.className = `log-entry level-${logData.log_level.toLowerCase()}`;
        listItem.id = `log-${logData.uuid}`;
        listItem.innerHTML = `
            <div class="log-content">
                <div class="log-meta">${logData.timestamp_str.split(',')[0]} - ${logData.log_level} - [${logData.client_id}]</div>
                ${escapeHtml(logData.message)}
            </div>
            <div class="button-container">
                <button class="btn-ticket" onclick="toggleTicket('${logData.uuid}')">Ticket</button>
            </div>
        `;
        document.getElementById('logList').insertBefore(listItem, logList.firstChild);
    };
</script>
```

Analysis:

1) SSE ensures real-time log updates without polling.
2) The chart dynamically updates, enhancing user interaction.
3) HTML escaping prevents XSS vulnerabilities.

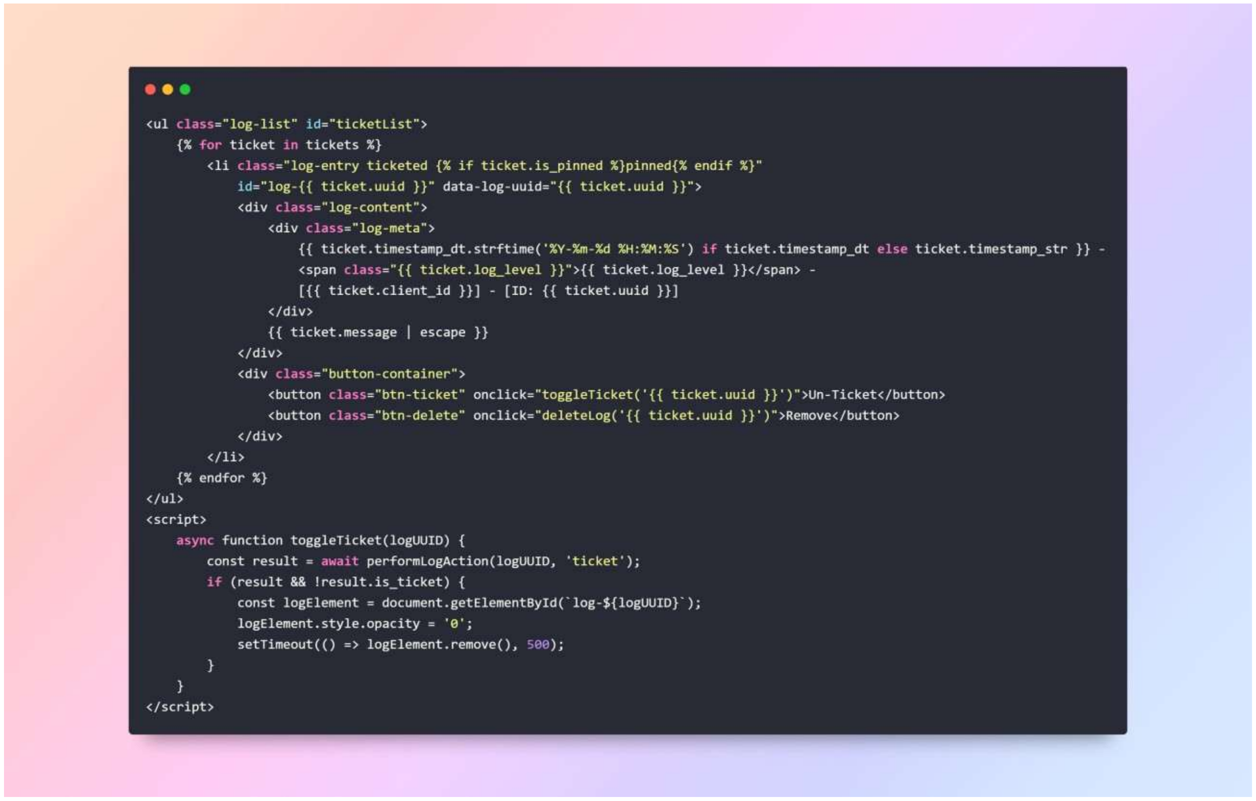
tickets.html

Purpose: Displays ticketed logs for issue tracking.

Functionality:

1) Lists only ticketed logs with un-ticket and delete buttons.
2) Uses similar styling to view_logs.html for consistency.
3) No SSE or chart, focusing on static ticket management.

PoC Code:



```
<ul class="log-list" id="ticketList">
    {% for ticket in tickets %}
        <li class="log-entry ticketed {% if ticket.is_pinned %}pinned{% endif %}"
            id="log-{{ ticket.uuid }}" data-log-uuid="{{ ticket.uuid }}">
            <div class="log-content">
                <div class="log-meta">
                    {{ ticket.timestamp_dt.strftime('%Y-%m-%d %H:%M:%S') if ticket.timestamp_dt else ticket.timestamp_str }} -
                    <span class="{{ ticket.log_level }}">{{ ticket.log_level }}</span> -
                    [{{ ticket.client_id }}] - [ID: {{ ticket.uuid }}]
                </div>
                {{ ticket.message | escape }}
            </div>
            <div class="button-container">
                <button class="btn-ticket" onclick="toggleTicket('{{ ticket.uuid }}')">Un-Ticket</button>
                <button class="btn-delete" onclick="deleteLog('{{ ticket.uuid }}')">Remove</button>
            </div>
        </li>
    {% endfor %}
</ul>
<script>
    async function toggleTicket(logUUID) {
        const result = await performLogAction(logUUID, 'ticket');
        if (result && !result.is_ticket) {
            const logElement = document.getElementById(`log-${logUUID}`);
            logElement.style.opacity = '0';
            setTimeout(() => logElement.remove(), 500);
        }
    }
</script>
```

Analysis:

- Streamlined interface for ticket management.
- Reuses JavaScript functions for maintainability.

Strengths

- **Scalability**: Handles multiple clients and concurrent requests.
- **Real-Time Monitoring**: SSE provides instant log updates.
- **Decoupled Components**: Clients operate independently of each other and asynchronously from the server's processing once a log is sent.
- **Heterogeneous Log Source Integration:** The system successfully aggregates logs from generic applications, structured Windows Event Logs, and unstructured Linux text files using a common network protocol and server endpoint.
- **Fault Tolerance**: Retries, fallback storage, and error handling ensure reliability.
- **Cross-Platform**: Supports Windows (client_eventlog.py) and Linux (client_linux.py).
- **User-Friendly**: Intuitive interfaces with charts and action buttons.
- **Extensibility**: File-based storage can be replaced with a database.

Conclusion

The Distributed Logging System, now enhanced with client_linux.py, effectively demonstrates distributed computing principles through decentralized log generation, centralized aggregation, and real-time monitoring. Its robust architecture, comprehensive error handling, and cross-platform support make it a practical solution for monitoring distributed applications. The addition of Linux log monitoring broadens its applicability, while PoC code showcases its functionality. Despite limitations like file-based storage, the system is well-positioned for enhancements, making it a strong foundation for advanced log management.

GitHub Link: CODE