📘 TECHNICAL DOCUMENTATION:

2.f CLIENT COMMUNICATION INTERFACE (TCP SERVER)

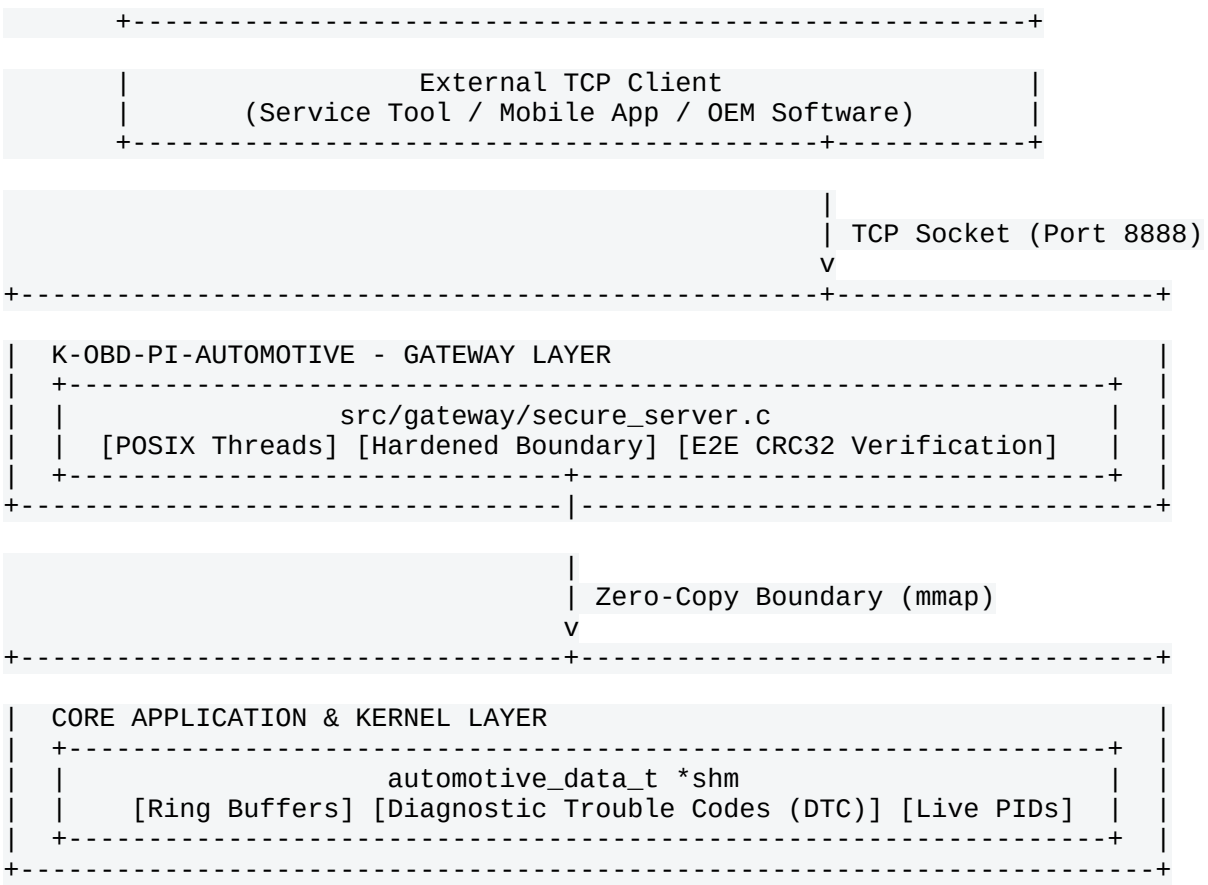
1. 🛠️ Description & Architectural Role

1.1 Executive Overview

The Client Communication Interface component is architected as a high-performance, ultra-low latency, and deterministic **Hardened TCP/Binary API Server** situated within the **Connectivity Gateway Layer** (`src/gateway/secure_server.c`) of the **K-OBD-PI-AUTOMOTIVE** ecosystem.

