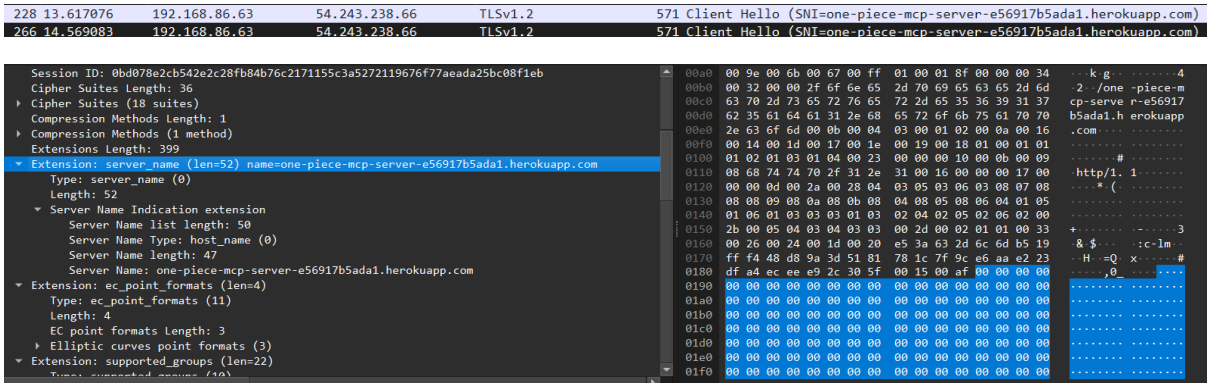
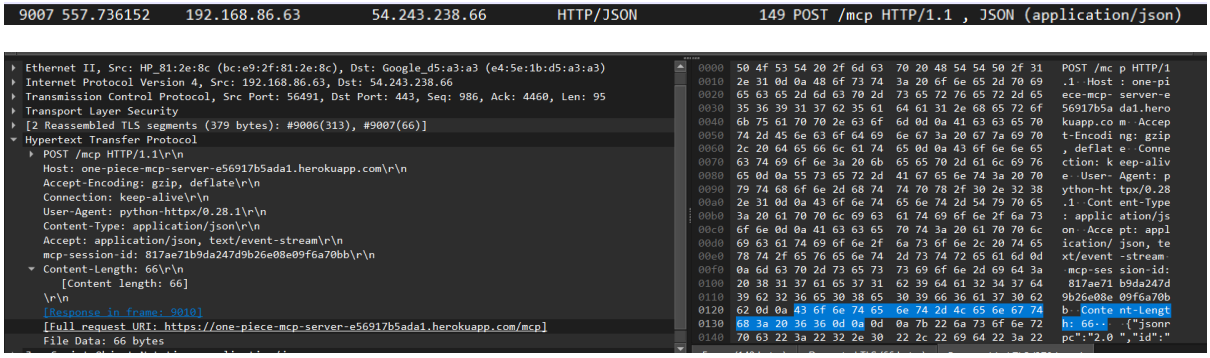
Universidad del Valle de
Guatemala
Redes
Javier Prado - 21496

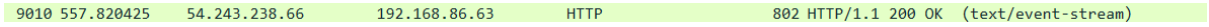
Proyecto 1

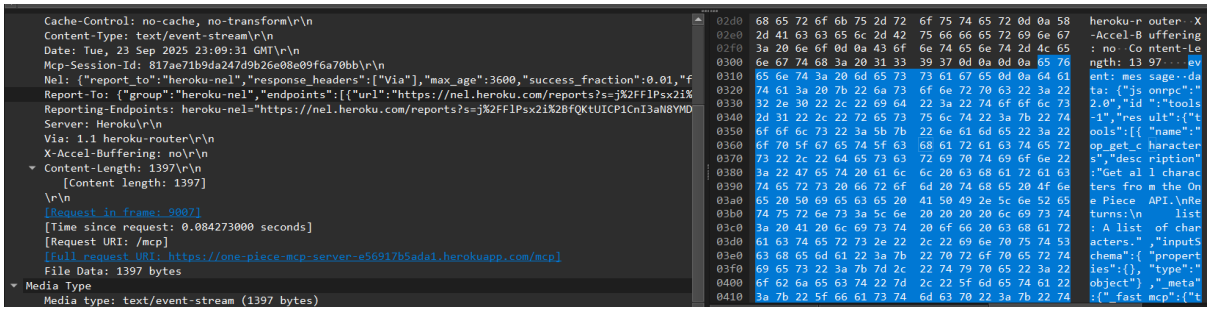
Handshake:



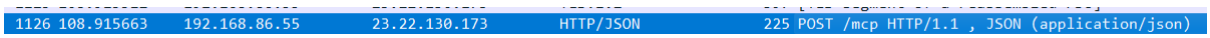
Sync:



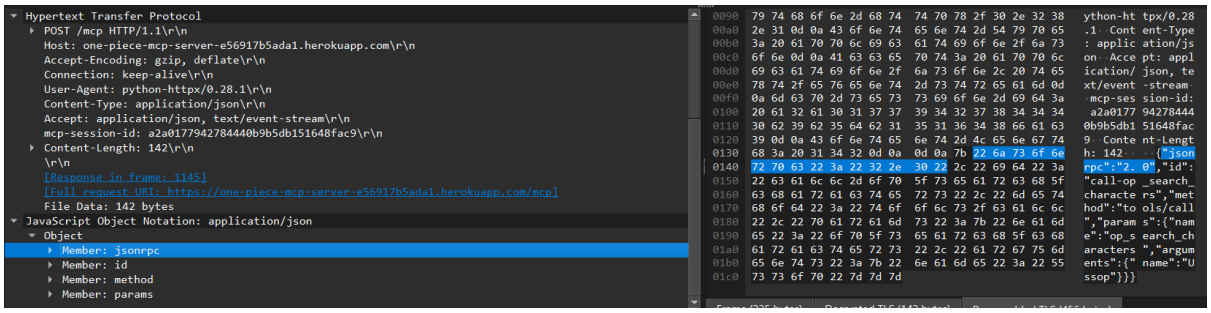




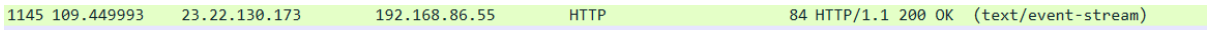
Mensaje de solicitud:

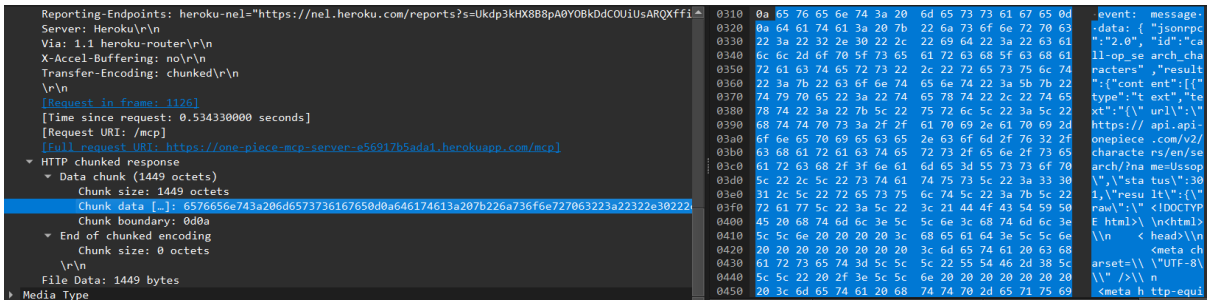


Universidad del Valle de
Guatemala
Redes
Javier Prado - 21496



Mensaje de respuesta:





¿Qué sucede a nivel de la capa de enlace, red, transporte y aplicación?

Durante la interacción entre mi chatbot y el MCP remoto, la capa de enlace (L2) transporta cada unidad de datos como una trama Ethernet II. En cada trama se observan las direcciones físicas locales: la MAC de destino corresponde al siguiente salto (típicamente el gateway o el switch del segmento) y la MAC de origen es la de mi NIC. El campo Type = 0x0800 indica que la carga útil entregada a la capa superior es IPv4.

Esta comunicación es local al enlace: las tramas solo viajan entre mi equipo y el dispositivo adyacente (switch/AP/gateway). En los saltos siguientes, cada equipo vuelve a re-encapsular el paquete con sus propias MAC de origen y destino; por eso, en la captura no aparece la MAC del servidor de Heroku, sino la del primer salto de mi red.

Durante la capa de red (L3) el tráfico viaja encapsulado en datagramas IPv4 desde mi host hacia la infraestructura de Heroku. En cada paquete se observan la IP de origen (mi equipo dentro de la LAN) y la IP de destino (un balanceador/edge de Heroku). Al salir de mi red, el router realiza NAT: la IP privada de mi equipo se traduce a una IP pública, por eso en Internet los paquetes aparecen con la IP pública del router y no con la privada de mi PC. A lo largo del trayecto, cada router decrementa el TTL (time-to-

live); en los paquetes recibidos desde la nube el TTL llega con un valor menor, consistente con el número de saltos recorridos.

