DOCUMENT: Backend Overview and Pseudocode

PROJECT: HPE Network Analysis Dashboard

TABLE OF CONTENTS:

1. main.py

CONTEXT:

- **Role:** The entry point of application and process lifecycle manager. It initializes the environment, handles OS signals for graceful shutdown, and launches the core WebSocket server.
- **Input:** Operating System signals (SIGINT, SIGTERM) and Keyboard Interrupts.
- **Output:** Initialized backend processes, console status logs, and clean process termination.
- **Dependencies:** asyncio, signal, websocket_server, capture_manager, Shared_State.

FLOW:

1. SIGNAL HANDLING & SAFETY

FUNCTION: signal_handler(sig, frame)

- **Trigger:** Invoked when the Operating System sends a termination signal (e.g., Ctrl+C or kill command).
- **Action:** Prints a shutdown message and sets shared_state.capture_active = False to immediately signal data collection loops to stop.
- **Logic:** Checks if an asyncio event loop is currently running.
  - If YES: Schedules the cleanup_and_exit() coroutine as a background task.
  - If NO: Forces an immediate hard exit using os._exit(0).

FUNCTION: cleanup_and_exit()

- **Role**: The asynchronous destructor for the application.
- **Action:** Awaits capture_manager.stop_tshark() to ensure the external tshark subprocess is terminated safely before the Python script dies.
- **Finalize:** Calls os._exit(0) to ensure the process terminates completely.

2. APPLICATION BOOTSTRAP

FUNCTION: main()

- **Setup**: Registers the signal_handler to listen for signal.SIGINT and signal.SIGTERM.
- **Discovery:** Calls capture_manager.get_device_ips() to identify and store local network interfaces and IP addresses (IPv4/IPv6) before the server starts.
- **Launch:** Executes asyncio.run(start_websocket_server()). This starts the main event loop and blocks execution here while the server runs.

- **Error Handling:** Wraps execution in a try/except block to catch KeyboardInterrupt (User Interruption) and generic Exceptions, ensuring errors are logged.
- **Shutdown:** Executes the finally block to print "Application shutdown complete" when the event loop ends.

3. ENTRY POINT

- **Condition:** if __name__ == "__main__":
- **Action:** Calls main() to start the application.

FOR PSEUDOCODE : Click here

2. capture_manager.py

CONTEXT:

- Role: The bridge between the Operating System's network layer and the Dashboard.
- Input: Raw network packets captured via the 'tshark' command-line tool.
- Output: Parsed packet objects and stream data stored in SHARED_STATE.
- Dependencies: Tshark (Wireshark), AppDetector, SharedState Module.

FLOW:

1. INITIALIZATION & DISCOVERY

FUNCTION: get_device_ips()

- Action: Uses `psutil.net_if_addrs()` to find local IP addresses.
- Target: Populates `shared_state.ip_address`, `ipv4_ips`, and `ipv6_ips`. (Crucial for distinguishing "Inbound" vs "Outbound" traffic later).

FUNCTION: get_network_interfaces()

- Action: Runs `subprocess.run(["tshark", "-D"])`.
- Result: Returns a list of available interfaces (e.g., "Wi-Fi", "Ethernet") for the UI dropdown.

2. CAPTURE SESSION LIFECYCLE

Triggered when the user clicks "Start" on the frontend.

FUNCTION: start_tshark(interface)

- Check: Is `shared_state.tshark_proc` already running?
  If YES: Return "Tshark already running".
- Construct Command: Prepare the `tshark` command with specific flags to extract fields: [frame.time, ip.src, ip.dst, dns.qry.name, tls.handshake...]
- Launch: Spawn the subprocess asynchronously. Set `shared_state.capture_active = True`.

3. THE PACKET LOOP

This is the main engine that runs while the dashboard is live.

FUNCTION: capture_packets(duration)

While `time.time() < duration` AND capture is active:

- Ingestion:
  Read a line from `tshark_proc.stdout`. Split the raw string by **"|"** into `parts`.
- Intelligence Layer:
  Call app_detector.detect_application(src, dst, ports, sni...):

- Identify if traffic is "Netflix", "Zoom", or "Unknown".
- Update `shared_state.ip_stats` for the GeoMap.