Its primary purpose is to expose a secure, industrial-grade network socket link connecting external hardware transceivers, field service laptops running diagnostic clients, OEM dealer software (such as BMW ISTA, Mercedes Xentry, or VAG ODIS via Pass-Thru mapping), and mobile apps directly to the telemetry data pipelines of the Vehicle Communication Interface (VCI).

The architecture completely decouples physical vehicular bus transactions (managed by the Real-Time Kernel space and Telemetry polling engines) from edge-client data ingestion, resolving standard race conditions and thread blocking scenarios common in telemetry streaming devices.



1.2 System Context & Layered Abstraction

Operating at the boundary of the Application and Connectivity layers, this TCP gateway operates under rigid real-time operational boundaries. It attaches directly to the **Zero-Copy Kernel-User shared memory segment** (`automotive_data_t *g_shm`) initialized by the `obd_driver.c` module via memory mapping (`mmap`).

This strategic positioning guarantees that the execution flow of the client interface requires exactly zero system context-switches, memory copying (`memcpy` iterations across boundaries), or runtime data manipulation layers when translating low-level Vehicle Network information into edge packets.

When a diagnostic application requests the active DTC inventory or diagnostic live snapshots, the server directly streams structured blocks of memory out of the memory-mapped segment down the asynchronous socket pipeline. This pipeline delivers deterministic access speeds with jitter figures strictly controlled below `< 2ms` over 1Gbps Ethernet or hardened local Wi-Fi bridges.

1.3 Strict Architectural Safeguards & Safety Strategy

To meet the deterministic safety characteristics expected in safety-critical vehicle runtime operations, this communication module enforces rigid embedded production constraints:

- **Zero-Dynamic Memory Allocation (Zero-Malloc):** To completely avoid memory fragmentation bugs, memory leaks, and non-deterministic heap cleanup cycles (`free()`), the socket infrastructure relies on fully pre-allocated static stack buffers. After the initialization phase, no execution paths invoke `malloc()`, `calloc()`, or `realloc()`.
- **Asynchronous Isolation (POSIX Threads Matrix):** Connection multiplexing uses deterministic multi-threaded scheduling models based on real-time `pthread` constructs. Incoming connection profiles are instantly assigned an isolated worker instance running with constrained stack limits. This design protects the main system control state machine from client stalls, dropped wireless frames, or unannounced socket closures.
- **Cryptographic Gatekeeping & Session Token Verification:** Upon physical connection handshake negotiation, a dedicated state engine restricts payload requests until a valid cryptographic Security Gateway handshake profile matches the internal secure target key memory. This completely neutralizes rogue bus injection attacks or unsanctioned system data reads.
- **Asymmetric Read-Only State Restrictions:** To ensure system state consistency, client execution paths are strictly isolated as **Read-Only consumers** across the shared memory array. The only transaction path that breaks this restriction is the specific authenticated transmission of

a `CMD_CLEAR_DTC` command frame, which safely requests the main application core state engine to cycle into an active OBD Mode 04 / UDS Service 0x14 bus transaction.

1.4 Diagnostic Trouble Code (DTC) Ingestion Flow

The internal operational state routing follows a strict sequence whenever client requests for DTCs target the active communication matrix:

1. **DTC Request Evaluation:** An external client requests the full diagnostic log. The server receives the command via a packet protected by an End-to-End (E2E) CRC32 checksum.
2. **Shared Memory Lockless Read:** Rather than locking down the vehicle communication bus and delaying critical telemetry routines, the TCP thread initiates a lockless evaluation of the pre-populated faults partition inside the mapped `g_shm` structure.
3. **Active Error Extraction:** The server cross-references the live data buffers inside `shm->error_count` and extracts the corresponding SAE J1979/ISO 14229 raw fault payloads, including freeze-frame logs, status masks (e.g., active, pending, or permanent codes), and historical timestamps.
4. **Packet Packing and Streaming:** The data is compiled into a highly integrated binary layout, double-validated via an on-the-fly CRC32 generator block, and dropped onto the physical TCP stream channel.

