Low-Level Design Document (LLD)

Network Analyzer Dashboard Project

Table of Contents :

1. Project Overview :

Project Name: Network Analyzer Dashboard

Description:

This backend service captures network traffic live from devices, analyzes the data to compute network performance metrics like throughput, latency, jitter, and packet loss, and streams this data efficiently to multiple clients via WebSocket connections. It leverages tshark for packet capture, uses asynchronous Python (asyncio) for handling concurrent clients, and maintains a robust, shared state across modules.

The backend is designed to work alongside a **Frontend Dashboard** application that visualizes this data in real-time, providing network administrators with intuitive insights through interactive graphs and charts.

---

2. Objectives :

- Provide real-time network packet capturing and analysis.
- Deliver live updates of network performance metrics to multiple clients.
- Support dynamic start/stop of capture sessions on chosen network interfaces.
- Provide real time visualizations through graphs and charts for better understanding.

---

3. Technology Stack :

Component	Technology/Library
Programming Language	Python
Packet Capture Tool	TShark (Wireshark CLI)
Async WebSocket	websockets Python library
Data Serialization	JSON
Frontend Framework	React.js
Visualization Libraries	recharts
OS Compatibility	Cross-platform (Linux, Windows, macOS)

---

4. Prerequisites:

The following tools must be installed and properly configured for the project to run:

- ***TShark*** – Used for capturing and analyzing network traffic from the command line. It is a terminal-based interface of Wireshark.
- ***Node.js*** – Required for running the React frontend and managing JavaScript-based dependencies.
- ***Python*** – Used for implementing the backend logic and handling server-side processes.

All tools should be accessible through environment variables or system path to ensure seamless execution.

---

5. System Architecture:

Diagram link:
https://www.canva.com/design/DAGy2Trz8Vg/bQebl0RzV6GvHz7vLijOFA/edit?utm_content=DAGy2Trz8Vg&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

6. Module Details :

6.1 capture_manager.py module

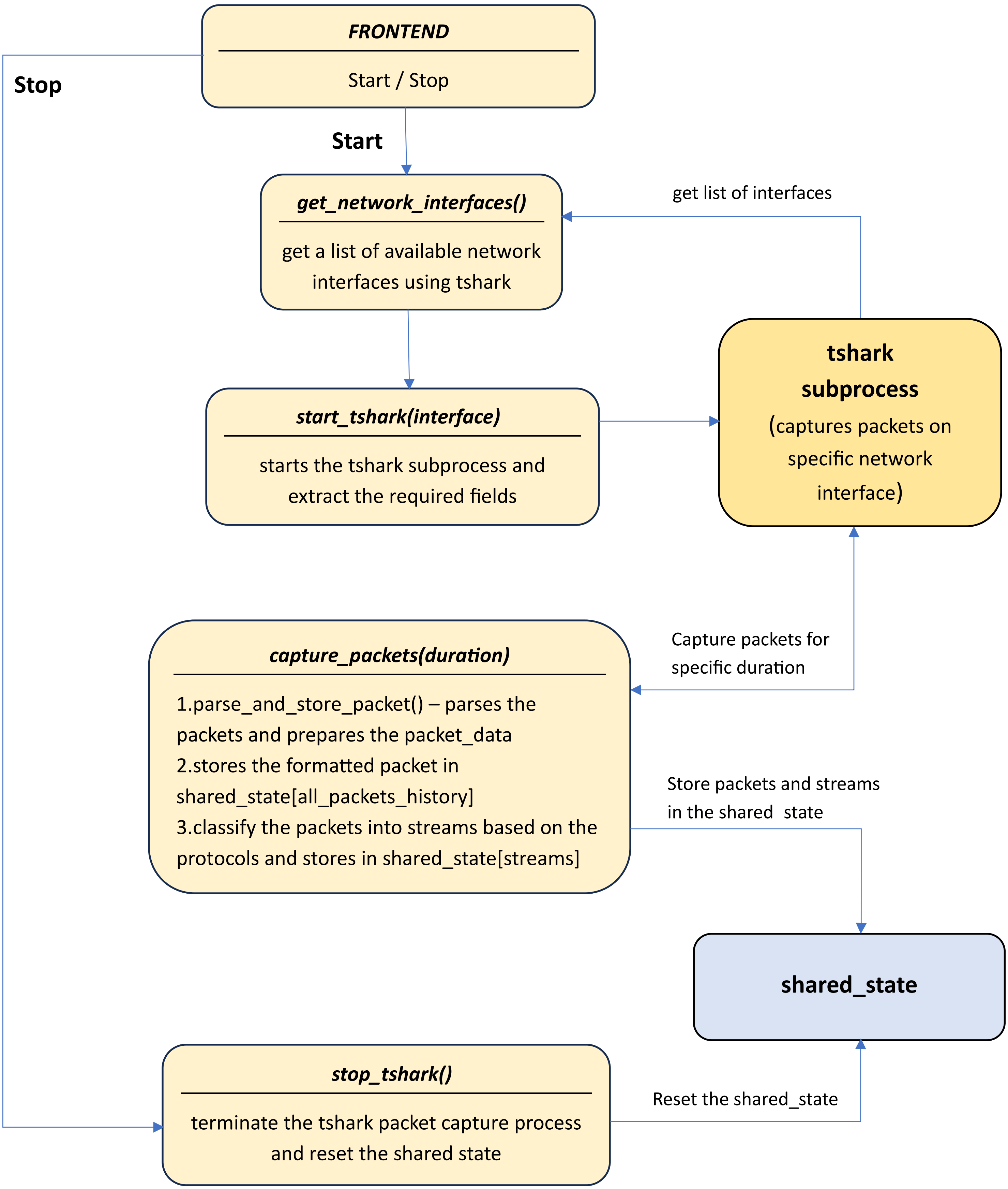
PURPOSE :

- Starting and stopping live packet capture using TShark
- Reading raw packet data line-by-line in real time
- Parsing and formatting packets for display on the frontend
- Grouping packets into logical streams (like TCP stream 1, UDP stream 2, etc.)
- Updating shared memory so that other parts (metrics, frontend) can use the data

CORE FUNCTIONS :

- **start_tshark()**: This function starts the tshark subprocess, providing it with the necessary command-line arguments to capture packets and format the output.
- **stop_tshark()**: This function gracefully terminates the tshark subprocess.
- **get_formatted_packets()**: A utility function that retrieves a formatted list of packets for the frontend display.
- **capture_packets()**: A function that reads packets from tshark's standard output, parses them into a structured format, and groups them into streams. This data is then stored in the shared_state module.
- **get_packet_statistics()**: A function that calculates the total packet count, stream count, and protocol distribution from the captured data.
- **get_network_interfaces():** A function get a list of available network interfaces using tshark.

