Concepts

Core architecture

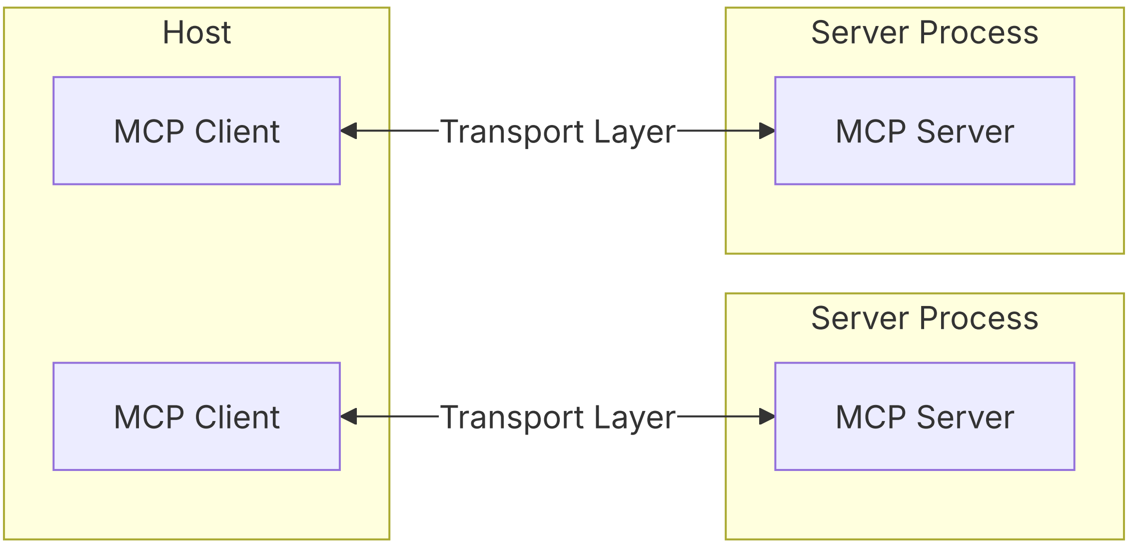
Understand how MCP connects clients, servers, and LLMs

The Model Context Protocol (MCP) is built on a flexible, extensible architecture that enables seamless communication between LLM applications and integrations. This document covers the core architectural components and concepts.

Overview

MCP follows a client-server architecture where:

- **Hosts** are LLM applications (like Claude Desktop or IDEs) that initiate connections
- **Clients** maintain 1:1 connections with servers, inside the host application
- **Servers** provide context, tools, and prompts to clients



Core components

Protocol layer

The protocol layer handles message framing, request/response linking, and high-level communication patterns.

TypeScript Python

```
class Protocol<Request, Notification, Result> {
    // Handle incoming requests
    setRequestHandler<T>(schema: T, handler: (request: T, extra: R

    // Handle incoming notifications
    setNotificationHandler<T>(schema: T, handler: (notification: T)

    // Send requests and await responses
    request<T>(request: Request, schema: T, options?: RequestOptio

    // Send one-way notifications
    notification(notification: Notification): Promise<void>
}
```

Key classes include:

- `Protocol`
- `Client`
- `Server`

Transport layer

The transport layer handles the actual communication between clients and servers. MCP supports multiple transport mechanisms:

1. **Stdio transport**

Uses standard input/output for communication

Ideal for local processes

2. **HTTP with SSE transport**

   Uses Server-Sent Events for server-to-client messages

   HTTP POST for client-to-server messages

All transports use **JSON-RPC** 2.0 to exchange messages. See the **specification** for detailed information about the Model Context Protocol message format.

Message types

MCP has these main types of messages:

1. **Requests** expect a response from the other side:

```
interface Request {
  method: string;
  params?: { ... };
}
```

2. **Results** are successful responses to requests:

```
interface Result {
  [key: string]: unknown;
}
```

3. **Errors** indicate that a request failed:

```
interface Error {
  code: number;
  message: string;
```