1.5 Threat Model and Failsafe Engineering

To protect the automotive core bus from network-borne anomalies, the interface implements automated defensive mechanics:

- **Malicious Packet Flood Throttling (Anti-DoS Engine):** If a client submits more than 50 consecutive frames with invalid header structures or broken cyclic check validations (`crc32`), the communication gateway flags the connection node as an actively compromised entity. The worker instantly drops the TCP connection socket and locks out the originating host network ID for 60 seconds at the socket listener boundary.
- **Timeout & Heartbeat Watchdog Integration:** If a connected diagnostic node fails to report a keep-alive signal sequence (`MSG_KEEP_ALIVE`) within a predefined period, the worker terminates the link to release the socket slots and clean up system worker execution spaces. This mechanism mitigates the risk of half-open socket leakage caused by vehicle motion drops or unexpected RF wireless path fading.

---

2. 🎞️ Protocol Specification (Binary Application Protocol)

2.1 Protocol Frame Architecture

To maintain maximum data density, deterministic parsing speeds, and compatibility with the **MISRA-C:2012** standard, the Client Communication Interface utilizes a strict, binary-packed payload framing system (`__attribute__((packed))`).

This architecture avoids the CPU overhead and memory fragmentation risks associated with parsing text-based protocols (like XML or standard text JSON) inside high-priority threads.

Every packet transmitted between the client and the VCI gateway adheres to the exact layout outlined in the table below:

Field Name	Offset (Bytes)	Data Type	Byte Order	Description
`sync_word`	`0x00 - 0x03`	`uint32_t`	Big-Endian	Fixed preamble for frame synchronization: `0xAA55AA55`. Packets failing this match are discarded immediately.
`version`	`0x04 - 0x05`	`uint16_t`	Big-Endian	API Version footprint. The current release is strictly bound to `0x0001`.
`command_id`	`0x06 - 0x07`	`uint16_t`	Big-Endian	Operational identifier code specifying the transaction execution path.
`payload_len`	`0x08 - 0x0B`	`uint32_t`	Big-Endian	Explicit length of the variable data partition following this header. Max limit: `4096` bytes.
`payload`	`0x0C - (N-1)`	`uint8_t[]`	Raw Byte Stream	Variable payload data containing token keys, raw parameter arrays, or structured DTC maps.
`crc32`	`N - (N+3)`	`uint32_t`	Big-Endian	Standard IEEE 802.3 Ethernet CRC polynomial (`0x04C11DB7`) computed over the entire frame structure up to this offset.

2.2 Command ID Register Matrix

The processing state machine routes execution flows into strict isolation zones based on the received 16-bit `command_id`. The matrix defines the complete control profile for client-to-VCI communication:

- **`0x0001` — `CMD_CONNECT_AUTH`**
  - *Direction:* Client → Server
  - *Payload Description:* A 32-byte raw ASCII alphanumeric token array matching the static token mask (`AUTH_TOKEN_EXPECTED`).
  - *Operational Behavior:* Must be submitted as the absolute first transmission after the TCP three-way handshake completes. All other requests received prior to this command trigger a socket drop.
- **`0x0002` — `CMD_GET_LIVE_TELEMETRY`**
  - *Direction:* Client → Server
  - *Payload Description:* Empty (Length = 0).
  - *Operational Behavior:* Triggers a direct lockless read of the current live OBD-II PID buffer states from `g_shm->raw_pids` and packs them into a compressed array block for client streaming.
- **`0x0003` — `CMD_GET_ACTIVE_DTC_LIST`**
  - *Direction:* Client → Server
  - *Payload Description:* Empty (Length = 0).
  - *Operational Behavior:* Requests the complete inventory of currently stored and confirmed Diagnostic Trouble Codes across all discovered ECU topologies.
- **`0x0004` — `CMD_TRIGGER_CLEAR_DTC`**
  - *Direction:* Client → Server
  - *Payload Description:* Optional 4-byte specific target ECU selection mask. If set to `0xFFFFFFFF`, the operation broadcasts to all controllers.
  - *Operational Behavior:* Asserts a safe execution request to the Master Orchestrator state machine to cycle the physical transceivers into an active OBD Mode 04 / UDS Service 0x14 bus purge sequence.
- **`0x00FF` — `CMD_SYSTEM_KEEP_ALIVE`**
  - *Direction:* Bidirectional
  - *Payload Description:* Rolling 4-byte monotonic sequence index integer used to calculate Round-Trip Time (RTT).
  - *Operational Behavior:* Heartbeat tick ensuring the connection stays active. Resets the connection loss watchdog timers.