FLOW DIAGRAM:



PSEUDO CODE :

function get_device_ips(interface_name):

```
   # Step 1: Initialize empty list to hold device IPs
   device_ips = []

   # Step 2: Get all network interfaces and their addresses
   all_addrs = psutil.net_if_addrs()

   # Step 3: Verify interface exists; if not, print error and return empty list
   if interface_name not in all_addrs:
      return device_ips

   # Step 4: Iterate through addresses on interface
   for addr in all_addrs[interface_name]:
      # Add IPv4 addresses
      if addr.family == socket.AF_INET:
         device_ips.append(addr.address)
      # Add IPv6 addresses both with and without zone info
      elif addr.family == socket.AF_INET6:
         device_ips.append(addr.address)  # with zone
         device_ips.append(addr.address.split('%')[0])  # without zone

   # Step 5: Remove duplicates from device_ips
   device_ips = list(set(device_ips))

   # Step 6: Return the list of IP addresses
   return device_ips
```

---

function get_network_interfaces():

```
   # Step 1: Run tshark -D command to list interfaces
   proc = subprocess.run(["tshark", "-D"], capture_output=True, text=True)
```

```
    # Step 2: Parse output line by line
    interfaces = []
    for line in proc.stdout.strip().split('\n'):
        match = re.search(r'^\d+\.\s+(.+?)(?:\s+\((.+)\))?$', line)
        if match:
            interface_id = match.group(1).strip()
            descriptive_name = match.group(2).strip() if match.group(2) else interface_id
            # Step 3: Add interface details to list
            interfaces.append({"id": interface_id, "name": descriptive_name})

    # Step 4: Return parsed list of interfaces
    return interfaces
```

---

function start_tshark(interface="Wi-Fi"):

```
    # Step 1: Check if tshark already running
    if shared_state.tshark_proc is not None:
        return False, "Tshark already running"

    # Step 2: Build tshark command with desired capture fields
    tshark_cmd = [
        "tshark", "-i", interface, "-T", "fields",
        "-e", "frame.number", "-e", "frame.time_epoch", "-e", "ip.src",
        # ...other fields...
        "-E", "separator=|", "-E", "header=n", "-E", "quote=n"
    ]

    # Step 3: Start tshark as subprocess and save process handle
    shared_state.tshark_proc = subprocess.Popen(
        tshark_cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True, bufsize=1
    )
    # Step 4: Mark capture as active
    shared_state.capture_active = True

    # Step 5: Return success message
```

```
    return True, "Tshark started successfully"
```

```
function parse_and_store_packet(parts):
    # Step 1: Extract fields from parts list
    frame_number = parts[0] or "N/A"
    timestamp = parts[1] or "N/A"
    source_ip = parts[2] or parts[16] or "N/A"
    dest_ip = parts[3] or parts[17] or "N/A"
    length = parts[4] or "0"
    protocols = parts[5] or "N/A"
    info = parts[6] or "N/A"

    # Step 2: Convert timestamp to readable format if valid
    if timestamp != "N/A":
        ts_float = float(timestamp)
        formatted_time = datetime.fromtimestamp(ts_float).strftime("%H:%M:%S.%f")[:-3]
    else:
        formatted_time = "N/A"

    # Step 3: Truncate info field if too long
    if len(info) > 100:
        info = info[:100] + "..."

    # Step 4: Return structured packet dictionary
    return {
        "no": frame_number,
        "time": formatted_time,
        "source": source_ip,
        "destination": dest_ip,
        "protocol": protocols,
        "length": length,
        "info": info
    }
```

```
function capture_packets(duration):
    # Step 1: Confirm tshark process running and capture active
    if not shared_state.tshark_proc or not shared_state.capture_active:
        return

    # Step 2: Clear existing streams and packet history
    shared_state.streams = {}
    shared_state.all_packets_history = []

    # Step 3: Note start time
    start = time.time()

    # Step 4: While duration not exceeded and capture active
    while time.time() - start < duration and shared_state.capture_active:
        # Step 4a: Check if tshark process ended unexpectedly
        if shared_state.tshark_proc.poll() is not None:
            break

        # Step 4b: Read a line of tshark output
        line = shared_state.tshark_proc.stdout.readline()

        # Step 4c: Split line by separator and parse packet
        parts = line.strip().split("|")
        packet = parse_and_store_packet(parts)

        # Step 4d: Store formatted packet for display
        shared_state.all_packets_history.append(packet)

        # Step 4e: Identify protocol and assign to stream group
        ip_proto = parts[15] or "N/A"
        proto = parts[5] or "N/A"
        tcp_stream = parts[7] or "N/A"
        udp_stream = parts[8] or "N/A"
        rtp_ssrc = parts[13] or "N/A"
        proto_name = protocol_map.get(ip_proto)
```

```
    key = None
    if (proto_name == "tcp" or proto == "tcp") and tcp_stream != "N/A":
        key = ("tcp", tcp_stream)
    elif proto_name == "udp" and udp_stream != "N/A":
        key = ("udp", udp_stream)
    elif "RTP" in proto.upper() and rtp_ssrc != "N/A":
        key = ("rtp", rtp_ssrc)
    else:
        key = ("other", "misc")

    if key not in shared_state.streams:
        shared_state.streams[key] = []
    shared_state.streams[key].append(parts)
```

---

function stop_tshark():

```
  # Step 1: Check if tshark process exists
  if shared_state.tshark_proc:
    # Step 2: Terminate tshark process gracefully
    shared_state.tshark_proc.terminate()

    # Step 3: Reset tshark handle and capture flag
    shared_state.tshark_proc = None
    shared_state.capture_active = False

    # Step 4: Clear stored streams, packets and metrics
    shared_state.streams = {}
    shared_state.all_packets_history = []
    # ...reset other metrics...

    # Step 5: Return success status
    return True, "Tshark stopped successfully"
  else:
    return False, "Tshark was not running"
```

```
function clear_all_packets():
   # Step 1: Clear streams dictionary
   shared_state.streams = {}
   # Step 2: Clear packet history list
   shared_state.all_packets_history = []

   # Step 3: Print confirmation message
   print("All packets cleared")
```

```
function get_formatted_packets(display_count):
   # Step 1: Return last 'display_count' packets from history
   return shared_state.all_packets_history[-display_count:]
```

```
function is_capture_active():
   # Step 1: Return capture active status
   return shared_state.capture_active
```

```
function get_streams():
   # Step 1: Return current packet streams dictionary
   return shared_state.streams
```

6.2 shared_state.py module

PURPOSE :

- Global storage module.
- Acts as the central hub for all application data.

ELEMENTS OF shared_state MODULE:

1. *streams* : A dictionary used to group captured packets by their network streams (e.g., TCP, UDP).
Data Structure : Dict[Tuple[str, str] , List[List [str]]]

```
{
("tcp", "1"): [
["1", "1695560445.123456", "192.168.0.2", "192.168.0.1", "60", "TCP", "SYN", "1", "", "", "", "",
"", "", "6", "", ""],
...
],
("udp", "0"): [ [...] ],
("rtp", "1234"): [ [...]
}
```

2. *protocol_distribution* : A dictionary that stores the packet counts of each protocol.
Data Structure : Dict

```
{
   "TCP": 0,
   "UDP": 0,
   "RTP": 0,
   "TLSV": 0,
   "QUIC": 0,
   "DNS": 0,
   "Others": 0
}
```

3. ***metrics_state :*** A dictionary that stores the calculated metrics, such as throughput, latency, and packet loss.

