

⌚ Solução Baseada no Repositório ESP32

📋 Fonte

Repositório: <https://github.com/MarceloClaro/xiaozhi-esp32-server/tree/main/main>

Este é o repositório original (servidor ESP32) que possui implementação completa e funcionando de MCP com câmera.

🔍 Descobertas Importantes

1. Protocolo de Comunicação Documentado

O repositório ESP32 possui documentação detalhada do protocolo:

- **URL:** <https://ccnphfhqs21z.feishu.cn/wiki/M0XiwldO9iJwHikpXD5cEx71nKh>
- Descreve como ESP32 ↔ xiaozhi-server comunicam via WebSocket
- Inclui formato de áudio, comandos de controle, mensagens de status

2. Estrutura MCP no Servidor Original

```
xiaozhi-server/
  ├── core/
  │   ├── providers/      # Provedores de AI (ASR, TTS, LLM, VAD)
  │   ├── handle/         # Handlers de mensagens
  │   │   ├── receiveAudioHandle.py
  │   │   ├── sendAudioHandle.py
  │   │   └── mcpHandle.py ★ Handler MCP
  │   └── functionHandler.py
  └── websocket_server.py
  ├── plugins_func/
  │   ├── functions/      # Sistemas de plugins/funções
  │   ├── loadplugins.py  # Carregador de plugins
  │   └── register.py    # Registro de funções
  └── config.yaml        # Configuração principal
```

3. Arquivo de Configuração MCP

O servidor ESP32 tem um arquivo específico:

- **Arquivo:** `mcp_server_settings.json`
- **Commit:** "update : 服务端MCP新增支持Streamable HTTP传输协议"
- **Data:** 3 meses atrás

⚠ Problema Identificado no Nossa Código

Comparação: ESP32 vs. Nossa Código

Aspecto	ESP32 (Funciona <input checked="" type="checkbox"/>)	Nosso Código (Problema <input checked="" type="checkbox"/>)
Registro de Tools	Via <code>loadplugins.py</code> + <code>register.py</code>	Via <code>add_tool()</code> direto
Handler MCP	<code>mcpHandle.py</code> dedicado	Integrado em <code>mcp_server.py</code>
Inicialização	Síncrona durante <code>__init__</code>	Async via <code>_register_all_tools()</code>
Tools List	Retorna lista completa	Retorna vazio (13 bytes)
Momento do Registro	ANTES de aceitar conexões	Após servidor já estar rodando?

🔧 Soluções Baseadas no ESP32

Solução 1: Sistema de Plugins Similar

Implementar sistema igual ao ESP32:

```
# plugins_func/register.py (NOVO - similar ao ESP32)
_registered_functions = {}

def register_function(name: str, description: str, parameters: dict):
    """Registra uma função no sistema de plugins"""
    def decorator(func):
        _registered_functions[name] = {
            'function': func,
            'description': description,
            'parameters': parameters
        }
        return func
    return decorator

def get_all_functions():
    """Retorna todas as funções registradas"""
    return _registered_functions
```

Modificar `take_photo` para usar decorator:

```
# src/mcp/tools/camera/camera.py
from plugins_func.register import register_function

CAMERA_SCHEMA = {
    "type": "object",
    "properties": {
```

```

        "question": {
            "type": "string",
            "description": "Pergunta sobre o que está vendendo"
        }
    },
    "required": ["question"]
}

@register_function(
    name="take_photo",
    description="Captura foto e analisa conteúdo visual",
    parameters=CAMERA_SCHEMA
)
def take_photo(arguments: dict) -> str:
    """Implementação existente"""
    # ... código atual ...

```

Carregar plugins no init:

```

# src/mcp/mcp_server.py
from plugins_func.register import get_all_functions

class MCPServer:
    def __init__(self, protocol: MCPProtocol):
        self.protocol = protocol
        self.tools: List[McpTool] = []

        # CRÍTICO: Carregar tools ANTES de qualquer outra coisa
        self._load_plugins_sync() # Síncrono!

        logger.info(f"[MCP] Servidor inicializado com {len(self.tools)} tools")

    def _load_plugins_sync(self):
        """Carrega plugins de forma SÍNCRONA (igual ao ESP32)"""
        logger.info("[MCP] Carregando plugins...")

        # Importa módulos para executar decorators
        import src.mcp.tools.camera.camera
        import src.mcp.tools.screenshot.screenshot
        # ... outros módulos

        # Obtém funções registradas
        functions = get_all_functions()

        for name, func_info in functions.items():
            tool = McpTool(
                name=name,
                description=func_info['description'],
                properties=self._convert_schema(func_info['parameters']),
                function=func_info['function']
            )

```

```

        self.tools.append(tool)
        logger.info(f" ✅ Tool carregada: {name}")

```

Solução 2: Inicialização Síncrona (Mais Simples)

