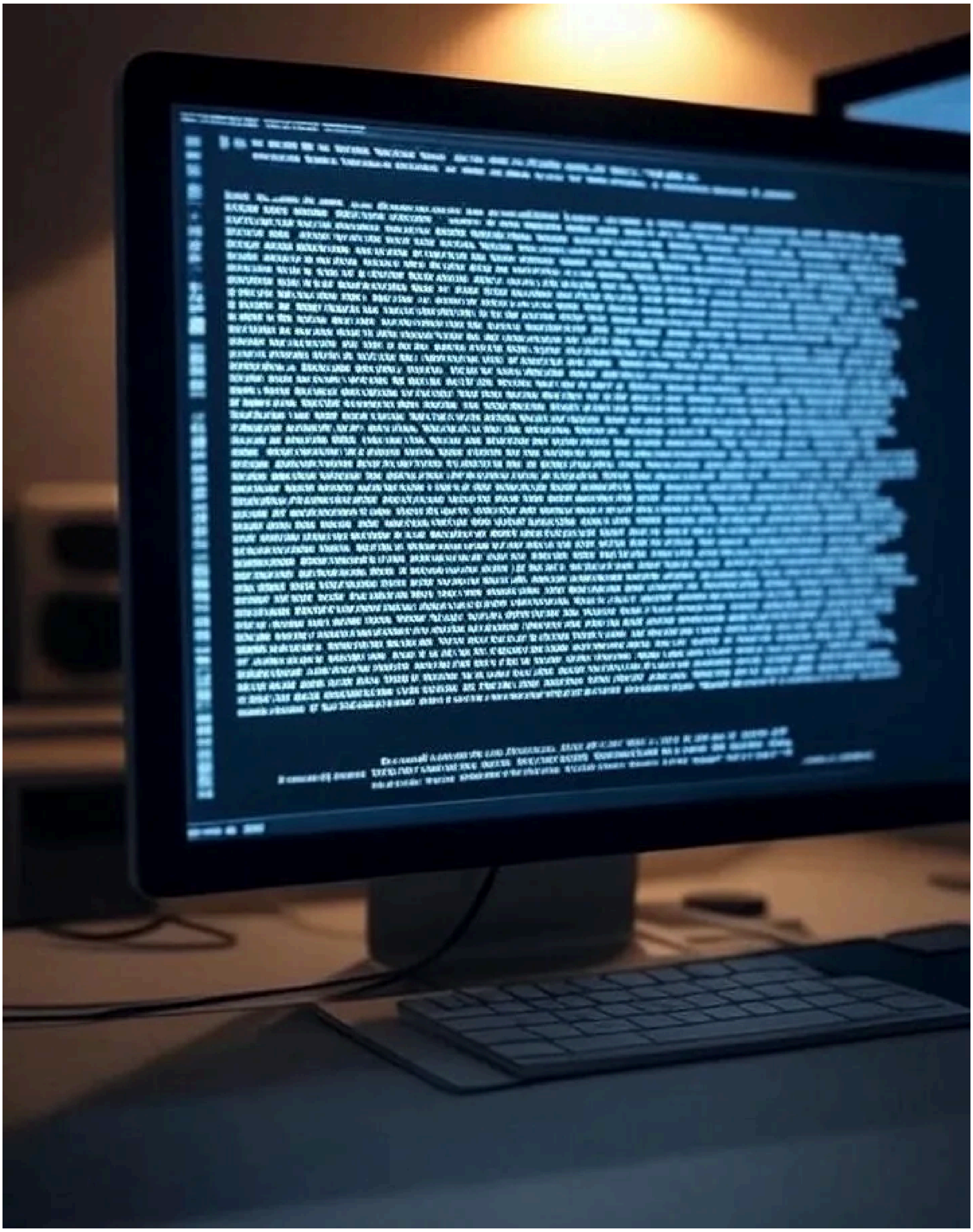
Understanding MCP Stdio transport

6 min read · Jul 20, 2025

Laurent Kubaski Following

This article is part of my MCP Explained article series.



1. Introduction

MCP supports 2 transports for client-server communication: stdio and streamable HTTP.

The specification for the stdio transport is very compact and consists of only 7 points. Anthropic also provides a free Model Context Protocol: Advanced Topics course with a dedicated chapter on the stdio transport.

In this article I want to expand on those two official sources of information, using FastMCP 2.0 for the implementation. I'll refer to this as simply "FastMCP" in this article, but keep in mind that:

- The Python examples on https://modelcontextprotocol.io are using FastMCP 1.0, not FastMCP 2.0.
- FastMCP 2.0 uses FastMCP 1.0 behind the hood.