Data Structure : Dict

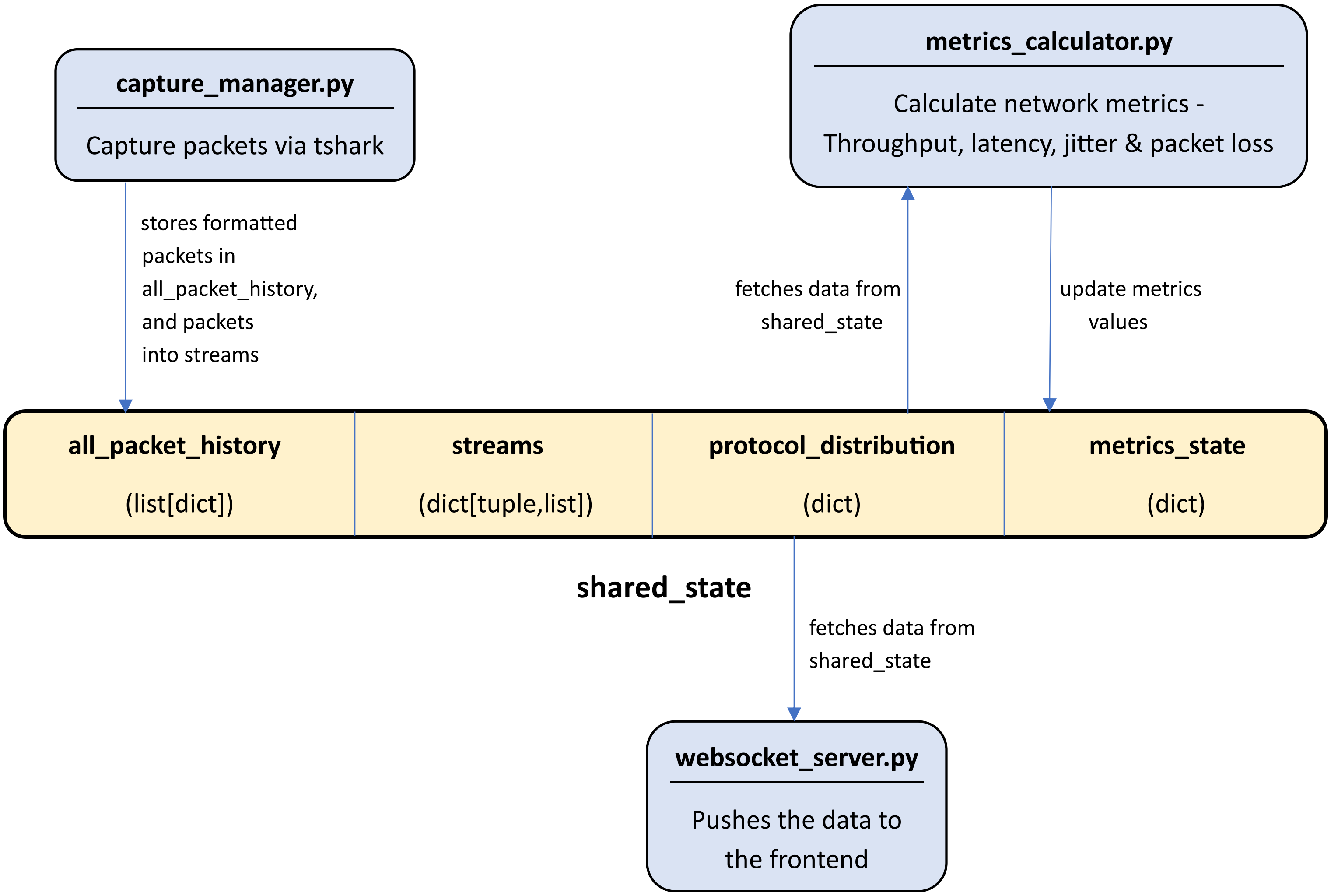
```
{
   "inbound_throughput": 0.0,
   "outbound_throughput": 0.0,
   "latency": 0.0,
   "jitter": 0.0,
   "packet_loss_count": 0,
   "packet_loss_percent": 0.0,
   "status": "stopped",
   "last_update": None,
   "protocol_distribution": protocol_distribution,
   "streamCount": 0,
   "totalPackets": 0}
```

4.*all_packets_history :* A list that stores all formatted packets captured during a session.

Data Structure : List[Dict]

```
[
{
"no": "1",
"time": "12:30:01.123",
"source": "192.168.0.1",
"destination": "192.168.0.2",
"protocol": "TCP",
"length": "60",
"info": "SYN, ACK"
},
...
]
```

FLOW DIAGRAM :



6.3 metrics_calculator.py module

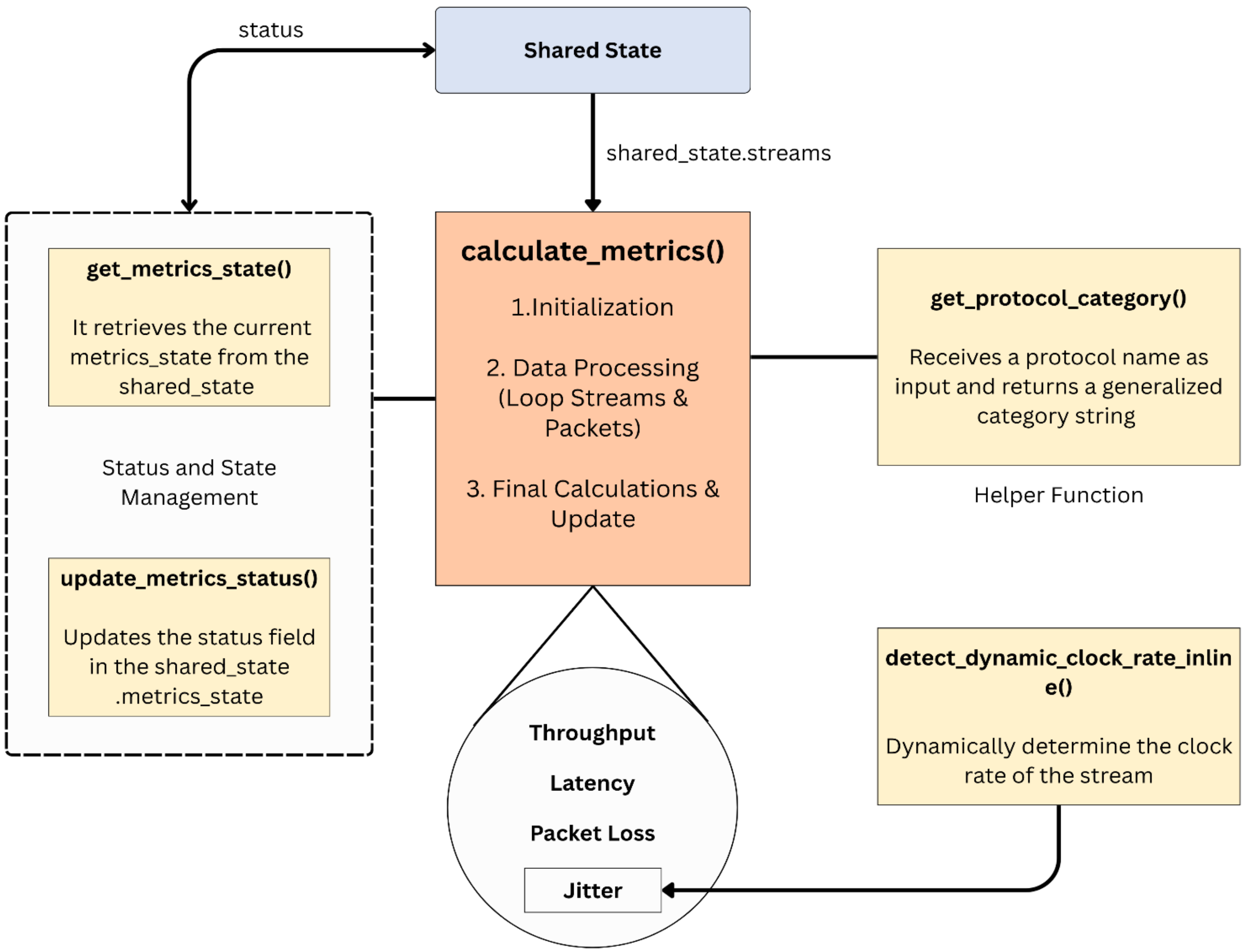
PURPOSE:

- Computing key network performance metrics by analyzing live packet streams.
- Calculating throughput, latency, packet loss, and jitter.
- Updating the shared_state with these metrics for real-time display.

CORE FUNCTIONS:

- **calculate_metrics()**: This function is the primary entry point for metric calculation. It retrieves packet data from the shared state, computes all the metrics, and updates the metrics_state dictionary.
- **update_metrics_status()**: A utility function to update the status field within the metrics_state (e.g., to "running" or "stopped").
- **get_metrics_state()**: Provides a safe, read-only copy of the current metrics_state to prevent accidental modification by other modules.
- **get_protocol_category()**: A helper function that categorizes a protocol name into a higher-level category (e.g., TCP, UDP, RTP).
- **detect_dynamic_clock_rate_inline()**: A helper function to dynamically detect the clock rate of an RTP stream based on time and timestamp differences, which is essential for accurate jitter calculation.

