The Evolving MCP Design

Paul S. Wang
Institute for Computational Mathematics
Kent State University

pwang@mcs.kent.edu

Contents

MCP Issues and Considerations

- Supporting IAMC: mathematical data encodings, computation controls
- Simple, powerful, and flexible
- Meeting client-to-server and server-to-client requirements
- Using different types of data-transfer encodings
- Employing stream connection between client and server
- Assuming peer-to-peer interactions
- Charging, authentication, security (usual)
- Object-oriented design/implementation (java?)

Client Requirements

- Sending computation commands
- Receiving computational results, in various forms of encoding
- Store results in files and resend stored results to other servers later
- Sending control commands to server
- Receiving control responses (e.g. server status information)
- Sending client status
- Receiving commands from server
- Responding to commands from server
- Issuing commands both synchronously (waiting for result before next command) and asynchronously (may issue next command without waiting for result of previous command)

- Receiving results out of order (A result identifies its command)
- Interrupting an on-going computation (may be several on-going)
- Disconnecting (immediately or after all results are in)

Server Requirements

- Receiving computation requests, in *literal ascii string* or *standard encoded* forms
- Sending computed results, in various encoded forms (MathML, MP, OpenMath, gif, pdf, ...)
- Handling both sync and async requests
- Sending control requests to client
- Receiving control responses (e.g. client status information)
- Sending server status
- Sending results back as they are produced
- Responding to control requests from client
- Disconnecting

MCP Protocol

MCP message format:

- First line:

  `Command MCP` *version*
  `Control MCP` *version*
  `Result MCP` *version*

- Header and body format (HTTP style)
- Header Key-value pairs (HTTP style)
- Consider features of HTTP-NG with session layer

Some Available Hearders

This will have to be consdered carefully and will evolve

- `Status:` normal, error *no*, ready, busy, terminating
- `ControlRequest:` *string*
- `ControlResponse:` *string*
- `Accept:` types of result
- `CanDo:` computing capabilities
- `EngineID:` compute engine identification
- `ClientID:` IAMC client identification
- `Mode:` `sync` or `aync`
- `Sequence:` linear sequence number to match results with commands

- `Content-Type:` body type (e.g. `application/x-math-mp`, or `text/MathML`) default `text/plain`
- `Content-length:` *bytes*
- `Transfer-Encoding:` if any, default none (useful for email transfer)

Example Client Computation Request

```
Command MCP 1.0
Mode: sync                           (or async, default sync)
Sequence: 1
Content-Type: application/x-math-mp  (or -infix, or -openmath)
<<Content-Type: application/x-math-ascii>>
Content-length: 356

    Body
```

Example Server Computation Response

```
Result MCP 1.0
Status: normal
Sequence: 1
Content-Type: application/x-math-mp
Content-length: 4000

Body
```


Example Client Control Request

```
Control MCP 1.0
ControlRequest: connect
Accept: mp,MathML
ClientID: XXXX
```

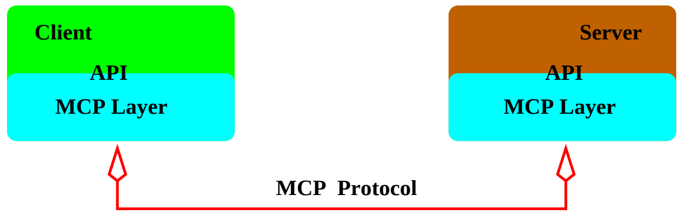
Example Server Control Response

```
Control MCP 1.0
ControlResponse: connect-confirmation
CandDo: factoring,differentiation,integration,plotting
EngineID: Maxima
```


Server to Client Control Requests

- Prompts for input from the user
- Choices for user/client selection
- Dialogue for user/client input
- Setting Cookie
- Disconnecting

MCP Layer Interfaces



Java Classes

- These will emerge as the OOD for MCP gets going. But for API considerations we see two classes:
  - `Mcp` class
  - `McpMsg` class

MCP Java API for IAMC Server

- `public Mcp(String canDo, String Accept)`
  Constructor. Initializes an object, in class Mcp, to receive input from standard input (`System.in`) and produce output to standard output (`System.out`). The standard I/O streams will be used for binary I/O (reading/writing bytes).
- `public Mcp(String canDo, String Accept, InputStream in, OutputStream out)`
  Constructor. Initializes an Mcp object to perform I/O with the given streams.
- `McpMsg getCommand()`
  Retrieves the next command from client as an `McpMsg`.
- `boolean putResult(McpMsg m)`
  Sends the computational result packed in the message m to the client.

- `McpMsg getAnswer(McpMsg question)`
  Retrieves the answer from client to the given question.
- `void ready(Boolean flag)`
  Indicating to client that server is ready for additional work (after aborting for example) or is not ready (engine is down for example).
- `void terminate()`
  Indicating to client that server is finished and disconnecting.
- `void pingclient()`
  Requesting client status. Assumes clinet is dead after a preset timeout interval.
- `String[] acceptList()`
  Returning the *accept* list from the client.

MCP Java API for IAMC Client

- `public Mcp(String Accept, String server)`
  Constructor. Constructor. Initializes an Mcp object to connect to the given server (domain name).
- `public Mcp(String Accept, int server)`
  Constructor. Constructor. Initializes an Mcp object to connect to the given server (IP address).
- `boolean syncCommand(McpMsg cmd)`
  Sends the command, given as an **`McpMsg`**, to the server in synchronous mode.
- `int asyncCommand(McpMsg cmd)`
  Sends the command, given as an **`McpMsg`**, to the server in asynchronous mode. Returns the sequence number as id for the command.
- `McpMsg getMsg()`
  Retrieves the computational result, a question from the server.

- `boolean abort([int i])()`
  Aborts the on-going **`syncCommand()`** call or the indicated **`asyncComand()`**. This also sends a control msg to the server to abort the target computation. Returns **`true`** on server acknowdedgement.
- `void terminate()`
  Indicating to server that client is finished and disconnecting.
- `void pingserver()`
  Requesting server status. Assumes server is dead after a preset timeout interval.
- `String[] candoList()`
  Returning the *can-do* list from the server.

Further Work

- More thorough requirement analysis
- OOD of MCP Layer
- Testing the design agast requirements
- OOP of Java-based MCP Layer
- Implementing and testing MCP Layer prototype