- Parsing & Storage

  Call parse_and_store_packet(parts):
  - Create a lightweight packet object (No, Time, Source, Dest, Info).
  - Append to `shared_state.all_packets_history`. (This list directly feeds the "Live Traffic" table in the UI).
- Stream Grouping

  Identify the connection type to key the data for metrics:
  - If TCP -> Key is `("tcp", stream_id)`
  - If UDP -> Key is `("udp", stream_id)`

  Append raw data to `shared_state.streams[key]`.
  (The `metrics_calculator` will read this later to compute throughput/latency).

4. CLEANUP & RESET

Triggered when the user clicks "Stop" or "Reset".

FUNCTION: stop_tshark():

- Signal the loop to break (`capture_active = False`).
- Terminate the `tshark_proc` subprocess safely.

FUNCTION: resetSharedState()

- Wipe the slate clean:
- Clear `shared_state.all_packets_history`.
- Clear `shared_state.streams`.
- Reset all throughput counters in `metrics_state`.

FOR PSEUDOCODE: Click here

3. websocket_server.py

CONTEXT:

- **Role:** It manages client connections, processes frontend commands, and coordinates the asynchronous background tasks for data collection, metric calculation, and AI summarization.
- **Input:** JSON commands from the Frontend (React), Raw packet streams, and System signals.
- **Output:** Real-time JSON data streams (metrics, packets, logs) broadcast to connected clients.
- **Dependencies:** asyncio, websockets, json, capture_manager, metrics_calculator, shared_state, llm_summarizer, geolocation_handler.

FLOW

1. SERVER INITIALIZATION

FUNCTION: start_websocket_server()

- **Action:** Initializes the WebSocket server on localhost.
- **Background Tasks:** Spawns three concurrent asynchronous loops to run alongside the server:
    - data_collection_loop(): For real-time metric updates.
    - geolocation_loop(): For resolving IP locations.
    - periodic_summary_loop(): For generating AI insights every 60s.
- **Wait:** Keeps the main event loop running indefinitely.

2. CLIENT CONNECTION LIFECYCLE

FUNCTION: websocket_handler(websocket)

- **Trigger:** A new client connects (e.g., user opens the dashboard in a browser).
- **Safety Check:** If shared_state.is_resetting is True, wait until the previous session clears.
- **Registration:** Adds the new client object to shared_state.connected_clients.
- **Bootstrap:** Immediately sends an initial_state message containing current metrics, packet history, and interface lists so the UI populates instantly.
- **Message Loop:** Awaits incoming JSON commands. Dispatches them to handle_command or handle_stop_capture_task.

- **Teardown:** Triggered on disconnect.
    - Remove client from the active list.
    - **Auto-Stop:** If NO clients remain, automatically calls capture_manager.stop_tshark() and resets the system state to save resources.

3. DATA BROADCASTING

FUNCTION: data_collection_loop()

- **Frequency:** Runs continuously (approx. every 0.1s).
- **Condition:** Active only if capture_active is True AND clients are connected.
- **Step 1 (Ingest):** Call capture_manager.capture_packets() to read buffered data from Tshark.
- **Step 2 (Compute):** Call metrics_calculator.calculate_metrics() to update throughput, latency, and jitter.
- **Step 3 (Broadcast):** Construct a JSON "update" payload (Metrics, New Packets, Top Talkers, Geo-IPs) and send it to all connected clients.

4. COMMAND PROCESSING

FUNCTION: handle_command(command, data)

- **start_capture:** Validates state (rejects if AI summary is currently generating).
- **get_interfaces:** Returns list of network adapters from capture_manager.

FUNCTION: handle_stop_capture_task(websocket, data)

- **Role:** Handles the complex stop flow as a background task to prevent blocking.
- **Action:** Stops Tshark and sets capture_active = False.
- **AI Integration:** Triggers llm_summarizer.generate_summary() to create the final session report.
- **Response:** Sends the final JSON response (Success + AI Summary) to the client.
- **Reset:** Calls capture_manager.resetSharedState() to prepare for the next run.

5. AI INTELLIGENCE LOOP

FUNCTION: periodic_summary_loop()

- **Logic:** Checks elapsed time every 10 seconds.
- **Trigger:** If session > 60s and time since last summary > 60s.