FLOW DIAGRAM:



PSEUDO CODE:

 # This function classifies a protocol name into a general category

```
def get_protocol_category(protocol_name):
   if not protocol_name or protocol_name == "N/A":
      return "OTHERS"
   protocol = protocol_name.upper().strip()

   if protocol == "TCP":
      return "TCP"
   elif protocol == "UDP":
      return "UDP"
   elif protocol == "RTP" or protocol == "SRTP":
      return "RTP"
   elif protocol == "QUIC":
      return "QUIC"
   elif protocol == "DNS":
      return "DNS"
   elif "TLSV" in protocol:
      return "TLSV"
   else:
      return "Others"
```

```
# This helper function detects the RTP clock rate from a packet stream
def detect_dynamic_clock_rate_inline(stream_packets):
   STATIC_PAYLOAD_RATES = {
      0: 8000, 3: 8000, 4: 8000, 5: 8000, 6: 16000, 7: 8000, 8: 8000, 9: 8000,
      10: 44100, 11: 44100, 12: 8000, 13: 8000, 14: 90000, 15: 8000,
      16: 11025, 17: 22050, 18: 8000, 26: 90000, 31: 90000, 32: 90000,
      33: 90000, 34: 90000,
   }

   if len(stream_packets) < 2:
      return None
   for i in range(1, min(5, len(stream_packets))):
      pkt1, pkt2 = stream_packets[i-1], stream_packets[i]
      time_diff = pkt2['arrival'] - pkt1['arrival']
```

```
        rtp_diff = pkt2['rtp_ts'] - pkt1['rtp_ts']
        if time_diff > 0 and rtp_diff > 0:
            calculated_rate = rtp_diff / time_diff
            known_rates = [8000, 16000, 22050, 44100, 48000, 90000]
            for known_rate in known_rates:
                if abs(calculated_rate - known_rate) / known_rate < 0.15:
                    return known_rate
            return 8000 if calculated_rate < 20000 else 90000
    return None
```

```
# Calculates network performance metrics from streams
def calculate_metrics():
    # 1. Initialize variables for all metrics (throughput, latency, etc.) to zero.
    # 2. Check for an empty stream list; if empty, return default zero values.
    # 3. Iterate through each stream and each packet within that stream.
    # 4. Inside the loop, perform calculations for each of the four metrics.
    # 5. After the loop, perform final calculations (e.g., averages, percentages).
    # 6. Update the 'metrics_state' in shared_state.py with the final values.
    # 7. Return the updated metrics state.
```

Metric 1: Throughput

```
# Logic for calculating inbound and outbound throughput.
inbound_bytes = 0
outbound_bytes = 0
start_time = float("inf")
end_time = 0

# Loop through packets
for packet in stream:
    length = packet.length
    time_rel = packet.time_epoch

    # Check if the packet's source IP is a known local address
    if packet.source_ip is a local address:
```

```
        outbound_bytes += length
    # Check if the packet's destination IP is a known local address
    elif packet.destination_ip is a local address:
        inbound_bytes += length

    # Update the start and end times for the duration calculation
    start_time = min(start_time, time_rel)
    end_time = max(end_time, time_rel)
# After the loop, calculate the final throughput
duration = max(end_time - start_time, 1.5)  # Use a minimum duration to prevent spikes
in_throughput_mbps = (inbound_bytes * 8) / duration / 1e6
out_throughput_mbps = (outbound_bytes * 8) / duration / 1e6
```

Metric 2: Latency

```
# Logic for calculating the weighted average Round-Trip Time (RTT) for TCP.
total_weighted_latency = 0
total_weight = 0

# Loop through streams
for stream in all_streams:
    if stream.protocol is "TCP":
        stream_rtt_sum = 0
        stream_rtt_count = 0

        # Loop through packets in a TCP stream
        for packet in stream.packets:
            rtt = packet.ack_rtt # RTT from the TCP acknowledgment
            if rtt is valid:
                stream_rtt_sum += rtt * 1000  # Convert to milliseconds
                stream_rtt_count += 1

        if stream_rtt_count > 0:
            stream_avg_rtt = stream_rtt_sum / stream_rtt_count
            stream_weight = stream.packets.length
            total_weighted_latency += stream_avg_rtt * stream_weight
```

```
        total_weight += stream_weight

# After the loop, calculate the final weighted average latency
latency_ms = total_weighted_latency / total_weight if total_weight > 0 else 0.0
```

Metric 3: Packet Loss

```
# Logic for counting packet loss for TCP and RTP.
total_tcp_retransmissions = 0
total_rtp_loss = 0
expected_packets_total = 0 # Cumulative total for loss percentage

# Loop through streams
for stream in all_streams:
  if stream.protocol is "TCP":
    # Loop through packets in a TCP stream
    for packet in stream.packets:
      # Check for TCP retransmission flags
      if packet is a retransmission and not a spurious one:
        total_tcp_retransmissions += 1
    expected_packets_total += stream.packets.length

  elif stream.protocol is "RTP":
    last_seq = None
    # Loop through packets in an RTP stream
    for packet in stream.packets:
      seq = packet.sequence_number
      # Check for a gap in sequence numbers
      if seq > last_seq:
        gap = seq - last_seq - 1
        if gap > 0:
          total_rtp_loss += gap
      last_seq = seq
    expected_packets_total += stream.packets.length

# After the loop, calculate the final cumulative loss and percentage
```

total_loss_this_batch = total_tcp_retransmissions + total_rtp_loss
cumulative_loss = previous_loss + total_loss_this_batch
loss_percentage = (cumulative_loss / expected_packets_total) * 100

Metric 4: Jitter

```
# Logic for calculating the weighted average jitter for RTP streams.
total_weighted_jitter = 0
total_jitter_weight = 0

# Loop through streams
for stream in all_streams:
    if stream.protocol is "RTP":
        jitter = 0
        prev_transit_time = None
        clock_rate = detect_dynamic_clock_rate_inline(stream.packets)

        # Loop through packets in an RTP stream
        for packet in stream.packets:
            rtp_ts = packet.rtp_timestamp
            arrival_time = packet.time_epoch

            # Calculate transit time and the difference from the previous packet
            transit_time = (arrival_time * clock_rate) - rtp_ts
            if prev_transit_time is not None:
                d = abs(transit_time - prev_transit_time)
                # Apply the RFC 3550 jitter formula
                jitter = jitter + (d - jitter) / 16
            prev_transit_time = transit_time

        # Calculate this stream's weighted jitter
        stream_weight = stream.packets.length
        total_weighted_jitter += jitter * stream_weight
        total_jitter_weight += stream_weight
# After the loop, calculate the final weighted average jitter
jitter = total_weighted_jitter / total_jitter_weight if total_jitter_weight > 0 else 0.0
```

 # Update the status in metrics state
def update_metrics_status(status):
   shared_state.metrics_state["status"] = status

 # Get current metrics state
def get_metrics_state():
   return shared_state.metrics_state.copy()

Input and Output Streams :

- **Inputs**: The module's primary input stream is the shared_state.streams dictionary, which contains raw packet data grouped by protocol and stream ID. It also uses shared_state.ip_address to determine the traffic direction for throughput calculations.
- **Outputs**: The module writes the final, calculated metrics to the shared_state.metrics_state dictionary. The output for each metric is a single, aggregated value for the batch.

Optimization Strategies :

- **Single Pass Calculation**: The module iterates through the packet stream only once to compute all four metrics simultaneously. This avoids redundant loops and improves performance.
- **Weighted Averages**: For latency and jitter, it uses a weighted average (weighted by packet count) to ensure that metrics are not skewed by short or inactive streams.
- **Dynamic Clock Rate Detection**: A helper function is used to dynamically detect the clock rate of an RTP stream, which is critical for accurate jitter calculations. This makes the calculation flexible and robust.