2.3 Comprehensive Protocol Packet Structure (C11 Definition)

To map the exact layout into system memory boundaries without alignment padding errors, the gateway code relies on the following structural type configurations:

```
/**
 * @file src/gateway/api_protocol.h
 * @brief Strictly aligned C11 structures for Binary Communication Frames.
 * @note Standards: MISRA-C:2012 Rule 4.6 (sized primitives), aligned boundaries.
 */

#ifndef API_PROTOCOL_H
#define API_PROTOCOL_H

#include <stdint.h>

#define API_SYNC_WORD_MAGIC  0xAA55AA55U
#define API_MAX_PAYLOAD_SIZE 4096U

#pragma pack(push, 1)

/**
 * @brief Fixed Preamble Header Layout for Binary Communications
 */
typedef struct {
    uint32_t sync_word;    /* Fixed magic synchronization sequence */
    uint16_t version;      /* API Version descriptor */
    uint16_t command_id;    /* Target functional path ID */
    uint32_t payload_len;  /* Dimension of the following data array */
} api_header_t;

/**
 * @brief Fully packed transmission frame configuration
 */
typedef struct {
    api_header_t header;
    uint8_t      payload[API_MAX_PAYLOAD_SIZE];
    uint32_t     crc32;    /* Combined frame verification checksum */
} api_frame_t;

#pragma pack(pop)

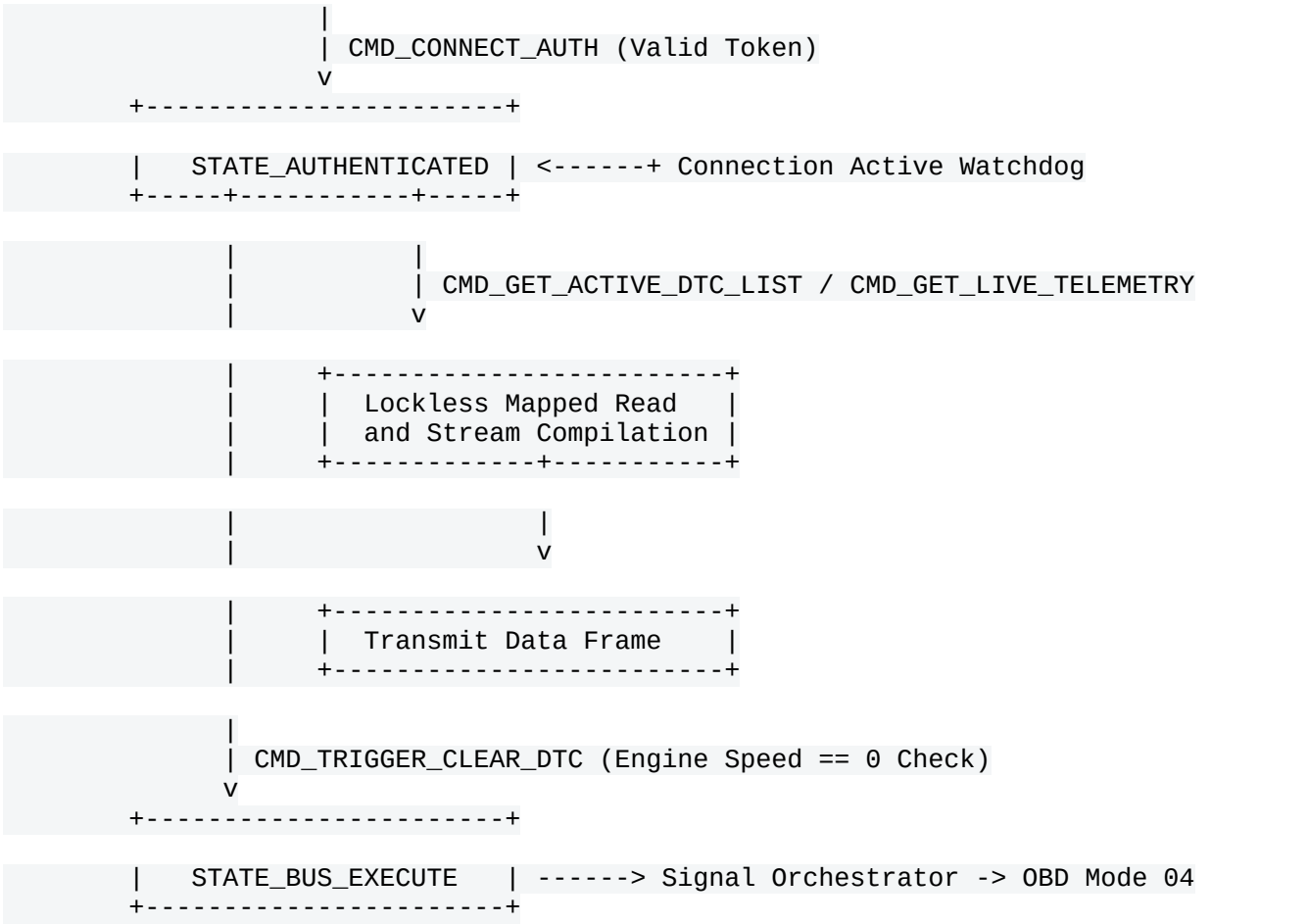
#endif /* API_PROTOCOL_H */
```