- **Action:** Generates a text summary llm_summarizer.generate_periodic_summary() and broadcasts it to the "Summary Bot" tab on the frontend.

FOR PSEUDOCODE: Click here

4. llm_summarizer.py

CONTEXT:

- **Role:** It translates raw technical metrics into human-readable network health reports and real-time status updates.
- **Input:** Aggregated metrics from shared_state (Throughput, Latency, Jitter, Protocol counts) and the GEMINI_API_KEY.
- **Output:** JSON-formatted text summaries ("Final Report" and "Periodic Updates") for the frontend UI.
- **Dependencies:** google.generativeai, dotenv, json, shared_state, datetime.

FLOW:

1. SETUP & UTILITIES

- **Initialization:** Loads environment variables to retrieve the GEMINI_API_KEY and configures the genai client.

FUNCTION: _format_throughput(bits)

- **Action:** Helper that converts raw bit counts into human-readable strings (e.g., "15.4 Mbps", "2.1 Gbps").

FUNCTION: analyze_protocol_performance(metrics)

- **Action:** Extracts average performance stats (Throughput, Latency, Jitter) for a specific protocol to prepare them for the AI prompt

2. FINAL SESSION REPORTING

FUNCTION: generate_summary()

- **Trigger:** Called by websocket_server immediately after the user clicks "Stop".
- **Step 1 (Data Aggregation):**
  - Overall: Calculates session duration, total PPS, and average Inbound/Outbound Throughput & Goodput.
  - Composition: Gathers final counts for IPv4/IPv6 and Encrypted/Unencrypted traffic.
  - Protocols: Iterates through TCP (Packet Loss/Latency), RTP (Jitter/Loss), UDP, QUIC, etc., to gather specific health metrics.

- **Step 2 (Prompt Engineering):**
  - Constructs a strict prompt defining the AI's persona as an "Expert Network Analyst" and feeds the pre-calculated data JSON into the prompt.
- **Step 3 (Execution):**
  - Awaits the Gemini API response and parses the result string into a valid Python dictionary to send to the frontend.

3. REAL-TIME INSIGHTS

FUNCTION: generate_periodic_summary()

- **Trigger:** Called every 60 seconds by the periodic_summary_loop in websocket_server
- **Prompting:** Constructs a concise prompt instructing Gemini to generate a short, 3-4 sentence status summary.
- **Execution:** Awaits the AI's JSON response and returns a timestamped object for the frontend "Summary Bot"
- **Fallback:** Returns a hardcoded string with basic statistics if the API key is missing or the request fails.

FOR PSEUDOCODE: Click here

5. geolocation_handler.py

CONTEXT:

- **Role:** It resolves IP addresses to physical locations (City, Country, Coordinates) and Hostnames to visualize traffic origins on the world map.
- **Input:** Unique public IP addresses extracted from captured packets.
- **Output:** enriched Geolocation objects appended to shared_state.new_geolocations for the frontend.
- **Dependencies:** aiohttp, asyncio, ipaddress, socket, shared_state, static_geolocation_db.

FLOW:

1. VALIDATION & UTILITIES

FUNCTION: is_public_ip(ip_str)

- **Action:** specific check to ensure the IP is routable over the public internet.
- **Logic:** Returns False for Local (LAN), Loopback, or Private ranges (e.g., 192.168.x.x), as these cannot be geolocated.

2. DATA ENRICHMENT

FUNCTION: fetch_geolocation(session, ip)

- **Reverse DNS:** Resolves the IP's hostname via socket.gethostbyaddr to provide context beyond just coordinates
- **Static Lookup:** Checks the internal STATIC_GEOLOCATION_DB first to instantly satisfy requests for known high-volume servers.
- **Rate Limiting:** Enforces a mandatory delay (min_time_between_calls) before external requests to strictly adhere to the API's quota.
- **External Query:** Fetches live City, Country, and Coordinate data from ip-api.com only when local resolution fails.
- **Result:** Returns a unified location dictionary or None, prioritizing cached data to minimize latency.

3. BATCH ORCHESTRATION

FUNCTION: process_geolocation_batch(ips_to_query)

- **Action:** Iterates sequentially through a list of IPs to respect the rate limit.
- **Merger:**
  - Calls fetch_geolocation.