Algorithms :

<u>Metric 1: Throughput</u>

- **Initialization**: inbound_bytes = 0, outbound_bytes = 0, start_time = infinity, end_time = 0.
- **Packet Loop**:
    - **Check**: If the packet's source IP is a known local address, increment outbound_bytes.
    - **Check**: If the packet's destination IP is a known local address, increment inbound_bytes.
    - **Update**: start_time is updated to the minimum time_rel.
    - **Update**: end_time is updated to the maximum time_rel.
- **Final Calculation**:
    - **Duration**: Calculated as max(end_time - start_time, 1.5). The 1.5 acts as a **Fallback condition** to prevent spikes from single-packet captures.
    - **Final Value**: in_throughput_mbps and out_throughput_mbps are calculated by converting bytes to megabits and dividing by duration.

<u>Metric 2: Latency</u>

- **Initialization**: total_weighted_latency = 0, total_weight = 0.
- **Stream Loop**:
    - **Condition**: Process only streams where stream.protocol is "TCP".
    - **Inner Loop**: Sum the packet.ack_rtt values, converting them to milliseconds.
    - **Calculation**: If RTT values are found, calculate stream_avg_rtt.

    - **Weighted Average**: Add stream_avg_rtt multiplied by stream.packets.length to total_weighted_latency. Add stream.packets.length to total_weight.
- **Final Value**: latency_ms is calculated as total_weighted_latency / total_weight with a zero check.

Metric 3: Packet Loss

- **Initialization**: total_tcp_retransmissions = 0, total_rtp_loss = 0.
- **Stream Loop**:
    - **Condition**: If stream.protocol is "TCP":
        - **Check**: Loop through packets and increment total_tcp_retransmissions if a packet is flagged as a retransmission and not a spurious one.
    - **Condition**: If stream.protocol is "RTP":
        - **Check**: Loop through packets, comparing the current packet.sequence_number to the last_seq to detect a gap.
        - **Update**: If a gap is found, increment total_rtp_loss.
- **Final Calculation**:
    - total_loss_this_batch is the sum of total_tcp_retransmissions and total_rtp_loss.
    - cumulative_loss is previous_loss + total_loss_this_batch.

Metric 4: Jitter

- **Initialization**: total_weighted_jitter = 0, total_jitter_weight = 0.
- **Stream Loop**:
    - **Condition**: Process only streams where stream.protocol is "RTP".

- **Inner Loop**:
  - **Initialization**: jitter = 0, prev_transit_time = None.
  - **Dynamic Clock Rate**: The clock_rate is determined by calling detect_dynamic_clock_rate_inline(stream.packets).
  - **Calculation**: transit_time is derived from the packet.arrival_time and packet.rtp_timestamp using the clock_rate.
  - **RFC Filter**: If prev_transit_time exists, calculate d as the absolute difference.
  - **Formula**: Apply the RFC 3550 jitter formula: jitter = jitter + (d - jitter) / 16.
  - **Update**: prev_transit_time is updated to the current transit_time.
- **Weighted Average**: Add jitter multiplied by stream.packets.length to total_weighted_jitter. Add stream.packets.length to total_jitter_weight.

- **Final Value**: jitter is calculated as total_weighted_jitter / total_jitter_weight with a zero check.

6.4 websocket_server.py module

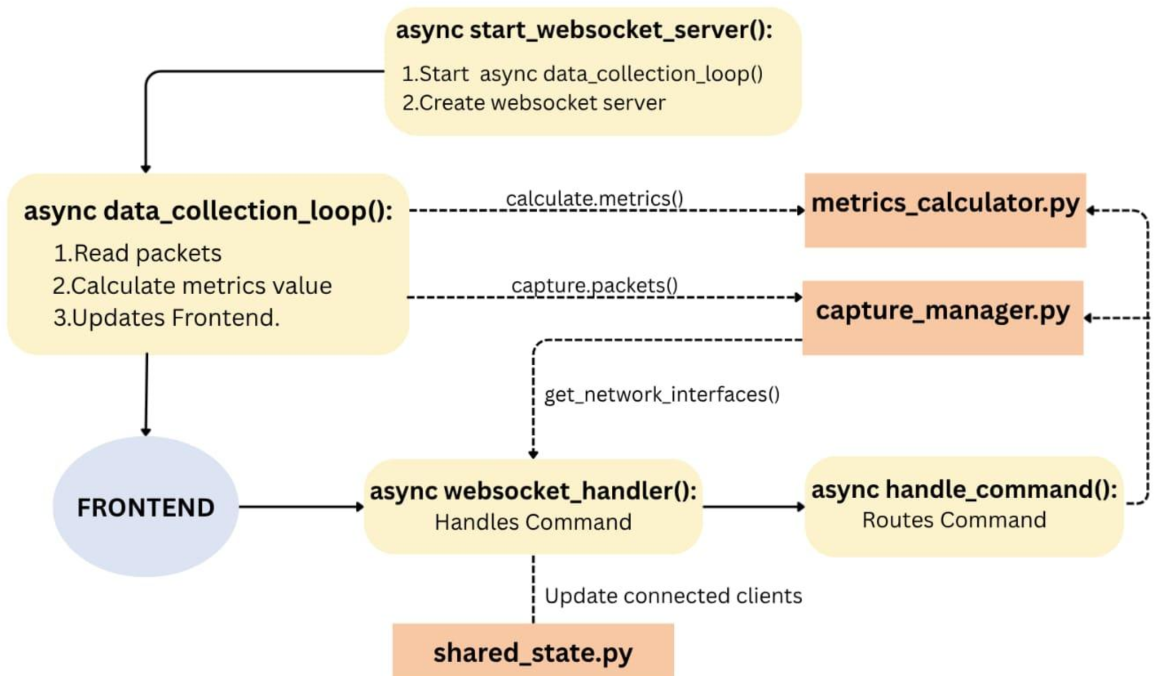
PURPOSE:

- Acts as the bridge between the backend packet capture system and the frontend dashboard.
- Ensures continuous real-time updates ,manages client communication and data flow.
- It is responsible for sending live network metrics (throughput, latency, jitter, packet loss, etc.) to the dashboard as soon as they are calculated.
- Receiving commands from the client (like start capture, stop capture, get interfaces)

CORE FUNCTIONS:

- **async start_websocket_server():** This is the top-level function that starts the data_collection_loop() as a concurrent async task and then creates the WebSocket server to listen for client connections.
- **async data_collection_loop()**: An asynchronous loop that runs continuously. It is responsible for:
  a) Calling capture_packets() to read from tshark's output.
  b) Invoking calculate_metrics() to compute the network metrics.
  c) Updating the frontend dashboard by sending real-time data.
- **async handle_command()**: A function that proceses incoming WebSocket commands, such as "start_capture" and "stop_capture." It routes these commands to the appropriate functions in other modules.
- **async websocket_handler()**: This function manages the lifecycle of a single connected client, from handling their incoming messages to managing disconnections.