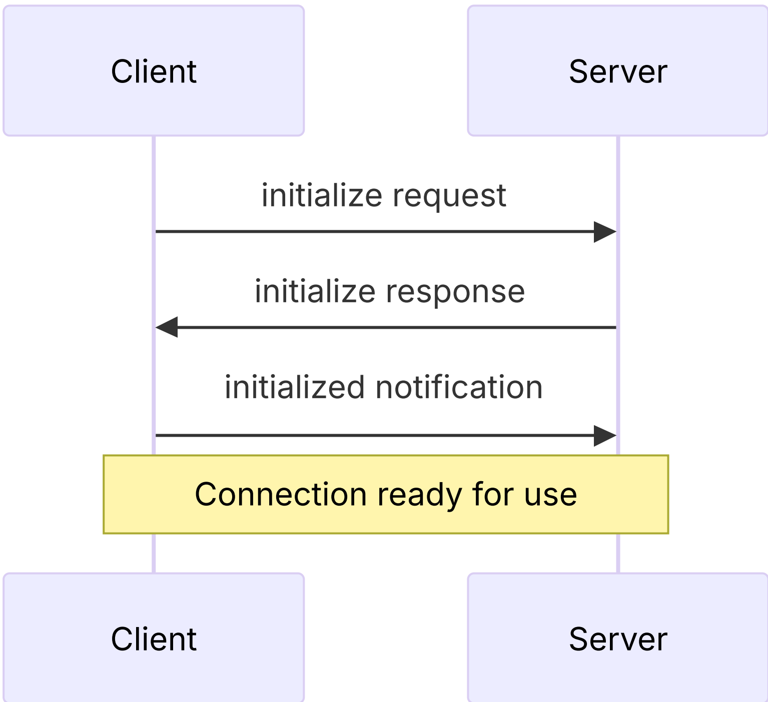
```
  data?: unknown;
}
```

4. **Notifications** are one-way messages that don't expect a response:

```
interface Notification {
  method: string;
  params?: { ... };
}
```

Connection lifecycle

1. Initialization



1. Client sends `initialize` request with protocol version and capabilities
2. Server responds with its protocol version and capabilities
3. Client sends `initialized` notification as acknowledgment
4. Normal message exchange begins

2. Message exchange

After initialization, the following patterns are supported:

- **Request-Response**: Client or server sends requests, the other responds
- **Notifications**: Either party sends one-way messages

3. Termination

Either party can terminate the connection:

- Clean shutdown via `close()`
- Transport disconnection
- Error conditions

Error handling

MCP defines these standard error codes:

```
enum ErrorCode {
  // Standard JSON-RPC error codes
  ParseError = -32700,
  InvalidRequest = -32600,
  MethodNotFound = -32601,
  InvalidParams = -32602,
  InternalError = -32603
}
```

SDKs and applications can define their own error codes above -32000.

Errors are propagated through:

- Error responses to requests
- Error events on transports
- Protocol-level error handlers

Implementation example

Here's a basic example of implementing an MCP server:

TypeScript Python

```
import { Server } from "@modelcontextprotocol/sdk/server/index.js"
import { StdioServerTransport } from "@modelcontextprotocol/sdk/se

const server = new Server({
  name: "example-server",
  version: "1.0.0"
}, {
  capabilities: {
    resources: {}
  }
});

// Handle requests
server.setRequestHandler(ListResourcesRequestSchema, async () => {
  return {
    resources: [
      {
        uri: "example://resource",
        name: "Example Resource"
      }
    ]
  };
});

// Connect transport
const transport = new StdioServerTransport();
await server.connect(transport);
```

Best practices

Transport selection

1. **Local communication**
   - Use stdio transport for local processes
   - Efficient for same-machine communication
   - Simple process management
2. **Remote communication**
   - Use SSE for scenarios requiring HTTP compatibility
   - Consider security implications including authentication and authorization

Message handling

1. **Request processing**
   - Validate inputs thoroughly
   - Use type-safe schemas
   - Handle errors gracefully
   - Implement timeouts
2. **Progress reporting**
   - Use progress tokens for long operations
   - Report progress incrementally
   - Include total progress when known
3. **Error management**
   - Use appropriate error codes
   - Include helpful error messages
   - Clean up resources on errors

Security considerations

1. **Transport security**
   - Use TLS for remote connections
   - Validate connection origins
   - Implement authentication when needed
2. **Message validation**
   - Validate all incoming messages
   - Sanitize inputs
   - Check message size limits
   - Verify JSON-RPC format
3. **Resource protection**
   - Implement access controls
   - Validate resource paths
   - Monitor resource usage
   - Rate limit requests
4. **Error handling**
   - Don’t leak sensitive information
   - Log security-relevant errors
   - Implement proper cleanup
   - Handle DoS scenarios

Debugging and monitoring

1. **Logging**
   - Log protocol events
   - Track message flow
   - Monitor performance
   - Record errors

2. **Diagnostics**

    - Implement health checks
    - Monitor connection state
    - Track resource usage
    - Profile performance

3. **Testing**

    - Test different transports
    - Verify error handling
    - Check edge cases
    - Load test servers

Was this page helpful?

Yes No

< Inspector

Resources >