2. A simple FastMCP client and server

Throughout this article, I'll use this simple FastMCP server and client:

```python
# server.py
from fastmcp import FastMCP

server = FastMCP("Greetings")

@server.tool()
async def greet_user_tool(name:str) -> str:
    result = f"Good day to you, {name}. I trust this message finds you well."
    return result

if __name__ == "__main__":
    server.run(transport="stdio")
```

```python
# client.py
from fastmcp import Client
import asyncio

client = Client(transport="server.py")

async def main():
    async with client:
        result = await client.call_tool(name="greet_user_tool", arguments={"nam
        print(result[0].text)

asyncio.run(main())
```

Note: this is actually not the recommended way to create a MCP tool since I've removed the tool & parameter descriptions to keep things compact.

3. Using stdio for client-server communication?

if you're not familiar with stdin, stdout & stderr, the Wikipedia article titled Standard streams gives a good explanation.

Usually, the stdio, stdout & stderr streams are connected to a text terminal. However when using the MCP stdio transport, **the MCP client launches the MCP server as a subprocess and then communicates with the server using those streams.**

Stdio transport (source: Anthropic)

So leaving aside MCPs for a moment, if you have a child script that looks like this:

```python
# stdio_child.py
name = input() # reads from stdin
print("Hello", name, flush=True) # writes to stdout
```

Then here is how a parent script can launch the child and communicate with it using stdio:

```
# stdio_parent.py
import subprocess

result = subprocess.run(
    ['python3', 'stdio_child.py'],
    input="Laurent", # writes to the child stdin
    capture_output=True,
    text=True
)
print("The child process returned:", result.stdout) # reads from the child stdo
```

Now back to FastMCP: the function that launches the subprocess is _create_platform_compatible_process (source code).

```
async def _create_platform_compatible_process(
    command: str,
    args: list[str],
    env: dict[str, str] | None = None,
    errlog: TextIO = sys.stderr,
    cwd: Path | str | None = None,
):
    if sys.platform == "win32":
        process = await create_windows_process(command, args, env, errlog, cwd)
    else:
        process = await anyio.open_process([command, *args], env=env, stderr=er
    return process
```

4. Stdout is only for MCP messages

So with that said, you now understand why the MCP specification states “*The server MUST NOT write anything to its stdout that is not a valid MCP message*”.

But wait… what if your MCP server tool has a print statement like this:

```
@server.tool()
async def greet_user_tool(name) -> str:
    print(">> << greet_user_tool", flush=True)
```

```
        result = f"Good day to you, {name}. I trust this message finds you well."
        return result
```

Is this going to crash the MCP client since you're now sending a random string to stdout ? Well, it turns out that if you use a FastMCP client, t**his does nothing at all.**

To understand why, you need to look at the stdout_reader() function used by the FastMCP client behind the hood to read the MCP messages sent from the server (source code)

```
    async def stdout_reader():
      [SNIP]
      for line in lines:
          try:
              message = types.JSONRPCMessage.model_validate_json(line)
          except Exception as exc:
              await read_stream_writer.send(exc)
              continue
      [SNIP]
```

If you put a break point on the "for line in lines" line, you'll see all the messages received by the client, including the one from the print statement used by the server.

So here is what's happening: the client receives a message from the server and validates it using Pydantic model_validate_json() : if validation fails — **which happens if the message is not a valid MCP message** — then the exception is sent to read_stream_writer.

We now need to see where that read_stream_writer is read: this happens in the _receive_loop() function (source code)

```
    async def _receive_loop(self) -> None:
        async with (
            self._read_stream,
            self._write_stream,
        ):
            async for message in self._read_stream:
                if isinstance(message, Exception):
                    await self._handle_incoming(message)
    [SNIP]
```

OK so when the client receives a message that is not a valid MCP message, this loop calls _handle_incoming()… and what happens here? Yep, you guessed it, **absolutely nothing.** More precisely, this calls the client default message handler which is empty:

```
async def _default_message_handler(
    message: RequestResponder[types.ServerRequest, types.ClientResult] | types.
) -> None:
    await anyio.lowlevel.checkpoint()
```

Voilà: this explains why nothing goes bad if the client receives a message from the server that is not a valid MCP message. That message is simply ignored by the client.

If you're curious, you can write your own custom message handler and add it to your client when you instantiate it:

```
from fastmcp import Client
import asyncio

async def custom_message_handler(message) -> None:
   print(f"Custom message handler received: {message}")

client = Client(
    transport="server.py",
    message_handler = custom_message_handler
)

async def main():
    async with client:
        result = await client.call_tool(name="greet_user_tool", arguments={"nam
        print(result[0].text)

asyncio.run(main())
```

Now if you run it, you'll see the exception that's raised:

```
Custom message handler received: 1 validation error for JSONRPCMessage
  Invalid JSON: expected value at line 1 column 1 [type=json_invalid, input_val
```

Important: **this behavior is specific to the FastMCP client**. Since the MCP specification explicitly states that "*The server MUST NOT write anything to its stdout that is not a valid MCP message*" then there is no guarantee that every MCP client out there will choose to silently ignore non-valid messages.