FLOW DIAGRAM:



PSEUDOCODE:

1. async def start_websocket_server():

#Start the data collection loop

```
asyncio.create_task(data_collection_loop())
```

#Start the server

```
server = await websockets.serve(websocket_handler, "localhost", 8765)
```

#Wait until server is closed

```
    await server.wait_closed()
```

2. async def websocket_handler(websocket):

```
  # On connection, register the client.
    shared_state.connected_clients[websocket] =
{"connected_at":datetime.now().isoformat()}
 # Send initial state.
    await websocket.send(json.dumps(data_to_send))
# Receive and process commands in a loop.
   async for message in websocket: data = json.loads(message);
  response = await handle_command(data.get("command"), data);
  await websocket.send(json.dumps(response));
# Handle disconnection.
  finally: del shared_state.connected_clients[websocket]
```

3. async def handle_command(command, data):

```
# Get network interfaces
if command == "get_interfaces":
return
{"type":"interfaces_response","interfaces":capture_manager.get_network_interfaces()}
# Start packet capture
elif command == "start_capture": capture_manager.clear_all_packets();
msg=capture_manager.start_tshark(data.get("interface","Wi-Fi"));
metrics_calculator.update_metrics_status("running" if success else "stopped"); return
{"type":"command_response","command":"start_capture","success":success,"message"
:msg}
```

Stop packet capture

elif command == "stop_capture": success,msg=capture_manager.stop_tshark(); metrics_calculator.update_metrics_status("stopped" if success else "running"); return {"type":"command_response","command":"stop_capture","success":success,"message":msg}

Get current metrics status

elif command == "get_status": return{"type":"status_response","metrics":metrics_calculator.get_metrics_state()}

Handle unknown commands

else: return {"type":"error","message":f"Unknown command: {command}"}

4. async def data_collection_loop():

Sleep for a short time to control update frequency

await asyncio.sleep(0.1)

Skip if no capture is active or no clients connected

if not capture_manager.is_capture_active() or not shared_state.connected_clients: continue

Capture packets and calculate metrics

capture_manager.capture_packets(3); metrics_calculator.calculate_metrics()

Get packet statistics

stats = capture_manager.get_packet_statistics()

Prepare to track disconnected clients

disconnected_clients = set()

Send updates to all connected clients

for client in list(shared_state.connected_clients.keys()): await client.send(json.dumps({"type":"update","metrics":metrics_calculator.get_metrics_state(),"new_packets":shared_state.all_packets_history,"stream_count":stats["stream_count"]}))

Remove disconnected clients

for client in disconnected_clients: del shared_state.connected_clients[client]

6.5 main.py module

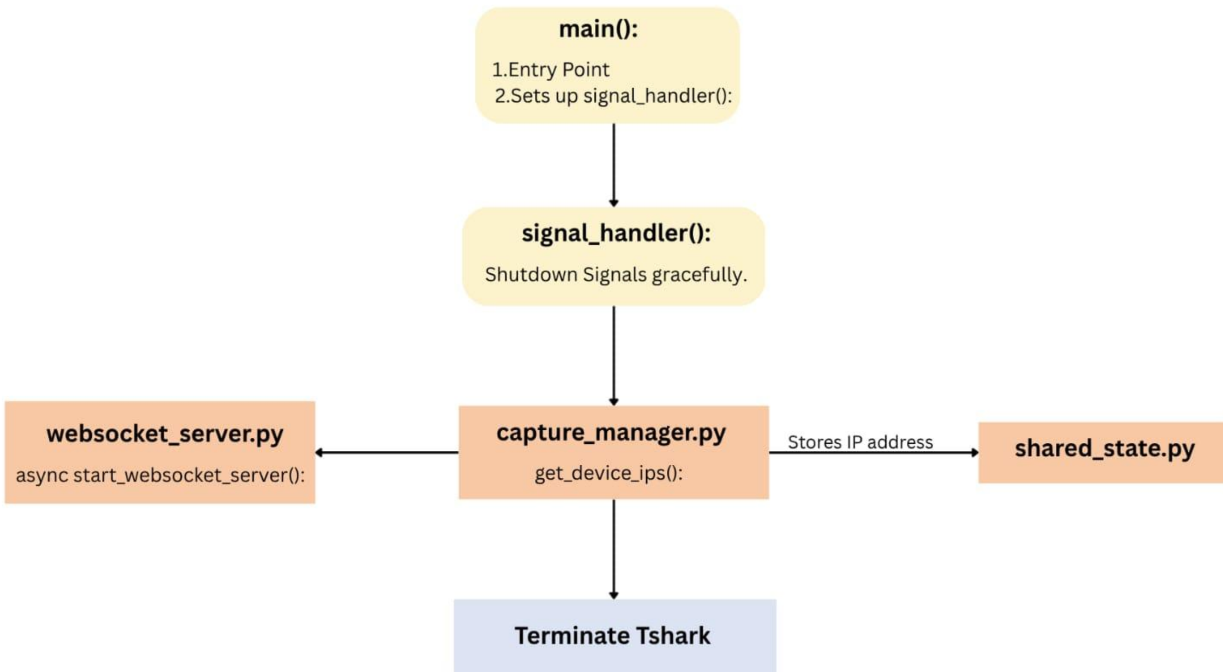
PURPOSE:

- Sets Up Graceful Shutdown.
- It serves as the application's entry point and handles its overall lifecycle.
- Handles Errors and Interruptions, and also guarantees resource cleanup.

CORE FUNCTIONS:

- **signal_handler()**: A function responsible for gracefully shutting down the application when a signal is received (e.g., Ctrl+C). It calls stop_tshark() to terminate the subprocess cleanly before the program exits.
- **main()**: The primary function that sets up signal handlers and starts the WebSocket server. It uses asyncio.run() to start the server and a try...finally block to ensure that stop_tshark() is always called for resource cleanup.

FLOW DIAGRAM:



PSEUDOCODE:

1. **def signal_handler(sig, frame):**
   # It stops tshark and print a shutdown message when signal is received
       capture_manager.stop_tshark()
       PRINT: “Received shutdown signal. Cleaning up..."

   #Sets Packet Capture process inactive.
      shared_state.capture_active = False

   #Exits gracefully when shutdown signal is received without raising an exception.
      os.exit(0)

2. **def main():**

#Set up signal handlers for graceful shutdown and print application start message.

```
signal.signal(signal.SIGINT, signal_handler)
signal.signal(signal.SIGTERM, signal_handler)
PRINT: "Network Monitoring Dashboard - Backend"
```

#Start the WebSocket server(async event loop) inside a try block.

```
asyncio.run(start_websocket_server())
```

#If user interrupts with Ctrl+C, print interruption message and if any other error occurs,   print the error message.

```
except KeyboardInterrupt:
PRINT: Application interrupted by user
except Exception as e:
PRINT: Application error {e}
```

#In the finally block, always stop tshark and print shutdown complete.

```
capture_manager.stop_tshark()
PRINT:"Application shutdown complete"
```

#Check if this script is the main program and call main()

```
if __name__ == "__main__":
main()
```

7. Data Structures :

Name	Type
all_packets_history	List of dicts
streams	Dict with tuple keys
metrics_state	Dict
packet_data	Dict
capture_active	Bool
protocol_distribution	Dict