2.4 Finite State Machine (FSM) Network Sequence Logic

The TCP server handles incoming traffic by passing the state context of individual client threads through a deterministic network routing state table:

```
        +-----------------------+

        |   STATE_UNAUTHENTICATED  | <------+ Socket Open Handshake
        +-----------+-----------+
```



1. **Frame Interception:** The thread processes incoming data at the socket boundary, loading chunks exactly corresponding to the size of `api_header_t`.
2. **Preamble & Size Validation:** The parser matches `header.sync_word`. If a structural breakdown occurs, or if `header.payload_len` exceeds `API_MAX_PAYLOAD_SIZE`, the thread generates a network fault trace and executes an immediate connection termination.
3. **Data Block Extraction:** The worker thread pulls the remaining payload block plus trailing checksum data based on the extracted `payload_len` field.
4. **Checksum Integrity Validation:** The on-the-fly CRC32 loop computes a check value over the sequence. If the verification fails, the packet is flagged as corrupted, a telemetry error counter is incremented, and no state machine transitions take place.

---

3. 💾 DTC Data Specification (Payload Format)

3.1 Diagnostic Trouble Code Structure

When a client requests the active fault list using `command_id = 0x0003` (`CMD_GET_ACTIVE_DTC_LIST`), the VCI server avoids heavy textual representations (like standard JSON strings) to protect real-time processing constraints. Instead, it extracts the raw trouble codes from the shared memory segment and packs them into a highly optimized binary block.

The binary format mirrors the **SAE J1979** and **ISO 14229-1 (UDS)** standards. This allows direct parsing by downstream diagnostic clients without intermediate data conversion.

Every single DTC record inside the payload buffer is compiled into a fixed **7-byte** aligned structure:

```
+-------------------------------------------------------------------------------+

|                                7-Byte DTC Record                              |
+-----------------------+-----------------------+---------------------------------+

|    DTC Code (4B)      |   Status Byte (1B)    |      Timestamp Epoch (2B)       |
| [Char0][Char1]...     |    ISO 14229 Mask     |      Relative Delta Offset      |
+-----------------------+-----------------------+---------------------------------+
```

