Documentación Detallada del Servidor TinyFaaS MQTT

Este documento explica en detalle la arquitectura, el flujo de funcionamiento y la estructura del código interno del servidor **TinyFaaS MQTT**, basado en el archivo server_tinyfaas_persistent_mqtt_v2.py.

---

1. 🏗️ Diagrama Conceptual y Flujo Operativo

El servidor **TinyFaaS MQTT** opera como un servicio persistente que mantiene una conexión continua con un **Broker MQTT**. El sistema está diseñado para manejar la gestión administrativa y la ejecución de funciones de manera **asíncrona y aislada** mediante el uso de *multithreading* y entornos virtuales.

Flujo de Funcionamiento (Diagrama de Bloques)

Bloque	Función Principal	Interacción
Clientes (Usuario/Admin)	Publican comandos (invocaciones o administración) en tópicos específicos.	Publicación MQTT.

Broker MQTT	Punto central de mensajería. Enruta todos los mensajes de petición (faas/*) y respuesta (faas/response/*).	Protocolo MQTT.
Servidor TinyFaaS (Hilo Principal)	Mantiene la conexión MQTT (loop_forever), gestiona las suscripciones y enruta los mensajes entrantes.	Conexión Persistente.

Manejador de Administrador	Procesa comandos síncronos de gestión (upload, delete, list, status). Modifica el estado de las funciones y la persistencia.	Lógica FaaS Core y Persistencia.
Pool de Hilos de Ejecución	Cada petición de **invocación** se despacha a un hilo separado para la ejecución del código, evitando el bloqueo del servidor principal.	Asincronía (threading).

Capa de Ejecución (FaaS)	Aísla el entorno (venv), carga dinámicamente el código de usuario y ejecuta la función main.	**Aislamiento de Entorno Virtual.**
Capa de Persistencia	Almacena el estado del sistema (functions.json) y el historial de logs (logs.json) en disco.	Archivos de estado.
Respuestas MQTT	Publica los resultados de la ejecución o de los comandos administrativos de vuelta al broker.	Publicación MQTT.

---

2. 🐍 Estructura del Código y Explicación Detallada

El código fuente (server_tinyfaas_persistent_mqtt_v2.py) está estructurado en bloques lógicos que reflejan los componentes arquitectónicos, asegurando la modularidad del sistema.

A. ⚙️ CONFIGURACIÓN DEL SERVIDOR

Define las constantes de la aplicación, las rutas de directorios y el estado global del sistema en memoria.

Componente	Tipo	Función Específica
MQTT_BROKER, **MQTT_PORT**, **etc.**	Constantes	Parámetros de conexión de red para el broker MQTT.
MQTT_BASE_TOPIC, **MQTT_RESPONSE_TOPIC**	Constantes	Prefijos de los tópicos para la entrada (faas) y salida (faas/response) de mensajes.
FUNCTIONS_DIR, **DATA_DIR**	Constantes	Rutas de directorios. El código de usuario y los entornos virtuales van en FUNCTIONS_DIR.

functions, **logs**	**Diccionarios Globales**	**Estado en Memoria.** functions guarda los metadatos de las funciones cargadas. logs guarda el historial de ejecución.

B. 💾 FUNCIONES DE PERSISTENCIA Y ENTORNO

Gestionan el estado persistente del servidor y el crucial aislamiento de las funciones mediante entornos virtuales.

Función	Propósito	Detalle del Funcionamiento

save_state() / **load_state()**	Persistencia	Se encargan de la **serialización** (escritura) y **deserialización** (lectura) de los diccionarios globales functions y logs hacia/desde el disco (.json), garantizando que el estado se mantenga tras un reinicio.

setup_function_env(func_name, req_b64)	Aislamiento	**Crea el entorno de ejecución.** Utiliza os.makedirs y **subprocess.run** para ejecutar: 1. python -m venv venv (creación del entorno virtual). 2. Si se dan dependencias (req_b64), las decodifica e instala con pip install -r requirements.txt.

delete_function_env(func_name)	Limpieza	Elimina por completo el directorio de la función (FUNCTIONS_DIR/{func_name}) y todos sus contenidos (código, logs y venv) usando **shutil.rmtree**.

C. 💡 LÓGICA CORE DE FaaS (Core Business Logic)

Contiene las funciones que ejecutan las tareas principales del servidor.

Función	Propósito	Detalle del Funcionamiento

core_upload_function(...)	Carga de Funciones	Desempeña el proceso de **carga del código**: decodifica los code_b64 y req_b64, los guarda como func.py y requirements.txt, e invoca a setup_function_env para preparar el entorno virtual y las dependencias.

execute_code(func_name, args, request_id)	**Ejecución Asíncrona**	1. Modifica **sys.path** para incluir el venv de la función. 2. Usa **importlib.util** para cargar el archivo func.py de forma dinámica. 3. Llama al punto de entrada **module.main(*args)**. 4. Captura el resultado o la excepción y lo registra antes de publicar la respuesta vía MQTT.

core_execute_function(...)	Despacho de Invocación	Crea y arranca un nuevo **hilo** (threading.Thread) para ejecutar execute_code. Esto es **fundamental** para la concurrencia, ya que libera el hilo principal de MQTT para seguir recibiendo comandos mientras la función se ejecuta.
core_get_status()	Monitoreo	Utiliza la librería **psutil** para obtener métricas de rendimiento (uso de CPU, memoria RSS del proceso TinyFaaS y memoria total del sistema) para responder a comandos de estado administrativo.

D. 🧩 CLASE TinyFaaS_MqttServer

Encapsula la interfaz con el protocolo MQTT, gestionando la conexión, la suscripción y el enrutamiento de mensajes.

Método/Propiedad	Propósito	Detalle del Funcionamiento
on_connect(...)	Conexión	Se activa al conectarse al broker. Es donde el cliente MQTT se **suscribe** a los tópicos de petición: faas/invoke/+ (invocaciones) y faas/admin/# (administración).
handle_admin_command(...)	Enrutamiento Admin	Analiza el subtópico del comando administrativo (ej. /upload, /delete) y llama a la función core_* correspondiente, publicando el resultado en el tópico de respuesta específico.

on_message(..., msg)	Manejo de Mensajes	Es el *callback* principal. Intenta decodificar el payload como JSON. **Enruta la petición** basándose en el tópico: si es faas/admin, llama a handle_admin_command; si es faas/invoke, llama al hilo de ejecución core_execute_function.
handle_error(...)	Manejo de Errores	Utilidad para estandarizar la respuesta de error. Publica un *payload* JSON estructurado con el detalle de la excepción en el tópico central: faas/response/error.

run()	Bucle Principal	Establece la conexión (client.connect()) e inicia el bucle de escucha indefinida (client.loop_forever()), que mantiene el servidor en ejecución hasta que se detiene manualmente.

E. 🚀 PUNTO DE ENTRADA (MAIN)

El bloque de inicio del programa, asegurando una inicialización correcta y un cierre controlado.

1. **load_state()**: Carga el estado anterior del servidor (funciones y logs) al iniciar.
2. **Instancia del Servidor**: Crea una instancia de TinyFaaS_MqttServer, inyectándole la función de ejecución (core_execute_function).
3. **try...except KeyboardInterrupt**: Ejecuta mqtt_server.run(). El bloque try/except permite que el servidor se cierre limpiamente al detectar la interrupción del teclado (Ctrl+C).