- Injects passively captured DNS names from shared_state.ip_to_dns if available.
- Injects Application detection tags from shared_state.ip_stats.

- **Storage:** Appends the final object to shared_state.new_geolocations, which the WebSocket server consumes and clears.

4. CONTINUOUS DISCOVERY

FUNCTION: geolocation_loop()

- **Frequency:** Runs every 2 seconds in the background.
- **Action:**
  - Calls extract_ips_from_packets() to find *new* public IPs in the active streams that haven't been queried yet.
  - Updates the shared_state.queried_public_ips set to prevent duplicate work.
  - Triggers process_geolocation_batch for the new targets.

FOR PSEUDOCODE: Click here

6. app_detector.py

CONTEXT:

- **Role:** The classification engine that tags network traffic with specific application names (e.g., "Netflix", "Slack") and categories (e.g., "Video", "Messaging").
- **Input:** Packet details including Source/Dest IPs, Ports, Protocols, DNS queries, and TLS/QUIC SNI tags.
- **Output:** An application info dictionary (e.g., {'app': 'Zoom', 'category': 'Video Call'}) or "Unknown".
- **Dependencies:** None (Standalone logic module utilizing internal static dictionaries and runtime caching).

FLOW:

1. KNOWLEDGE BASE & CACHING

- **Static Dictionaries:**
  - **Action:** Loads predefined mappings for identification ,maps domain patterns and port mapping.
- **Dynamic Cache:**
  - **Action:** Maintains ip_to_app_cache, a runtime dictionary that learns IP-to-App associations from DNS/SNI traffic to identify subsequent packets.

2. MAIN DETECTION PIPELINE

FUNCTION: detect_application(src_ip, dst_ip, src_port, dst_port, ...)

- **Role:** The interface called by capture_manager for every packet to determine its origin.
- **Trigger:** Invoked for every single packet captured to assign an Application Name and category (e.g., "Video", "Social Media").
- **Deep Inspection:** Prioritizes sni_hostname (TLS) and quic_sni (QUIC) to instantly identify encrypted traffic sources like Netflix or Zoom.
- **Deep Inspection:** Prioritizes sni_hostname (TLS) and quic_sni (QUIC) to instantly identify encrypted traffic sources like Netflix or Zoom.
- **Cache Lookup:** Checks if the packet's Source or Destination IP matches a previously identified service in the runtime memory.
- **Fallback:** Defaults to analyzing the Destination Port (e.g., 53=DNS, 443=HTTPS) using PORT_MAPPINGS if no higher-layer metadata is available.

3. SUPPORTING UTILITIES

FUNCTION: identify_app_from_domain(domain)

- **Action:** Iterates through DOMAIN_PATTERNS to find a matching keyword within the provided domain string.

FUNCTION: cache_dns_mapping(ip, domain)

- **Action:** Updates the global ip_to_app_cache to associate a specific IP address with an identified application for future reference.

FOR PSEUDOCODE: Click here

7. metrics_calculator.py

CONTEXT:

- **Role:** The computational engine that processes raw packet streams to generate real-time performance metrics
- **Input:** Raw packet data organized by stream from shared_state.streams.
- **Output:** Updated global metric objects in shared_state.py
- **Dependencies:** datetime, shared_state.

FLOW:

1. UTILITIES & HELPERS FUNCTION: update_encryption_composition(protocol, metrics)

- **Action:** Checks if the protocol name contains keywords like "TLS", "QUIC", or "SSH".
- **Target:** Increments encrypted_packets or unencrypted_packets counters to track security adoption.

FUNCTION: update_running_metrics(key, metrics, ...)

- **Action:** Updates the persistent running_state with new peak and sum values.
- **Logic:** Calculates moving averages for Throughput, Latency, and Jitter to ensure historical accuracy beyond the current capture window.

FUNCTION: update_top_talkers(src, dst, length)

- **Action:** Tracks the volume of data exchanged between specific Source and Destination IPs.
- **Filter:** Only processes "Outbound" traffic (where Source IP belongs to the local device).