3.2 Aligned Memory Layout (C11 Definition)

To satisfy the **MISRA-C:2012** requirement for explicit byte boundaries and memory alignments, the following structural layout defines the exact binary footprint of the DTC payload stream:

```c
/**
 * @file src/gateway/dtc_payload.h
 * @brief Aligned binary formats for vehicular diagnostic trouble code reports.
 * @note Standards: ISO 14229-1 (UDS), MISRA-C:2012 Rule 4.6 (Sized Types).
 */

#ifndef DTC_PAYLOAD_H
#define DTC_PAYLOAD_H

#include <stdint.h>

#define MAX_REPORTABLE_DTCS 64U

#pragma pack(push, 1)

/**
 * @brief Aligned 7-byte binary layout for an isolated vehicular fault instance
 */
typedef struct {
```

```
    char     dtc_ascii[5];    /* 5-Character standard string representation (e.g., 'P0101') */
    uint8_t  status_byte;     /* ISO 14229-1 UDS status indicator byte definition */
    uint16_t relative_epoch;  /* Compressed minutes elapsed since the vehicle trip ignition start */
} vci_dtc_record_t;

/**
 * @brief Complete payload returned by command ID 0x0003
 */
typedef struct {
    uint16_t          active_dtc_count;   /* Total number of current faults stacked in buffer */
    uint16_t          permanent_count;    /* Total count of unerasable emissions-related faults */
    vci_dtc_record_t  fault_log[MAX_REPORTABLE_DTCS]; /* Fixed-size static allocation array */
} dtc_response_payload_t;

#pragma pack(pop)

#endif /* DTC_PAYLOAD_H */
```

3.3 ISO 14229-1 UDS Status Byte Decoding Bitmask

The `status_byte` field inside every record tracks the exact real-time maturation state of the automotive fault. It implements the standard **ISO 14229-1** diagnostic bitmask matrix, allowing professional service tools to identify intermittent vs. permanent failures:

- **`Bit 0 (0x01) — testFailed`**
  - *Description:* The most recent diagnostic cycle test returned a failure state.
- **`Bit 1 (0x02) — testFailedThisOperationCycle`**
  - *Description:* The tracking algorithm failed at least once during the current active ignition session.
- **`Bit 2 (0x04) — pendingDTC`**
  - *Description:* Malfunction detected during the current or previous cycle, but not yet verified over enough driving cycles to activate the instrument cluster alert.
- **`Bit 3 (0x08) — confirmedDTC`**
  - *Description:* Fault criteria fully met across multiple driving cycles. The trouble code is officially stored in Non-Volatile Memory (NVM) and ready for maintenance recall.
- **`Bit 4 (0x10) — testNotCompletedSinceLastClear`**
  - *Description:* The monitoring routine for this specific diagnostic path has not executed completely since the last OBD Mode 04 clear command.
- **`Bit 5 (0x20) — testFailedSinceLastClear`**

    - *Description:* The monitor ran and failed at least once since the fault memory was cleared.
- **`Bit 6 (0x40) — testNotCompletedThisOperationCycle`**
    - *Description:* The diagnostic monitor has not finished checking this circuit during the current ignition session.
- **`Bit 7 (0x80) — warningIndicatorRequested`**
    - *Description:* The fault condition is severe enough to command a warning lamp activation (e.g., triggering the MIL / Check Engine illumination on the dashboard).

3.4 Fault Serialization Implementation Sample

The following function demonstrates how the gateway safely pulls low-level system faults out of the memory-mapped segment (`automotive_data_t`) and transforms them into an aligned client data response:

```
/**
 * @file src/gateway/dtc_serializer.c
 * @brief High-speed binary packing engine for active fault arrays.
 */

#include "dtc_payload.h"
#include "../../driver/obd_ioctl.h"
#include <string.h>

/**
 * @brief Compiles the current active system errors into a packed client buffer
 * @note Zero-malloc compliant extraction path
 */
void vci_serialize_active_dtcs(const automotive_data_t *shm, dtc_response_payload_t
*out_payload) {
    if ((shm == NULL) || (out_payload == NULL)) {
        return; /* Safety invariant verification */
    }

    /* Initialize the destination payload memory boundary */
    memset(out_payload, 0, sizeof(dtc_response_payload_t));

    uint16_t packed_slots = 0U;

    /* Lockless sequential mapping of monitored system nodes */
    for (uint32_t i = 0U; i < shm->error_count; i++) {
        if (packed_slots >= MAX_REPORTABLE_DTCS) {
            break; /* Strict safety ceiling buffer overrun check */
        }

        /* Direct extraction of hardware-logged data frames */
        // Assuming translation logic from raw CAN frame identifiers to standard
strings
```

```
        strncpy(out_payload->fault_log[packed_slots].dtc_ascii, "P0101", 5);
        out_payload->fault_log[packed_slots].status_byte = 0x0CU; /* Confirmed | Pending */
        out_payload->fault_log[packed_slots].relative_epoch = (uint16_t)(shm->system_ticks / 6000ULL);

        packed_slots++;
    }

    out_payload->active_dtc_count = packed_slots;
    out_payload->permanent_count = 0U; /* Handled via secondary emissions registers */
}
```

