MCP Architecture

The JSON-RPC 2.0 message format MCP uses, the stdio and Streamable HTTP transports, the initialize/discover/invoke connection lifecycle, and capability negotiation.

Intermediate 20 min

Last updated: August 22, 2026
Read online: https://www.learnforgex.com/en/topics/mcp-architecture

---

MCP Architecture

"Introduction to MCP" described the three roles -- host, client, and server -- and the three primitives a server can expose. This lesson goes one layer deeper: what actually travels between a client and a server, in what order, and over what kind of connection. None of this changes what was already established -- it's the mechanical layer underneath it, and it's what "Building an MCP Server", this category's final lesson, will exercise directly with real, running code.

From Concepts to Wire Format: JSON-RPC 2.0

Every message a client and server exchange is a **JSON-RPC 2.0** message -- a small, well-established, text-based format for making requests and getting responses, unrelated to MCP itself (MCP simply adopted it as its message format). A request names a **method** and supplies **params**; a response carries a matching **id** and either a **result** or an error. When "Introduction to MCP" described a client asking a server to list its tools, that request and response look like this on the wire:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "get_capital_city",
    "arguments": { "country": "Japan" }
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "content": [
      { "type": "text", "text": "The capital of Japan is Tokyo." }
    ]
  }
}
```

What matters for this lesson is that MCP doesn't invent its own message format -- it uses JSON-RPC 2.0 as the wire format. Most operations, like `tools/list`, `tools/call`, and `resources/read`, follow this request/response shape (some message types in the specification, such as one-way *notifications* that expect no response, fall outside that request/response pair, but this lesson doesn't need to go into their detail). This is the layer "Building an MCP Server"'s SDK code hides behind ordinary function calls -- but it's genuinely what's being sent underneath.

Transports: stdio and Streamable HTTP

JSON-RPC messages need an actual channel to travel over -- MCP calls this the **transport**, and it's deliberately kept separate from everything above it, so the same server logic works no matter which transport carries it. Two are most common: **stdio**, where the client launches the server as a local subprocess and messages travel over its standard input/output -- typical for a desktop AI application or IDE talking to a tool running on the same machine -- and **Streamable HTTP**, where the server runs as an independent, possibly remote, process reachable over HTTP, letting one server be shared by multiple clients or hosts at once. Which transport is in use is invisible to the tool-calling logic itself -- a tool's `name`, `description`, and behavior are identical either way.

The Connection Lifecycle: Initialize, Discover, Invoke

What follows isn't a complete inventory of every possible message type in the MCP protocol -- it's a deliberately simplified base lifecycle, enough to understand tool/resource usage at this lesson's level. Regardless of transport, an MCP connection goes through these three phases:

1. **Initialize** -- the client and server exchange an `initialize` request and response before anything else happens, agreeing on a protocol version and stating what each side supports (see "Capability Negotiation" below).
2. **Discover** -- the client asks what's available: `tools/list` for tools, `resources/list` for resources, `prompts/list` for prompts, as introduced conceptually in "Introduction to MCP". The server responds with each item's name, description, and schema.

3. **Invoke** -- the client sends a request to actually use something -- `tools/call` to run a tool, `resources/read` to fetch a resource -- and the server performs the real work and returns the result, exactly as "MCP and the Tool-Calling Loop" described conceptually.

A single connection typically runs one initialize phase, then repeats discovery and invocation many times over its lifetime.

Capability Negotiation

During initialize, the client and server don't just agree on a protocol version -- they each state which optional features they actually support (for example, whether a server supports resources at all, or whether a client can receive certain kinds of notifications). This is called **capability negotiation**, and it exists because not every host or server needs every MCP feature: a minimal server that only exposes tools doesn't need to implement resource support, and a client only needs to prepare for the capabilities a given server actually declares, rather than every capability the protocol could theoretically support.

Where Our Hands-On Example Fits (In-Memory Transport)

"Building an MCP Server" builds a real client and server using the official TypeScript SDK, and connects them with an **in-memory transport** -- both sides run in the same process, exchanging the exact same JSON-RPC messages described above, just without stdio or a network socket in between. This is a deliberate simplification for a self-contained, runnable lesson: a production setup almost always uses stdio or Streamable HTTP, with the server as a genuinely separate process. The initialize/discover/invoke lifecycle, the message shapes, and the tool-calling behavior are identical either way -- only the transport underneath changes.

Best Practices

- Don't assume a specific transport when reasoning about MCP behavior -- as "Transports: stdio and Streamable HTTP" showed, the same server logic and message shapes apply whether the connection is local (stdio) or remote (Streamable HTTP).
- When a connection misbehaves, check which lifecycle phase it's in (see "The Connection Lifecycle: Initialize, Discover, Invoke") -- a failure during `initialize` is a very different problem than one during a `tools/call` invocation.
- Rely on capability negotiation rather than assuming a server supports every MCP feature -- a client should check what a server actually declared during initialize before trying to use it (see "Capability Negotiation").

Common Mistakes

- **Thinking JSON-RPC is something MCP invented.** As "From Concepts to Wire Format: JSON-RPC 2.0" explained, JSON-RPC 2.0 is a pre-existing, general-purpose message format -- MCP adopted it rather than designing a new one.
- **Assuming an in-memory transport is how MCP is normally deployed.** "Where Our Hands-On Example Fits (In-Memory Transport)" was chosen for this course specifically to keep the example self-contained -- real deployments almost always use stdio or Streamable HTTP, with the server as a genuinely separate process.
- **Skipping straight to invocation while debugging a connection.** As "The Connection Lifecycle: Initialize, Discover, Invoke" laid out, initialize and discovery happen first -- an invocation failure often has its real cause in one of those earlier phases.

Summary, Cheat Sheet, and Glossary

Summary

- MCP doesn't invent its own message format -- it uses JSON-RPC 2.0 as its wire format, a pre-existing, general-purpose one.
- A **transport** (commonly stdio for local processes, or Streamable HTTP for remote ones) carries those messages; tool-calling behavior is identical regardless of which one is used.
- Every connection goes through the same lifecycle: **initialize** (agree on protocol version and capabilities), **discover** (`tools/list` and similar), and **invoke** (`tools/call` and similar).
- **Capability negotiation**, during initialize, lets each side declare only the optional features it actually supports.
- This course's hands-on example uses an **in-memory transport** to stay self-contained -- the same lifecycle and message shapes apply as they would over stdio or Streamable HTTP.

Cheat Sheet

- Wire format = JSON-RPC 2.0 (request: method + params + id; response: id + result or error).
- Transport = stdio (local subprocess) or Streamable HTTP (remote, shareable).
- Lifecycle = initialize -> discover (`tools/list` , `resources/list` , `prompts/list`) -> invoke (`tools/call` , `resources/read`).
- Capability negotiation = each side declares supported optional features during initialize.

Glossary

- **JSON-RPC 2.0:** the pre-existing, general-purpose, text-based message format MCP uses as its wire format for client-server exchanges.

- **Transport:** the actual channel JSON-RPC messages travel over -- commonly stdio (local subprocess) or Streamable HTTP (remote).
- **stdio transport:** a transport where the client launches the server as a local subprocess and messages travel over standard input/output.
- **Streamable HTTP transport:** a transport where the server runs as an independent process reachable over HTTP.
- **Initialize phase:** the first phase of any MCP connection, where client and server agree on a protocol version and declare supported capabilities.
- **Capability negotiation:** each side declaring, during initialize, which optional MCP features it actually supports.
- **In-memory transport:** a transport where client and server run in the same process, used in this course to keep the hands-on example self-contained.