En la capa de transporte, TCP establece la sesión (3-Way Handshake), aplica fiabilidad, control de flujo y de congestión, segmenta/reensambla datos y cierra con FIN/RST. Sobre ese flujo estable y cifrado (tras TLS) circulan los mensajes de aplicación que tú capturaste: el SYNC (tools/list), los REQUEST (tools/call) y sus RESPONSE (mismo id). La latencia de cada llamada puede medirse en L4/L7 como el cambio ntre el paquete del request y el de su response dentro del mismo tcp.stream.

A nivel de aplicación se observa (1) un SYNC inicial tools/list para descubrir capacidades, requests tools/call con sus argumentos, y responses con result/error, siempre correlacionados por id. Las cabeceras Accept, Content-Type y mcp-session-id sostienen la sesión y el formato

Servidor Soccer

Tool	**Endpoint (método y ruta relativa)**	**Parámetros (path / query)**	**Especificación / Descripción**	**Requisitos (headers / env)**
get_competitions	**GET** /competitions	—	Lista todas las competiciones disponibles.	Header: X-Auth-Token = API_KEY. Env: API_KEY, FOOTBALL_BASE, HTTP_TIMEOUT (opcional).
get_teams_competitions	**GET** /competitions/{competition_id}/teams	competition_id (path, p. ej. PL, CL)	Lista los equipos de una competición.	Header: X-Auth-Token = API_KEY. Env: API_KEY, FOOTBALL_BASE, HTTP_TIMEOUT.
get_teams_by_competition	**GET** /competitions/{competition_id}/teams	competition_id (path)	(Funcionalidad duplicada de la anterior) Lista	Igual que arriba.

			equipos por competición.	
get_matches_by_competition	**GET** /competitions/{competition_id}/matches	competition_id (path)	Lista los partidos de una competición.	Header/env como arriba.
get_team_by_id	**GET** /teams/{team_id}	team_id (path)	Devuelve el detalle de un equipo.	Header/env como arriba.
get_top_scorers_by_competitions	**GET** /competitions/{competition_id}/scorers	competition_id (path)	Goleadores top de una competición.	Header/env como arriba.
get_player_by_id	**GET** /persons/{player_id}	player_id (path)	Devuelve el detalle de un jugador/persona.	Header/env como arriba.
get_info_matches_of_a_player	**GET** /persons/{player_id}/matches	player_id (path)	Lista los partidos de una persona/jugador.	Header/env como arriba.

One piece server

Tool	**Endpoint (método y ruta relativa)**	**Parámetros (path / query)**	**Especificación / Descripción**	**Requisitos (headers / env)**
op_get_characters	**GET** /characters/en	—	Devuelve la lista de personajes.	Env: ONE_PIECE_BASE, HTTPX_TIMEOUT (opcional). Maneja errores y response.json().

op_get_character_by_id	**GET** /characters/en/{character_id}	character_id (path, int)	Devuelve el detalle de un personaje por ID.	Env: ONE_PIECE_BASE, HTTPX_TIMEOUT.
op_search_characters	**GET** /characters/en/search/	**query** opcionales: name, job, bounty, age, size	Busca personajes con filtros. Si no envías ningún filtro, devuelve error útil. Devuelve {url, status, result} o raw si el server no manda JSON.	Env: ONE_PIECE_BASE, HTTPX_TIMEOUT. Usa httpx y maneja redirecciones/CT.

Conclusiones y comentario sobre el proyecto

Este proyecto me permitió cerrar el ciclo completo de una integración cliente–servidor moderna: diseñé y expuse herramientas mediante MCP (Model Context Protocol), las consumí desde un chatbot, desplegué un servidor remoto y verifiqué el tráfico con Wireshark hasta clasificar los mensajes JSON-RPC en sync, request y response.

Lo más retador fue establecer la comunicación correcta con el MCP remoto. Tuve que entender cómo formatear correctamente JSON-RPC 2.0 (campos jsonrpc, id, method, params), qué cabeceras HTTP exige el servidor (Accept: application/json, text/event-stream y Content-Type: application/json) y cómo manejar el estado de sesión vía mcp-session-id.

En términos de implementación, MCP Fast hizo muy directa la creación de tools: el patrón “defino una función → la publico como tool → la invoco con tools/call” facilita pensar en la capa de aplicación sin pelear. La separación de transportes (stdio para local y streamable-http para remoto) me obligó a ser explícito al configurar el chatbot y me ayudó a entender por qué a veces “no había tráfico” (estaba usando stdio en lugar de HTTP).

Repositorios:

Javilejoo/chatbot-soccerMCP

Javilejoo/Soccer-MCP-Server