

Concepts

Transports

Learn about MCP's communication mechanisms

Transports in the Model Context Protocol (MCP) provide the foundation for communication between clients and servers. A transport handles the underlying mechanics of how messages are sent and received.

Message Format

MCP uses **JSON-RPC 2.0** as its wire format. The transport layer is responsible for converting MCP protocol messages into JSON-RPC format for transmission and converting received JSON-RPC messages back into MCP protocol messages.

There are three types of JSON-RPC messages used:

Requests

```
{
  jsonrpc: "2.0",
  id: number | string,
  method: string,
  params?: object
}
```

Responses

**Model Context Protocol**

```

{
  jsonrpc: "2.0",
  id: number | string,
  result?: object,
  error?: {
    code: number,
    message: string,
    data?: unknown
  }
}

```

Notifications

```

{
  jsonrpc: "2.0",
  method: string,
  params?: object
}

```

Built-in Transport Types

MCP includes two standard transport implementations:

Standard Input/Output (stdio)

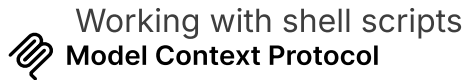
The stdio transport enables communication through standard input and output streams. This is particularly useful for local integrations and command-line tools.

Use stdio when:

- Building command-line tools

- Implementing local integrations

- Needing simple process communication



TypeScript (Server) TypeScript (Client) Python (Server) Python (Client)

~~Concepts~~ > **Transports**

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

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

Server-Sent Events (SSE)

SSE transport enables server-to-client streaming with HTTP POST requests for client-to-server communication.

Use SSE when:

Only server-to-client streaming is needed

Working with restricted networks

Implementing simple updates

TypeScript (Server) TypeScript (Client) Python (Server) Python (Client)

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

```
const transport = new SSEServerTransport("/message", response);  
await server.connect(transport);
```



Concepts > **Transports**

Custom Transports

MCP makes it easy to implement custom transports for specific needs. Any transport implementation just needs to conform to the Transport interface:

You can implement custom transports for:

Custom network protocols

Specialized communication channels

Integration with existing systems

Performance optimization

TypeScript Python

```
interface Transport {  
  // Start processing messages  
  start(): Promise<void>;  
  
  // Send a JSON-RPC message  
  send(message: JSONRPCMessage): Promise<void>;  
  
  // Close the connection  
  close(): Promise<void>;  
  
  // Callbacks  
  onclose?: () => void;  
  onerror?: (error: Error) => void;  
  onmessage?: (message: JSONRPCMessage) => void;  
}
```

Error Handling

Model Context Protocol

Transport implementations should handle various error scenarios:

Concepts > **Transports**

1. Connection errors
2. Message parsing errors
3. Protocol errors
4. Network timeouts
5. Resource cleanup

Example error handling:

TypeScript **Python**

```
class ExampleTransport implements Transport {
  async start() {
    try {
      // Connection logic
    } catch (error) {
      this.onerror?.(new Error(`Failed to connect: ${error}`));
      throw error;
    }
  }

  async send(message: JSONRPCMessage) {
    try {
      // Sending logic
    } catch (error) {
      this.onerror?.(new Error(`Failed to send message: ${error}`));
      throw error;
    }
  }
}
```

Best Practices

Model Context Protocol

When implementing or using MCP transport:

Concepts > **Transports**

1. Handle connection lifecycle properly
2. Implement proper error handling
3. Clean up resources on connection close
4. Use appropriate timeouts
5. Validate messages before sending
6. Log transport events for debugging
7. Implement reconnection logic when appropriate
8. Handle backpressure in message queues
9. Monitor connection health
10. Implement proper security measures

Security Considerations

When implementing transport:

Authentication and Authorization

Implement proper authentication mechanisms

Validate client credentials


Use secure token handling

Implement authorization checks

Data Security

Use TLS for network transport

Encrypt sensitive data

 Validate message integrity
Model Context Protocol
Implement message size limits

Sanitize input data
Concepts → Transports

Network Security

- Implement rate limiting
- Use appropriate timeouts
- Handle denial of service scenarios
- Monitor for unusual patterns
- Implement proper firewall rules

Debugging Transport

Tips for debugging transport issues:

1. Enable debug logging
2. Monitor message flow
3. Check connection states
4. Validate message formats
5. Test error scenarios
6. Use network analysis tools
7. Implement health checks
8. Monitor resource usage
9. Test edge cases
10. Use proper error tracking

Was this page helpful?

 Yes

 No



[< Roots](#)
[Concepts](#) > **Transports**

Roadmap >