Day98_MCP_Client_Server_Claude_Theory.ipynb

October 16, 2025

1 MCP Client and Server with Claude — Full Theory & Practical Plan

1.1 Introduction

1.1.1 What is MCP?

MCP stands for **Model Context Protocol**.
It is a standardized communication protocol that allows AI models and external tools to interact smoothly and securely.

MCP makes it possible to build **Agentic AI systems**, where: - The **AI model** (e.g., Claude) focuses on **reasoning** and decision-making. - The **MCP server** exposes **tools** (functions, APIs, services). - The **MCP client** acts as a **bridge** between the AI and the server.

In short:
> **Claude thinks → MCP Client listens → MCP Server executes → Claude answers.**

1.2 Why MCP?

1.2.1 Traditional vs MCP

In traditional AI apps: - LLMs only respond to text prompts. - No real-world action or integration.

With MCP: - The AI can call real tools. - It can fetch live data, execute logic, access APIs, and more. - It becomes a **true agent**, not just a chatbot.

1.2.2 Benefits of Using MCP

- **Structured communication:** standardized tool calls.
- **Modular system:** you can add/remove tools easily.
- **Real-time execution:** server runs logic, Claude uses the results.
- **Security:** clear separation between model and tool execution.

1.3 Key Concepts

1.3.1 MCP Server

- The server is like a **toolbox**.

- Each tool is a Python function exposed using MCP capabilities.
- It waits for requests from the client.

Example tools: - `calculator` - `get_weather` - `fetch_data_from_api` - `translate_text`

1.3.2 MCP Client

- The client is the **bridge** between Claude and the server.
- It knows what tools are available (capability discovery).
- When Claude wants something, the client calls the server tool.

1.3.3 Claude (The Agent)

- Claude receives user input.
- It decides if a tool is needed.
- It instructs the MCP client to call a specific tool.
- It uses the result to craft a smart final answer.

1.4 MCP Communication Flow

1.4.1 Step-by-Step Flow

1. User sends a prompt to Claude.
2. Claude decides: "I need to use a tool for this."
3. Claude sends a request to the MCP client.
4. MCP Client calls the right tool on the MCP server.
5. MCP Server runs the function and returns structured data.
6. MCP Client sends this data back to Claude.
7. Claude uses the tool's result and gives the **final answer**.

1.5 System Architecture

1.5.1 Components

- **Claude (Agent)** — brain
- **MCP Client** — connector
- **MCP Server** — tool executor
- **Tools** — custom functions or APIs

1.5.2 Visualization

```
[User] → [Claude Agent]  [MCP Client]  [MCP Server] → [Tools / APIs]
```

1.6 Project Setup

1.6.1 Environment

- Python 3.10+
- Virtual environment
- MCP SDK installed
- Claude API key from Anthropic
- `dotenv` to manage environment variables

1.6.2 Folder Structure

```
mcp_project/
   server/
       tools_server.py
   client/
       client_runner.py
   .env
   README.md
```

1.6.3 Install Required Packages

```
python -m venv venv
source venv/bin/activate      # or venv\Scripts\activate on Windows

pip install mcp python-dotenv requests
```

Add `.env`:

```
CLAUDE_API_KEY=your_api_key_here
```

1.7 Building MCP Server

1.7.1 Theory

- Server exposes **capabilities** — these are like "skills" the agent can use.
- Each capability is a Python function.
- The server runs continuously, waiting for calls from the client.

1.7.2 Example Implementation

```
from mcp.server import MCPServer

server = MCPServer()

@server.capability("say_hello")
def say_hello(name: str):
    return {"message": f"Hello, {name}!"}
```

```python
@server.capability("calculator")
def calculator(a: int, b: int):
    return {"result": a + b}

@server.capability("fetch_weather")
def fetch_weather(city: str):
    return {"city": city, "temperature": "28°C", "condition": "Sunny"}

if __name__ == "__main__":
    server.start()
```

Run:

```
python server/tools_server.py
```

Your server is now live with 3 capabilities.

1.8 Building MCP Client

1.8.1 Theory

- The client knows how to **connect** to the server.
- It can **list** available tools.
- It can **call** any tool with proper arguments.