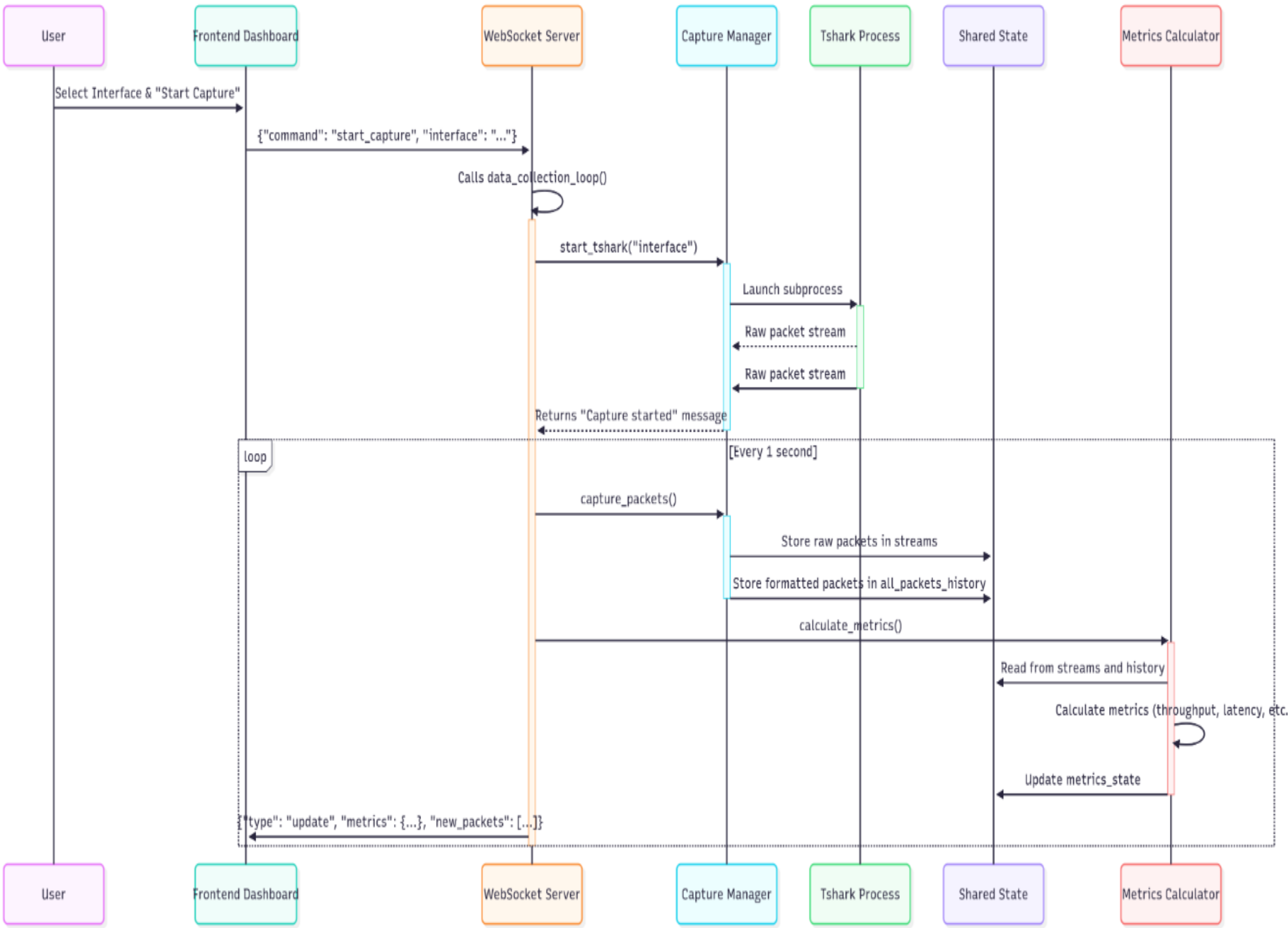
---

8. Data Transfer between Source and Destination :

Source	Data	Destination	Description
User	Commands (start/stop/etc.)	Frontend Dashboard	User triggers capture, commands
Frontend Dashboard	Commands	WebSocket Server	Sends commands over WS
WebSocket Server	Command data	Capture Manager	Starts/stops capture
Capture Manager	Packet data	Shared State	Saves captured packets
tshark	Captured packets	Capture Manager	External packet capture
Metrics Calculator	Reads packets from Shared State	Shared State	Updates metrics in shared state
WebSocket Server	Metrics + packets data	Frontend Dashboard	Sends delta updates over WS
Frontend Dashboard	Displays updated metrics/graphs	User	Visualizes real-time data

---

9. Sequence Diagram :



10. WebSocket API Specification :

Command	Request Payload	Response Payload	Description
get_interfaces	{ "command": "get_interfaces" }	{ "type": "interfaces_response", "interfaces": [...] }	List all available interfaces
start_capture	{ "command": "start_capture", "interface": "WiFi" }	{ "type": "command_response", "command": "start_capture", "success": true, "message": "Capture started" }	Start capturing on interface
stop_capture	{ "command": "stop_capture" }	{ "type": "command_response", "command": "stop_capture", "success": true, "message": "Capture stopped" }	Stop capturing
get_status	{ "command": "get_status" }	{ "type": "status_response", "metrics": {...} }	Get current network metrics
updates (push)	None (server-initiated)	{ "type": "update", "metrics": {...}, "new_packets": [...], "stream_count": n }	Periodic data updates pushed to clients

---

11. Error Handling and Recovery :

- **Client Disconnect:** Server detects connection closure and removes client from registry.
- **Capture Errors:** tshark start/stop failures are reported to clients with error messages.
- **Signal Handling:** OS signals (SIGINT, SIGTERM) gracefully stop capture and exit application.
- **Invalid Commands:** Server responds with error message JSON for unknown commands.

---

12. Frontend Dashboard Overview:

The backend is paired with a Frontend Dashboard application designed to provide network operators with an intuitive and interactive user interface for monitoring network health in real time.

Key Features of the Frontend Dashboard:

- **Real-time Graphs:**
  - **Throughput Graph:** Shows bytes transmitted over time, helping identify bandwidth bottlenecks.
  - **Latency and Jitter Visualization:** Line charts track delay and its variation across time, highlighting network performance stability.
  - **Packet Loss Monitoring:** Display packet loss trends to quickly spot network reliability issues.
- **Protocol Distribution Chart:**
  - Pie or bar charts display the proportion of captured packets by protocol (TCP, UDP, RTP, etc.) enabling quick insight into traffic composition.
- **Live Packet Stream Viewer:**
  - A scrollable, real-time feed of captured packets with metadata, timestamps, and source/destination info.
- **User Interaction:**
  - Ability to start and stop captures on selected network interfaces directly from the dashboard.
  - Display current capture status and errors if any.

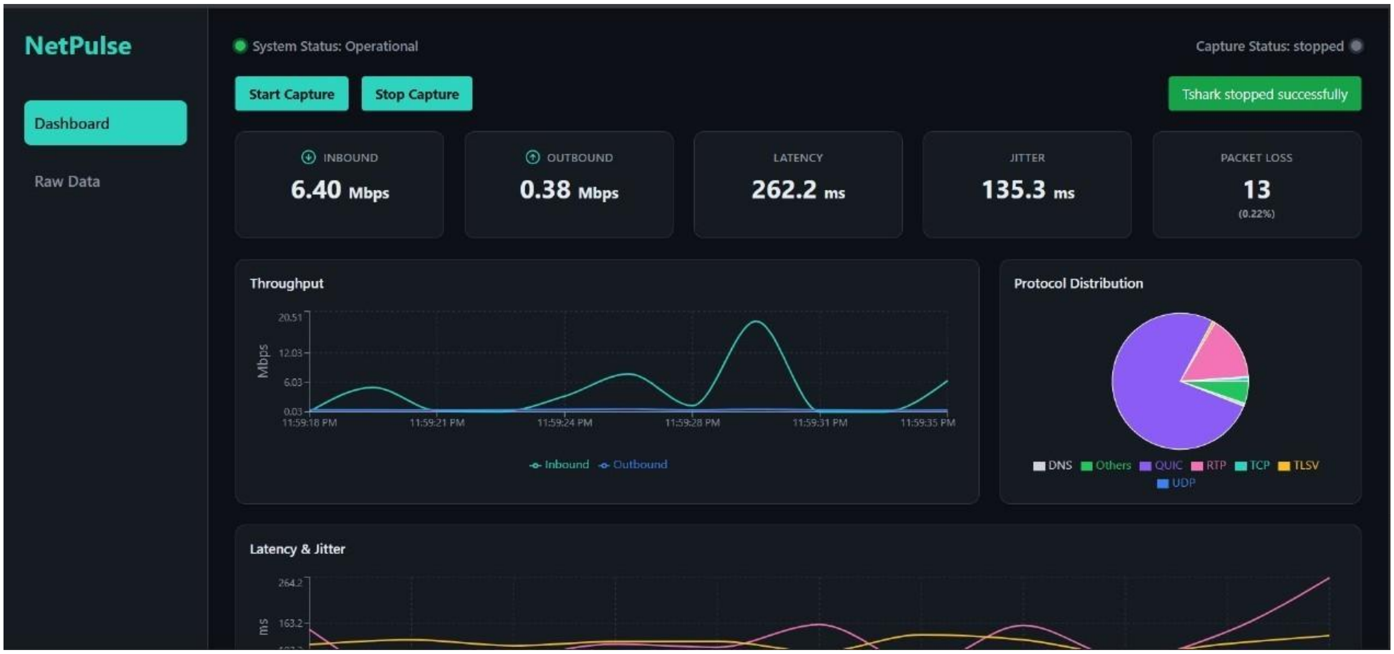
Communication between Frontend and Backend:

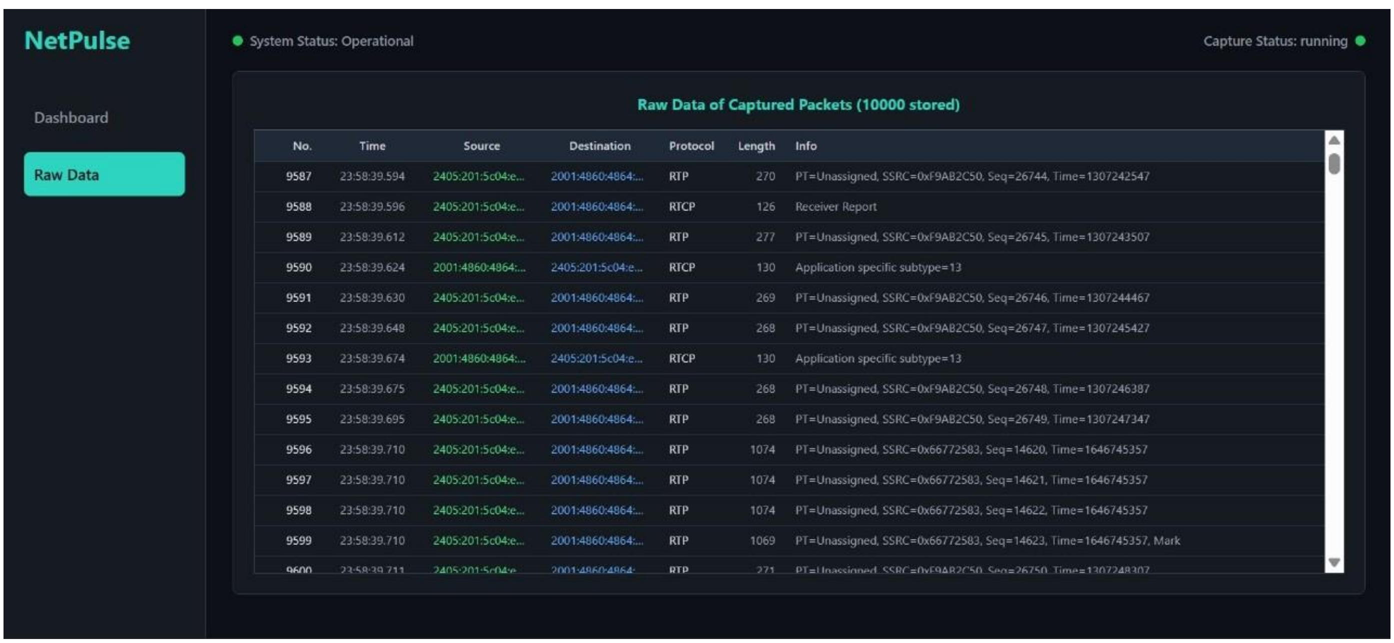
- Utilizes **WebSocket** connections to maintain a persistent, low-latency channel for streaming metrics and control commands.
- Frontend sends commands (start_capture, stop_capture, etc.) and listens for structured JSON updates.
- The backend efficiently batches and pushes delta updates to minimize bandwidth and improve responsiveness.

13. Glossary :

Term	Definition
tshark	Command-line network protocol analyzer tool (part of Wireshark) used for packet capture and analysis.
Latency	Time delay between sending and receiving packets, typically measured in milliseconds.
Jitter	Variation in packet delay over time, affecting quality in real-time communications.
Packet Loss	Percentage of packets lost during transmission, indicating network reliability issues.
WebSocket	Protocol providing full-duplex communication channels over a single TCP connection for real-time data exchange.
Recharts	A React-based charting library built on D3.js, used in the frontend dashboard to visualize metrics such as throughput, latency, and protocol distribution through interactive charts.

15.Snap-Shots :


NetPulse
Dashboard
Raw Data
System Status: Operational
Capture Status: stopped
Start Capture
Stop Capture
Tshark stopped successfully
INBOUND
6.40 Mbps
OUTBOUND
0.38 Mbps
LATENCY
262.2 ms
JITTER
135.3 ms
PACKET LOSS
13
(0.22%)
Throughput
Mbps
20.51
12.03
6.03
0.03
11:59:18 PM
11:59:21 PM
11:59:24 PM
11:59:28 PM
11:59:31 PM
11:59:35 PM
Inbound
Outbound
Protocol Distribution
DNS
Others
QUIC
RTP
TCP
TLSV
UDP
Latency & Jitter
264.2
163.2
ms


NetPulse
Dashboard
Raw Data
System Status: Operational
Capture Status: running
Raw Data of Captured Packets (10000 stored)
No. Time Source Destination Protocol Length Info
9587 23:58:39.594 2405:201:5c04:e... 2001:4860:4864:... RTP 270 PT=Unassigned, SSRC=0xF9AB2C50, Seq=26744, Time=1307242547
9588 23:58:39.596 2405:201:5c04:e... 2001:4860:4864:... RTCP 126 Receiver Report
9589 23:58:39.612 2405:201:5c04:e... 2001:4860:4864:... RTP 277 PT=Unassigned, SSRC=0xF9AB2C50, Seq=26745, Time=1307243507
9590 23:58:39.624 2001:4860:4864:... 2405:201:5c04:e... RTCP 130 Application specific subtype=13
9591 23:58:39.630 2405:201:5c04:e... 2001:4860:4864:... RTP 269 PT=Unassigned, SSRC=0xF9AB2C50, Seq=26746, Time=1307244467
9592 23:58:39.648 2405:201:5c04:e... 2001:4860:4864:... RTP 268 PT=Unassigned, SSRC=0xF9AB2C50, Seq=26747, Time=1307245427
9593 23:58:39.674 2001:4860:4864:... 2405:201:5c04:e... RTCP 130 Application specific subtype=13
9594 23:58:39.675 2405:201:5c04:e... 2001:4860:4864:... RTP 268 PT=Unassigned, SSRC=0xF9AB2C50, Seq=26748, Time=1307246387
9595 23:58:39.695 2405:201:5c04:e... 2001:4860:4864:... RTP 268 PT=Unassigned, SSRC=0xF9AB2C50, Seq=26749, Time=1307247347
9596 23:58:39.710 2405:201:5c04:e... 2001:4860:4864:... RTP 1074 PT=Unassigned, SSRC=0x66772583, Seq=14620, Time=1646745357
9597 23:58:39.710 2405:201:5c04:e... 2001:4860:4864:... RTP 1074 PT=Unassigned, SSRC=0x66772583, Seq=14621, Time=1646745357
9598 23:58:39.710 2405:201:5c04:e... 2001:4860:4864:... RTP 1074 PT=Unassigned, SSRC=0x66772583, Seq=14622, Time=1646745357
9599 23:58:39.710 2405:201:5c04:e... 2001:4860:4864:... RTP 1069 PT=Unassigned, SSRC=0x66772583, Seq=14623, Time=1646745357, Mark