**Ping-Pong Mechanism**

The ping-pong mechanism in the code is used to ensure that the WebSocket connection between the server and the client remains active and responsive. This mechanism involves the server periodically sending a PING frame to the client and expecting a PONG frame in response. If the server does not receive a PONG frame within a specified timeout, it increments a missed PONG counter. If too many PONG frames are missed, the server closes the WebSocket connection.

**Key Components**

1. *Ping Task (`websocket\_ping\_task`):*

- This task runs in an infinite loop, periodically sending PING frames to the client.

- It waits for a PONG response within a specified timeout (30 seconds).

- If a PONG is not received, it increments the missed PONG counter.

- If the number of missed PONGs exceeds a threshold (>5), it closes the WebSocket connection

2. *WebSocket Handler* ***(ws\_handler****):*

- This handler processes incoming WebSocket frames.

- When a PONG frame is received, it resets the ***pong\_received*** flag and missed PONG counter

**Code Explanation**

*Ping Task* ***(websocket\_ping\_task****):*

**void websocket\_ping\_task(void \*arg) {**

**httpd\_ws\_frame\_t ws\_frame = {**

**.type = HTTPD\_WS\_TYPE\_PING, // WebSocket frame type**

**.payload = NULL, // No payload for a PING frame**

**.len = 0 // Length of the payload**

**};**

**while (1) {**

**if (ws\_context.sock\_fd != -1) { // Ensure a client is connected**

**// Send PING frame**

**if (httpd\_ws\_send\_frame\_async(ws\_context.server\_handle, ws\_context.sock\_fd, &ws\_frame) != ESP\_OK) {**

**ESP\_LOGE("PING", "Failed to send PING frame");**

**ws\_context.missed\_pongs++;**

**} else {**

**ESP\_LOGI("PING", "PING frame sent");**

**ws\_context.pong\_received = false; // Reset pong\_received flag**

**}**

**// Wait for PONG response**

**vTaskDelay(pdMS\_TO\_TICKS(WS\_PING\_TIMEOUT\_MS));**

**if (!ws\_context.pong\_received) {**

**ws\_context.missed\_pongs++;**

**ESP\_LOGW("PING", "PONG not received, missed count: %d", ws\_context.missed\_pongs);**

**} else {**

**ws\_context.missed\_pongs = 0; // Reset missed\_pongs if PONG is received**

**}**

**// Close connection if too many PONGs are missed**

**if (ws\_context.missed\_pongs >= WS\_MAX\_MISSED\_PONGS) {**

**ESP\_LOGE("PING", "Client unresponsive, closing connection");**

**httpd\_sess\_trigger\_close(ws\_context.server\_handle, ws\_context.sock\_fd); // Close WebSocket connection**

**ws\_context.sock\_fd = -1; // Reset socket**

**}**

**}**

**vTaskDelay(pdMS\_TO\_TICKS(WS\_PING\_INTERVAL\_MS)); // Wait before sending the next PING**

**}**

**}**

*WebSocket Handler (****ws\_handler****):*

**esp\_err\_t ws\_handler(httpd\_req\_t \*req)**

**{**

**// ... existing code ...**

**// Check the frame type**

**if (ws\_pkt.type == HTTPD\_WS\_TYPE\_TEXT) {**

**// Data frame received**

**ESP\_LOGI(TAG, "Received data frame: %.\*s", ws\_pkt.len, (char \*)ws\_pkt.payload);**

**// Print data to UART**

**uart\_write\_bytes(UART\_NUM\_0, (const char \*)ws\_pkt.payload, ws\_pkt.len);**

**//ESP\_LOGI(TAG, "Forwarded data to UART");**

**} else if (ws\_pkt.type == HTTPD\_WS\_TYPE\_PONG) {**

**// PONG frame received**

**ESP\_LOGI(TAG, "Received PONG frame");**

**ws\_context.pong\_received = true; // Update PONG flag**

**} else {**

**ESP\_LOGW(TAG, "Unsupported WebSocket frame type: %d", ws\_pkt.type);**

**}**

**// Free the allocated memory for the payload**

**free(ws\_pkt.payload);**

**return ESP\_OK;**

**}**

**Generate URI**

The **generate\_uri** function generates a unique URI based on the device's MAC address. This URI is used to identify the WebSocket endpoint for the device.

**Key Components**

1. *MAC Address Retrieval*:

- The function retrieves the MAC address of the device using the `esp\_wifi\_get\_mac` function.

2. *URI Generation*:

- The function generates a URI string using the last three bytes of the MAC address.

- The URI is formatted as `/SI\_XXYYZZ`, where `XX`, `YY`, and `ZZ` are the hexadecimal representations of the last three bytes of the MAC address.

**Code Explanation**

*Generate URI* ***(generate\_uri****):*

**void generate\_uri(char \*uri, size\_t size) {**

**uint8\_t mac[6];**

**esp\_err\_t err = esp\_wifi\_get\_mac(WIFI\_IF\_STA, mac);**

**if (err != ESP\_OK) {**

**ESP\_LOGE(TAG, "Failed to get MAC address");**

**return;**

**}**

**// Generate URI using snprintf**

**int ret = snprintf(uri, size, "/SI\_%02X%02X%02X", mac[3], mac[4], mac[5]);**

**if (ret < 0 || ret >= size) {**

**ESP\_LOGE(TAG, "URI generation failed or truncated");**

**return;**

**}**

**ESP\_LOGI(TAG, "Generated URI: %s", uri);**

**}**

**Summary**

- *Ping-Pong Mechanism*: Ensures the WebSocket connection remains active by periodically sending PING frames and expecting PONG responses. If too many PONGs are missed, the connection is closed.

- *Generate URI*: Generates a unique URI based on the device's MAC address, which is used to identify the WebSocket endpoint for the device When the user scans the QR code, it contains a "pop" value in the format PWD\_XXYYZZ, where XXYYZZ represents the last six digits of the Wi-Fi MAC address. By concatenating this value with SI\_, app developers can generate unique endpoints.