5. How do we log then?

Since you cannot log on stdout in the MCP servr, meaning that you're left with 3 options:

5.1. Log to a file

Self-explanatory: instead of logging on the console, you log to a file (or to any location that is not stdout really).

5.2. Log to stderr

Using a print statement:

```python
# server.py
[SNIP]

@server.tool()
async def greet_user_tool(name) -> str:
    print(">> << greet_user_tool", file=sys.stderr)
    result = f"Good day to you, {name}. I trust this message finds you well."
    return result

[SNIP]
```

Or using the get_logger() utility function from FastMCP:

```python
# server.py
import sys

from fastmcp import FastMCP
from fastmcp.utilities.logging import get_logger

logger = get_logger(__name__)
server = FastMCP("Greetings")

@server.tool()
async def greet_user_tool(name) -> str:
    logger.info(">> << greet_user_tool")
    result = f"Good day to you, {name}. I trust this message finds you well."
    return result

if __name__ == "__main__":
    server.run(transport="stdio")
```

If you look at the source code, you'll see that the FastMCP logger logs on stderr:

```python
# logging.py
[SNIP]
handler = RichHandler(
      console=Console(stderr=True),
      rich_tracebacks=enable_rich_tracebacks,
  )
[SNIP]
```

5.3. Delegate to the client

See Server Logging in the FastMCP documentation

6. Bonus: don't forget to flush

Let me paste again the MCP Server tool with the print statement:

```python
@server.tool()
async def greet_user_tool(name) -> str:
    print(">> << greet_user_tool", flush=True)
```

```
        result = f"Good day to you, {name}. I trust this message finds you well."
        return result
```

See "flush=True" ? Initially this wasn't there and I've lost 5 hours of my life trying to understand why the client was never receiving this message. Even Cursor has not been able to figure this out.

Then I discovered the stdout_writer() function that the MCP Server uses the send the MCP messages back to the client (source code)

```
    async def stdout_writer():
        try:
            async with write_stream_reader:
                async for session_message in write_stream_reader:
                    json = session_message.message.model_dump_json(by_alias=True,
                    await stdout.write(json + "\n")
                    await stdout.flush()
        except anyio.ClosedResourceError:
            await anyio.lowlevel.checkpoint()
```

See "stdout.flush()" ? Yep: if you don't do this, then when the MCP client stops, there may still be some messages buffered in the stdout stream… which was the case for me.

7. Further Readings

- Model Context Protocol: Advanced Topics (Anthropic MCP Course)

Mcp Server

Artificial Intelligence

Following

Written by Laurent Kubaski

109 followers · 3 following

I'm a product manager at Salesforce, more info here: https://kubaski.com/

No responses yet

Bgerby

What are your thoughts?

More from Laurent Kubaski

Laurent Kubaski

MCP Resources explained (and how they differ from MCP Tools)

This article is part of my MCP Explained series and a follow up to my MCP Prompts Explained article.

Aug 1 71 3

Laurent Kubaski

Ollama prompt templates

This article is part my “Ollama Explained” series.

Feb 9 17 1

Laurent Kubaski

Connecting your Python MCP Server to Copilot Chat in Visual Studio Code

This article is part of my MCP Explained series.

Jun 20 5

Laurent Kubaski

Ollama endpoints options parameter

This article is part my “Ollama Explained” series.

Mar 16 37 1

See all from Laurent Kubaski

Recommended from Medium

Laurent Kubaski

MCP Explained (article series)

Don’t worry, the Internet is already swamped with generic articles on that topic so I’m not planning to write another one: this article is...

Jun 20 👏 6

Aparna Prasad

🚀 Build Your First MCP Server in TypeScript — Step-by-Step Guide (with Code) 💻

What is MCP (Modal Context Protocol)?

Jul 30 1

Anil Goyal

How to Test your MCP Server using MCP Inspector

The above is my blog on creating a MCP server using python using FastMCP.

Jul 16 5 1

alexeylark

ChatGPT Custom MCP Connectors with Developer Mode

After the release of ChatGPT Custom Connectors with MCP support, we look at what's available to us at the moment.

Sep 20 14

Yash Bhaskar

MCP's Three Core Capabilities: Tools, Resources, and Prompts

Connecting an AI to external software is the key to building powerful applications. But how do you do it in a way that is both flexible and...

Jun 30 7

yulyct

A Beginner’s Guide to Building a Custom MCP Server and Agent Integration

AI is evolving quickly. Every week brings something new, a model, a library, an SDK, or another research paper. Among these rapid…

May 21 51

See more recommendations