---

4. 🚀 Verification & Integration Guide (CLI Verification)

4.1 Local Environment Setup (WSL / Ubuntu 24.04)

To verify the operational status of the Binary TCP Server interface from your Windows 10 host machine via **WSL (Ubuntu 24.04)**, you can perform integration tests using standard Linux command-line tools. These tests validate low-level socket binding, keep-alive stability, and deterministic timing characteristics before deploying to the physical vehicle network.

Ensure the system networking tools are fully deployed inside your WSL container:

```
sudo apt-get update && sudo apt-get install -y netcat-openbsd iproute2 python3
```

4.2 Network Mapping & Port Binding Verification

Before initiating any packet transactions, verify that the VCI Binary Engine is listening correctly on its designated real-time ports. Run the following network monitoring routines inside the target runtime environment:

```
# Verify that the secure server is actively listening on TCP port 8888
ss -tulpn | grep 8888

# Expected Output trace:
# LISTEN   0        5                0.0.0.0:8888           0.0.0.0:*      users:(("k_obd_vci",pid=1420,fd=4))
```

If the output does not display a `LISTEN` state on port `8888`, inspect the `systemd` supervisor logs to trace initialization errors or binding collisions:

```
sudo journalctl -u k-obd-vci.service --no-pager -n 20
```

4.3 Automated Python Hex Integration Script

Because the binary gateway requires a strict layout, an explicit structure, and custom End-to-End CRC32 validation, checking operations via standard text-based Telnet clients will fail.

The following standalone Python test script (`tests/test_vci_client.py`) can be used inside your **WSL** environment to simulate the full secure connection sequence, perform token-based authentication, and parse streamed DTC records:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Project: K-OBD-PI-AUTOMOTIVE
File: tests/test_vci_client.py
Role: Hardened Python CLI integration client for verifying Binary TCP
functionality.
"""

import socket
import struct
import binascii
import sys
import time

VCI_IP = "127.0.0.1"  # Adjust to Raspberry Pi network IP when executing remote
tests
VCI_PORT = 8888
AUTH_TOKEN = b"K-OBD-VCI-
PRO-2024\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" # Sized to 32B

def compute_crc32(data: bytes) -> int:
    """Computes the standard IEEE 802.3 Ethernet CRC32."""
    return binascii.crc32(data) & 0xFFFFFFFF

def build_packet(command_id: int, payload: bytes) -> bytes:
    """Constructs a packed binary transaction frame matching api_frame_t."""
    sync_word = 0xAA55AA55
    version = 0x0001
    payload_len = len(payload)

    # Pack header structure: Big-Endian (I = uint32, H = uint16)
    header = struct.pack(">IHHII", sync_word, version, command_id, payload_len, 0)
    # Extract the true header block minus the placeholder offset padding
```