Se não quiser reescrever tudo, mova o registro para ANTES:

```

# src/mcp/mcp_server.py
class MCPServer:
    def __init__(self, protocol: MCPProtocol):
        self.protocol = protocol
        self.tools: List[McpTool] = []

        # ★ SOLUÇÃO RÁPIDA: Chamar sync AGORA
        import asyncio
        loop = asyncio.get_event_loop()
        if loop.is_running():
            # Se loop já está rodando, usar create_task
            asyncio.create_task(self._register_all_tools())
        else:
            # Se loop ainda não iniciou, usar run_until_complete
            loop.run_until_complete(self._register_all_tools())

        # OU usar threading para não bloquear
        import concurrent.futures
        with concurrent.futures.ThreadPoolExecutor() as executor:
            future = executor.submit(asyncio.run, self._register_all_tools())
            future.result() # Espera completar

    logger.info(f"[MCP] {len(self.tools)} tools registradas no __init__")

```

Solução 3: Lazy Loading com Lock

Adicionar verificação em tools/list:

```

# src/mcp/mcp_server.py
class MCPServer:
    def __init__(self, protocol: MCPProtocol):
        self.protocol = protocol
        self.tools: List[McpTool] = []
        self._tools_loaded = False
        self._tools_lock = asyncio.Lock()

    async def _ensure_tools_loaded(self):
        """Garante que tools estejam carregadas antes de usar"""
        if not self._tools_loaded:
            async with self._tools_lock:
                if not self._tools_loaded: # Double-check pattern

```

```

        await self._register_all_tools()
        self._tools_loaded = True

    async def _handle_tools_list(self, message: Dict[str, Any]) -> Dict[str,
Any]:
        """Handler com garantia de tools carregadas"""
        # ★ CRÍTICO: Garantir que tools foram carregadas
        await self._ensure_tools_loaded()

        logger.info(f"[MCP TOOLS/LIST] Total de tools registradas:
{len(self.tools)}")
        logger.info(f"[MCP TOOLS/LIST] Tools disponíveis:")
        for tool in self.tools:
            logger.info(f" - {tool.name}")

        # ... resto do código ...

```

Próximos Passos Recomendados

Passo 1: Verificar Logs Atuais

```

python main.py --mode gui --protocol websocket
# Procurar por: [MCP TOOLS/LIST] Total de tools registradas: X

```

Se X = 0: Usar Solução 1 ou 2 (problema de inicialização)

Se X > 0: Problema é no retorno/serialização (Solução 3)

Passo 2: Implementar Solução Mais Adequada

Recomendação:

- **Projeto pequeno:** Use Solução 2 (inicialização síncrona)
- **Projeto em crescimento:** Use Solução 1 (sistema de plugins)
- **Quick fix:** Use Solução 3 (lazy loading)

Passo 3: Testar com Comando Direto

```

# Teste bypass (deve funcionar mesmo com bug):
curl -X POST http://localhost:8000/mcp/vision/explain \
-H "Content-Type: application/json" \
-H "Authorization: Bearer SEU_TOKEN" \
-d '{"question": "O que você vê?", "image": "BASE64_IMAGE"}'

```

Diferenças Arquiteturais Importantes

ESP32 Server (Original)

Fluxo de Inicialização:

1. app.py executa
2. Carrega config.yaml
3. Importa todos os módulos (executa decorators)
4. Plugins auto-registraram via @register_function
5. WebSocketServer inicia (tools já estão lá)
6. Cliente conecta → tools/list → retorna lista completa

Nosso Código (py-xiaozhi)

Fluxo de Inicialização:

1. main.py executa
2. Inicializa Application
3. MCPServer.__init__ cria instância
4. _register_all_tools() é async (não executa imediatamente?)
5. Cliente conecta → tools/list → lista vazia?

🔗 Links Importantes

- **Repositório ESP32:** <https://github.com/MarceloClaro/xiaozhi-esp32-server>
- **Protocolo WebSocket:** <https://ccnphfhqs21z.feishu.cn/wiki/M0XiwlD09iJwHikpXD5cEx71nKh>
- **MCP Settings:** [main/xiaozhi-server/mcp_server_settings.json](#)
- **Plugin System:** [main/xiaozhi-server/plugins_func/](#)

✓ Checklist de Implementação

- Executar com debug logs e verificar contagem de tools
- Se count=0: Implementar uma das 3 soluções
- Se count>0: Verificar serialização/retorno
- Testar com comando direto (bypass)
- Comparar com código ESP32 original
- Adaptar sistema de plugins se necessário
- Validar com comando de voz "tire uma foto"

🎓 Lições do Código ESP32

1. **Sincronicidade importa:** Tools devem estar registradas ANTES do servidor aceitar conexões
2. **Sistema de plugins:** Usar decorators + registro automático é mais robusto
3. **Handler dedicado:** [mcpHandle.py](#) separado facilita manutenção
4. **Documentação:** Protocolo bem documentado facilita debugging

5. Configuração externa: `mcp_server_settings.json` permite ajustes sem code changes

Status:  Documento criado com soluções baseadas no repositório ESP32

Próximo: Executar aplicação e verificar logs com debug