Proyecto 1 - Uso de un protocolo existente

Nancy Mazariegos – 22513

ESPECIFICACIÓN DE LOS SERVIDORES MCP DESARROLLADOS

- Servidor remoto de conversión de temperatura (Flask).
- Objetivo: exponer una “tool” remota para convertir temperaturas Celsius ↔ Fahrenheit, invocable por chatbot u otros clientes usando JSON-RPC 2.0 sobre HTTP.

Stack y despliegue:

- Runtime: Python 3.11.
- Framework: Flask con gunicorn.
- Plataforma: Google App Engine (Standard).
- Escalado sugerido: min_instances = 1 para evitar cold start.
- Observabilidad: gcloud app logs read/tail por versión.

Modelo de transporte y aplicación:

- Capa de transporte: TCP con TLS en producción (HTTPS).
- Capa de aplicación: HTTP/1.1 o HTTP/2 (según negociación con el front-end de Google).
- Formato de mensajes: JSON.
- Estilo de API: JSON-RPC 2.0 (campos jsonrpc, id, method, params).

Endpoints expuestos:

- GET / → health básico. Respuesta 200 con JSON: { "service": "remote-temp-mcp (flask)", "rev": "...", "status": "ok" }.
- GET /health → health explícito con estado y versión. Respuesta 200.
- POST / → entrada JSON-RPC 2.0.
  Ejemplo de request: { "jsonrpc": "2.0", "id": 1, "method": "convert_temp", "params": { "value": 25, "unit": "C" } }.
  Ejemplo de response éxito: { "jsonrpc": "2.0", "id": 1, "result": "25.0 °C = 77.00 °F" }.
  Errores de cliente: 400 con códigos JSON-RPC -32700 (parse error) o -32600 (invalid request).
  Método inexistente: 404 con código JSON-RPC -32601 (method not found).
- Opcionales de estilo MCP para pruebas:
  GET /mcp/tools/list → describe herramientas disponibles.
  POST /tools/convert_temp/call → acepta cuerpo con { "arguments": { "value": 32, "unit": "F" } } y devuelve { "result": "..." }.

Tool disponible:

- Nombre: convert_temp.
- Parámetros:
  value (float, requerido) temperatura de entrada.
  unit (string, opcional) “C” por defecto, también admite “F”.

- Validaciones:
  value no numérico → “ERROR: <value> debe ser numérico.”
  unit distinta de C o F → “Unidad no válida. Usa ‘C’ para Celsius o ‘F’ para Fahrenheit.”

Códigos HTTP esperados:

- 200: operación correcta y health.
- 400: JSON inválido o request mal formado.
- 404: método no reconocido.
- 5xx: errores no controlados (por ejemplo, crash en arranque). En despliegues previos se observaron 503 cuando la instancia estaba unhealthy o el binario no coincidía.

EXPLICACIÓN POR CAPAS: ENLACE, RED, TRANSPORTE Y APLICACIÓN

Capa de enlace (L2):

- En el host del cliente, los datagramas IP se encapsulan en tramas Ethernet o Wi-Fi con direcciones MAC origen/destino.
- En cada salto local (switch, AP) las tramas se reencapsulan. No hay lógica de aplicación; es entrega hop-by-hop.

Capa de red (L3):

- DNS resuelve el dominio *.r.appspot.com a una IP pública del front-end de Google.
- El ruteo IP transporta los paquetes por Internet; puede intervenir NAT de salida del cliente. TTL y posible fragmentación no cambian la semántica de la app.

Capa de transporte (L4):

- Se establece una conexión TCP (three-way handshake).
- Sobre TCP se negocia TLS (HTTPS) y, mediante ALPN, HTTP/1.1 o HTTP/2. En HTTP/2 pueden multiplexarse varios streams en la misma conexión.
- Si el proceso de la app no escucha o muere, el proxy puede responder 503 aunque las capas L2/L3/L4 estén sanas.

Capa de aplicación (L7):

- Protocolo HTTP: POST / con Content-Type application/json para la llamada de negocio; GET / y GET /health para healthchecks.
- Protocolo lógico JSON-RPC 2.0: se validan jsonrpc, id, method, params. Si method = convert_temp, se ejecuta la conversión y se devuelve result; si el método no existe o el JSON es inválido, se devuelven los códigos y mensajes correspondientes.

- En la experiencia del proyecto, la ausencia o fallo de endpoints de salud y/o un binario distinto al desplegado causaron 503. Al añadir health endpoints y asegurar min_instances = 1, la plataforma consideró la instancia healthy y los 503 desaparecieron.

Mapa de síntomas por capa:

- 503 Service Unavailable: típicamente problema de aplicación/plataforma (instancia unhealthy o crash), no de red.
- 500: error interno de aplicación.
- 400/404: petición mal formada o método inexistente en la capa de aplicación.
- Ping/latencia OK más 503: indica que L2/L3/L4 están bien y la falla es de health/boot en L7.

COMENTARIOS:

- Robustez operativa: incorporar GET / y GET /health y fijar min_instances = 1 estabilizó la disponibilidad en App Engine y eliminó 503 por unhealthy.
- Simplicidad efectiva: JSON-RPC 2.0 sobre HTTP es ligero y suficiente para exponer tools remotas; facilita pruebas con cURL/Postman y la integración con el chatbot.
- Separación de responsabilidades: aislar la lógica de conversión del transporte permitió migrar a Flask sin romper el contrato externo.
- Observabilidad: versionado visible (REV) y filtrado de logs por versión agilizaron el diagnóstico de "código desplegado vs. código ejecutándose".
- Entendimiento de la plataforma: un 503 no implica falla de red; suele significar que la aplicación no arrancó o está marcada como unhealthy por los healthchecks.
- Trabajo futuro recomendado: autenticación (API key o JWT), rate limiting, validación formal con JSON Schema, métricas y trazas, pruebas de carga, y pipeline de despliegue con promoción controlada de versiones.

CONCLUSIONES:

- Al migrar a Flask + gunicorn y exponer GET / y /health con min_instances=1, se erradicaron los 503 por "unhealthy" y se logró disponibilidad estable con mejor latencia percibida.
- Mantener el contrato JSON-RPC 2.0 sobre HTTP desacopló al cliente de la implementación, facilitó pruebas (cURL/Postman) y permite evolucionar la lógica sin romper integraciones.
- El uso de versionado visible (REV), logs por versión y promoción de tráfico controlada aceleró el diagnóstico y operación; el pequeño costo de mantener una instancia mínima se compensó con mayor confiabilidad.