1.8.2 Example Implementation

```python
from mcp.client import MCPClient

client = MCPClient("http://localhost:8000")

# Discover available tools
print("Available Capabilities:", client.get_capabilities())

# Call calculator tool
result = client.invoke("calculator", {"a": 5, "b": 10})
print(result)
```

Run:

```
python client/client_runner.py
```

1.9 Adding Claude as the Agent

1.9.1 Theory

Claude is the **reasoning layer**. We will:

- Send the user's prompt to Claude API.

- If needed, Claude tells us which tool to call.
- MCP Client executes the tool.
- Claude uses the result.

1.9.2 Example Claude Integration

```
import os, json, requests
from dotenv import load_dotenv
from mcp.client import MCPClient

load_dotenv()
CLAUDE_API_KEY = os.getenv("CLAUDE_API_KEY")

client = MCPClient("http://localhost:8000")

def ask_claude(prompt, tool_result=None):
    headers = {
        "Authorization": f"Bearer {CLAUDE_API_KEY}",
        "Content-Type": "application/json",
    }
    messages = [
        {"role": "system", "content": "You are a helpful AI with access to tools."},
        {"role": "user", "content": prompt}
    ]
    if tool_result:
        messages.append({"role": "system", "content": f"Tool result: {tool_result}"})

    payload = {
        "model": "claude-3-sonnet-20240229",
        "messages": messages
    }

    resp = requests.post("https://api.anthropic.com/v1/messages", headers=headers, json=payloa
    return resp.json()

# Example usage
result = client.invoke("calculator", {"a": 5, "b": 10})
final = ask_claude("What is 5 + 10?", tool_result=result)
print(json.dumps(final, indent=2))
```

1.10 Understanding Tool Orchestration

1.10.1 What Happens Internally

- Claude receives "What is 5 + 10?"
- Decides: need calculator tool.
- MCP Client executes tool.

- MCP Server returns 15.
- Claude uses the result to explain the answer to the user.

1.10.2 Advantages

- Modular system
- Real-time reasoning
- Easier to scale with more tools

1.11 Adding More Tools

1.11.1 How to Add New Tools

- Add a new `@server.capability` in the server file.
- Example: Translator

```
@server.capability("translate")
def translate(text: str, lang: str):
    return {"translated_text": f"Translated '{text}' to {lang}"}
```

- Restart server and test it from client.

1.11.2 Why Modular Tools Are Important

- You can build a **tool library** once.
- Claude can combine multiple tools intelligently.
- Reusability → Faster development.

1.12 Error Handling and Debugging

1.12.1 Common Errors

- Server not running → Client can't connect
- Wrong capability name → 404 error
- Input mismatch → Tool fails

1.12.2 Tips

- Print logs in both server and client
- Validate inputs before sending
- Handle exceptions gracefully

1.13 Security and Best Practices

1.13.1 Security

- Use authentication in production
- Keep Claude API key safe

- Use HTTPS for communication

1.13.2 Best Practices

- Always return structured JSON
- Document every tool clearly
- Keep code modular and clean
- Version your tools if needed

1.14 Real-World Use Cases

1.14.1 Example Applications

- AI personal assistant with live tools
- Automated report generation
- AI + IoT tool integration
- AI connecting to business APIs (CRM, analytics, etc.)
- Multi-agent collaboration systems

1.14.2 How Companies Use MCP

- Secure tool execution
- Centralized AI reasoning
- Flexible system design

1.15 Summary

1.15.1 What We Learned

- What MCP is and why it matters
- How MCP Server and Client work
- How Claude acts as an intelligent agent
- How to build and expose tools
- How to integrate everything together
- Debugging, security, and best practices

1.16 Next Steps — Learning Project

1.16.1 What We'll Build Next

In the next practical notebook, we'll:

- Build a **full working Agentic AI Project**:
  - Claude as the brain
  - MCP Client as the bridge
  - MCP Server with **multiple real tools**

  - Automatic **tool selection & orchestration**

- Add real-world integrations (e.g., weather API, calculator, file tools, task manager).

1.16.2 Goal

To convert this theory into a **practical, working AI system** that responds intelligently to complex queries using multiple tools.

Theory → Practice → Real Agentic AI Project