2. CORE CALCULATION ENGINE Trigger: Invoked periodically by the websocket_server loop.

FUNCTION: calculate_metrics()

- **Step 1: Reset & Initialize**
  - Checks if shared_state.streams is empty. If yes, zeroes out all metrics and returns.
  - Creates temporary metric templates (tcp_temp, rtp_temp, etc.) using make_temp_metrics().
- **Step 2: Stream Processing (The Loop)**
  - Iterates through every stream in shared_state.streams.
  - **TCP Analysis:**
    - Extracts tcp.analysis.ack_rtt for Latency calculations.
    - Counts retransmissions (tcp.analysis.retransmission) to compute Packet Loss.

- **RTP Analysis (VoIP/Video):**
  - Detects sequence number gaps to calculate precise Packet Loss.
  - Computes Jitter using RFC 3550 logic, dynamically detecting clock rates.
- **General Protocols (UDP, QUIC, DNS):**
  - Accumulates byte counts for Inbound/Outbound Throughput.
  - Calculates Goodput by subtracting header overheads (Ethernet, IP, TCP/UDP headers).

- **Step 3: Aggregation & Normalization**
  - Converts total accumulated bytes into Bits Per Second (bps) based on the capture duration.
  - Calculates weighted averages for Latency and Jitter (weighted by packet count per stream).
  - Computes final percentages for Packet Loss, IPv4/IPv6 distribution, and Encryption.

3. STATE SYNCHRONIZATION FUNCTION: Finalize Updates

- **Persistence:** Calls update_running_metrics for every protocol to update the global Peak and Average statistics.
- **Publishing:**
  - Updates shared_state.metrics_state with the new snapshot.
  - Updates protocol-specific dictionaries (shared_state.tcp_metrics, etc.).
  - calls calculate_top_talkers() to refresh the "Top 7" visualization list.

FOR PSEUDOCODE: Click here

8. shared_state.py

CONTEXT:

- **Role:** The central in-memory database and state manager for the application. It acts as the single source of truth, accessible by all backend modules to read/write global data.
- **Input:** Raw packet data from capture_manager and calculated statistics from metrics_calculator.
- **Output**: Unified state objects consumed by websocket_server (for broadcasting) and llm_summarizer (for reporting).
- **Dependencies:** None. (This module is a dependency for others).

FLOW:

1. PROCESS CONTROL VARIABLE: capture_active

- **Role:** Master boolean flag for the capture loop.
- **Action:** Set True/False by capture_manager to start or stop data ingestion.

VARIABLE: is_resetting

- **Role:** Concurrency lock.
- **Action:** Prevents new WebSocket connections while the previous session is clearing.

VARIABLE: tshark_proc

- **Role:** Handle for the external OS process.
- **Action:** stored to allow immediate termination of the Tshark subprocess.

2. DATA STORAGE (BUFFERS) VARIABLE: all_packets_history

- **Role:** Complete list of parsed packet dictionaries.
- **Action:** Feeds the "Live Traffic" table; cleared on reset.

VARIABLE: streams

- **Role:** Dictionary grouping packets by connection (e.g., ('tcp', stream_id)).
- **Action:** Used by metrics_calculator to compute latency and jitter across related packets.

VARIABLE: ip_address

- **Role:** List of local device IPs.
- **Action:** Populated at startup to distinguish Inbound vs. Outbound traffic.

3. METRICS & ANALYTICS VARIABLE: metrics_state

- **Role:** The primary state object broadcast to the Frontend.
- **Action:** Updates continuously with Throughput, Goodput, and PPS values.

VARIABLE: tcp_metrics / rtp_metrics

- **Role:** Protocol-specific health counters.
- **Action:** Stores computed Latency (TCP), Jitter (RTP), and Packet Loss data.

VARIABLE: top_talkers_top7

- **Role:** Visualization data for the Sankey diagram.
- **Action:** sorted list of the highest volume source-destination pairs.

VARIABLE: running_state

- **Role:** Internal accumulator for averages.
- **Action:** Tracks sums and counts (not sent to frontend) to compute accurate averages over time.

VARIABLE: new_geolocations

- **Role:** A temporary buffer holding newly resolved IP locations (Latitude / Longitude/City).
- **Usage:** Populated by geolocation_handler and cleared by websocket_server immediately after broadcasting to the map.

VARIABLE: Global Counters (tcp_lost_packets_total, etc.)

- **Role:** Accumulators for session-wide statistics that must persist across individual capture ticks (e.g., total packets lost since start).

FOR PSEUDOCODE: Click here