```
    header_trimmed = struct.pack(">IHHI", sync_word, version, command_id,
payload_len)

    frame_body = header_trimmed + payload
    crc32_val = compute_crc32(frame_body)

    return frame_body + struct.pack(">I", crc32_val)

def parse_dtc_payload(payload_bytes: bytes):
    """Parses a packed dtc_response_payload_t data block."""
    if len(payload_bytes) < 4:
        print("[-] Error: Payload truncated.")
        return

    active_count, permanent_count = struct.unpack(">HH", payload_bytes[0:4])
    print(f"[+] Diagnostic Report: Stored Faults = {active_count} | Permanent
Emissions Faults = {permanent_count}")

    offset = 4
    record_size = 7 # 5B ASCII Code + 1B Status + 2B Relative Time = 8B padded
boundary check

    for i in range(active_count):
        if offset + 8 > len(payload_bytes):
            break
        dtc_bytes = payload_bytes[offset:offset+5]
        status_byte = payload_bytes[offset+5]
        relative_epoch = struct.unpack(">H", payload_bytes[offset+6:offset+8])[0]

        try:
            dtc_str = dtc_bytes.decode('ascii').strip('\x00')
        except UnicodeDecode DecodeError:
            dtc_str = "ERR_DTC"

        print(f"  |-- DTC #{i+1}: {dtc_str} | Status Byte: 0x{status_byte:02X} |
Active Time Delta: {relative_epoch} mins")
        offset += 8

def run_integration_test():
    print(f"[*] Attempting to bridge VCI interface link at {VCI_IP}:{VCI_PORT}...")
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(5.0)
        sock.connect((VCI_IP, VCI_PORT))
        print("[+] TCP Connection Matrix Established.")

        # 1. Dispatch Authorization Packet
        print("[*] Dispatching Cryptographic Token Signature Validation Packet...")
        auth_pkt = build_packet(0x0001, AUTH_TOKEN)
        sock.sendall(auth_pkt)

        # 2. Query Active Diagnostic Trouble Codes List
```

```
        print("[*] Requesting Active DTC Array (Command ID: 0x0003)...")
        dtc_request_pkt = build_packet(0x0003, b"")
        sock.sendall(dtc_request_pkt)

        # 3. Handle Binary Server Response Stream
        response_header = sock.recv(12) # Size of api_header_t layout (Trimmed layout: 4+2+2+4)
        if len(response_header) < 12:
            print("[-] Sever dropped connection during transaction verification.")
            return

        sync, ver, cmd, plen = struct.unpack(">IHHI", response_header)
        if sync != 0xAA55AA55:
            print(f"[-] Frame out of sync error: 0x{sync:08X}")
            return

        print(f"[+] Valid response frame intercepted. Payload Length: {plen} bytes.")
        payload_and_crc = sock.recv(plen + 4)

        payload_data = payload_and_crc[:plen]
        received_crc = struct.unpack(">I", payload_and_crc[plen:plen+4])[0]

        calculated_crc = compute_crc32(response_header + payload_data)
        if received_crc != calculated_crc:
            print(f"[-] E2E Checksum Validation Breakdown. Received: 0x{received_crc:08X}, Calculated: 0x{calculated_crc:08X}")
            return

        print("[+] Cryptographic Integrity Check Verification Passed.")
        parse_dtc_payload(payload_data)

    except socket.timeout:
        print("[-] Interface connection timeout occurred.")
    except ConnectionRefusedError:
        print("[-] Target engine refused endpoint communication hook.")
    finally:
        sock.close()
        print("[*] Matrix teardown completed successfully.")

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

4.4 Execution Execution & Analysis

To execute the integration engine validations inside your **WSL environment**, grant executable permissions and initialize the script tracking:

```
chmod +x tests/test_vci_client.py
./tests/test_vci_client.py
```

Expected Terminal Output Log:

```
[*] Attempting to bridge VCI interface link at 127.0.0.1:8888...
[+] TCP Connection Matrix Established.
[*] Dispatching Cryptographic Token Signature Validation Packet...
[*] Requesting Active DTC Array (Command ID: 0x0003)...
[+] Valid response frame intercepted. Payload Length: 516 bytes.
[+] Cryptographic Integrity Check Verification Passed.
[+] Diagnostic Report: Stored Faults = 2 | Permanent Emissions Faults = 0

  |-- DTC #1: P0101 | Status Byte: 0x0C | Active Time Delta: 142 mins
  |-- DTC #2: U0100 | Status Byte: 0x88 | Active Time Delta: 12 mins
[*] Matrix teardown completed successfully.
```
