Model Context Protocol (MCP) Implementation in Go

This project demonstrates an implementation of the Model Context Protocol (MCP) in Go, showcasing how clients can communicate with a server over standardized JSON-RPC interfaces, without using third party libraries.

Project Overview

The Model Context Protocol enables structured communication between clients and servers, particularly useful for AI model interactions. This implementation consists of two main components:

- **MCP Server**: A process that runs in the background, exposing capabilities through a JSON-RPC interface
- **MCP Client**: A process that communicates with the server through stdin/stdout pipes

The communication happens over standard input/output streams, with messages formatted according to the JSON-RPC 2.0 specification.

Table of Contents

- Features
- Project Structure
- MCP Server
- MCP Client
- Building and Running
- Protocol Details
- Example Usage
- Security Considerations

Features

- **JSON-RPC 2.0 Implementation**: Full implementation of the JSON-RPC 2.0 specification
- **Bidirectional Communication**: Communication over stdin/stdout pipes
- **Robust Error Handling**: Comprehensive error handling for network issues, timeouts, and more
- **Graceful Shutdown**: Clean termination of processes with proper resource cleanup
- **Logging**: Detailed logging for debugging and monitoring
- **Extensible Design**: Easy to add new capabilities to the server

Project Structure

```
mcp/
├── README.md           # This file
├── mcp-client/         # Client implementation
│   ├── main.go         # Client code
│   └── Makefile        # Build instructions for client
└── mcp-server/         # Server implementation
```

```
    ├── main.go         # Server code
    └── Makefile        # Build instructions for server
```

Model Context Protocol

Workflow

An MCP-capable client acts as a bridge between an LLM and an MCP server. Here’s how this workflow typically functions:

- Client-Server Interaction: The MCP client connects to the MCP server to access its capabilities, such as tools, resources, or prompts. These capabilities can include querying databases, accessing APIs, or retrieving structured data.
- LLM Context Integration: When the LLM processes a request and determines it needs external data or tools, it communicates with the MCP client embedded in its host application. The client forwards the request to the MCP server, retrieves the response, and integrates this information into the LLM's context.
- LLM Isolation: The LLM itself does not directly access the MCP server. Instead, it relies on the MCP client to handle communication with the server and manage responses. This design ensures modularity and security by isolating the LLM from direct server interactions.
- Workflow Example: For instance, if a user asks an LLM about weather data, the LLM identifies it needs real-time information from an external source. The MCP client sends a request to the MCP server for weather data, retrieves the result, and injects it into the LLM's context . It is up to the MCP client to analyze a request for data from the LLM and convert that to a request to the appropriate MCP server. Typically the client prompts will inform the LLM of the available MCP resources and ask for results in some form, such as free-flow or JSON.

```
Available tools: get_forecast(lat, lon), get_alerts(state)
Respond with JSON: {"tool": "get_forecast", "params": {...}}
```

Client - Server Connection

An MCP client can connect to an MCP server by either of two methods:

- STDIO transport
  - This is the default transport mechanism that uses standard input/output streams for communication. The server must be an executable on the local machine where the client can access it. The client will start the server as a subprocess and use an STDIO connection to communicate with the server.
- HTTP Stream
  - This transport implements streamable HTTP for network-based communication and is compliant with the latest MCP specifications.

This implementation uses the default STDIO transport.

MCP Server

The MCP server is a Go application that:

1. Initializes and exposes capabilities via JSON-RPC over stdio
2. Listens for incoming requests on stdin
3. Processes requests and returns responses on stdout
4. Handles proper initialization and shutdown

Server Capabilities

As an example, the server implements the following capabilitie:

- **RandomString**: Generates cryptographically secure random strings with configurable length
- More capabilities can be easily added by extending the `MCPService` struct

Key Server Components

- **MCPService**: Implements the service methods that clients can call
- **LoggingServerCodec**: Custom JSON-RPC codec with enhanced logging
- **Signal Handling**: Proper handling of termination signals for graceful shutdown
- **Request Processing**: Thread-safe processing of client requests

MCP Client

The MCP client is a Go application that can be used to test the server:

1. Launches the MCP server as a subprocess
2. Communicates with the server over stdin/stdout pipes
3. Provides a clean API for calling server capabilities
4. Handles errors and timeouts robustly

Client Features

- **Command-line Arguments**: Support for specifying the server path
- **Automatic Initialization**: Handles the server initialization protocol
- **Capability Discovery**: Automatically discovers server capabilities
- **Timeout Handling**: Prevents hanging if the server is unresponsive
- **Clean Shutdown**: Ensures the server subprocess is properly terminated

Key Client Components

- **MCPClient**: Main struct that manages the server process and communication
- **JSON-RPC Handling**: Implements the client side of the JSON-RPC 2.0 protocol
- **ID Validation**: Ensures responses match their corresponding requests
- **EOF Handling**: Properly handles unexpected server termination

Building and Running

Prerequisites

- Go 1.18 or later

Makefile Build

- cd to top level project directory
- make build
  - builds both executables
- make clean
  - removes binaries

Building the Server

```
cd mcp-server
go build -o mcp-server .
```

Building the Client

```
cd mcp-client
go build -o mcp-client .
```

Running the Client

```
./mcp-client
```

With an optional custom server path. The default is ../bin/mcp-server, if you are running the client from its location in the project.

```
./mcp-client -server /path/to/custom/mcp-server
```

Protocol Details

Initialization

1. Client launches the server subprocess
2. Server sends an initialization message with its capabilities
3. Client processes the initialization message and stores capabilities

Request-Response Cycle

1. Client sends a JSON-RPC request to the server's stdin
2. Server processes the request and executes the requested method
3. Server sends a JSON-RPC response to stdout

4. Client reads the response and validates it

Message Format

All messages follow the JSON-RPC 2.0 specification:

Request

```
{
  "jsonrpc": "2.0",
  "method": "MCPService.RandomString",
  "params": {"Length": 20},
  "id": 1
}
```

Response

```
{
  "jsonrpc": "2.0",
  "result": {"Result": "3f7ac68z1xPq9dYh5w"},
  "id": 1
}
```

Example Usage

Here's a simple example of using the client API:

```
// Create a new client
client, err := NewMCPClient("/path/to/mcp-server")
if err != nil {
    log.Fatalf("Failed to create client: %v", err)
}
defer client.Close()

// Initialize the client
initResp, err := client.Initialize()
if err != nil {
    log.Fatalf("Failed to initialize: %v", err)
}

// Check if the RandomString capability is available
if client.HasCapability("RandomString") {
    // Call the RandomString method
    randomStr, err := client.RandomString(20)
    if err != nil {
        log.Fatalf("Failed to generate random string: %v", err)
    }
```

```
    fmt.Printf("Random string: %s\n", randomStr)
}
```

Security Considerations

- **Input Validation**: The server validates all inputs to prevent abuse
- **Maximum String Length**: Random string generation has a maximum length to prevent DoS attacks
- **Unbiased Random Generation**: Using cryptographically secure randomness with rejection sampling
- **Proper Resource Cleanup**: Ensures resources are released even during abnormal termination
- **Timeout Handling**: Prevents hanging in case of unresponsive components

---

This project serves as both a functional implementation and an educational resource for understanding client-server communication patterns, JSON-RPC, and Go's concurrency features.