Technical Assessment Documentation

KMBS Solutions Engineering Center

Version: 547a96ce (2020-01-27)

Contents

Online Version

You can find the most up-to-date version of this documentation, as well as a live demo of the application, on the Technical Assessment site. By opening your browser's JavaScript console on the Live Demo page, you can view an example of the client/server protocol.

Intro

In the following documentation, all file paths are relative to the `technical-assessment/` directory contained in the Technical Assessment Bundle.

Requirements

You will implement a game server (feel free to skip ahead to the rules). We have provided a web client to render the game and manage user interactions. Your server will communicate with the client through an API. The client is dumb and knows nothing about the game. It will be your responsibility to implement the game logic and maintain the game state. You can implement the server in any language you choose.

We have exposed three APIs to communicate with the Client (each API targets a different set of programming languages). You will find a full specification for each of the three APIs in the corresponding documentation. We have strived to ensure that the specification is clear, but if you are confused, feel free to request clarification. Regardless of which API you choose, your game server will be analyzed by a comprehensive test suite that does know the rules of the game - and all known corner cases. Your submission is not immediately discounted if it fails one or two of the tests in the suite; Your application will also be reviewed by several pairs of human eyes to assess code quality (microcosm) and application design (macrocosm).

You may provide your code to us by whatever means is most convenient. Along with your code, you should provide a README file with attribution for any external libraries or code as well as any specific instructions for running your application. You are welcome, but not required, to host your application.

The Game

A connect-the-dots game for two players.

Definitions

- octilinear line - a horizontal, vertical, or 45° diagonal line

Rules

- The game is played on a 4x4 grid of 16 nodes.
- Players take turns drawing octilinear lines connecting nodes.
- Each line must begin at the start or end of the existing path, so that all lines form a continuous path.
- The first line may begin on any node.
- A line may connect any number of nodes.
- Lines may not intersect.
- No node can be visited twice.
- The game ends when no valid lines can be drawn.
- The player who draws the last line is the loser.

Example Game

Each move is numbered. Lines that connect more that two nodes have each segment numbered. Player 1 made the odd numbered moves and Player 2 made the even numbered moves. Player 1 made the first move (1) and was forced to make the last move (9). Thus, Player 2 won.

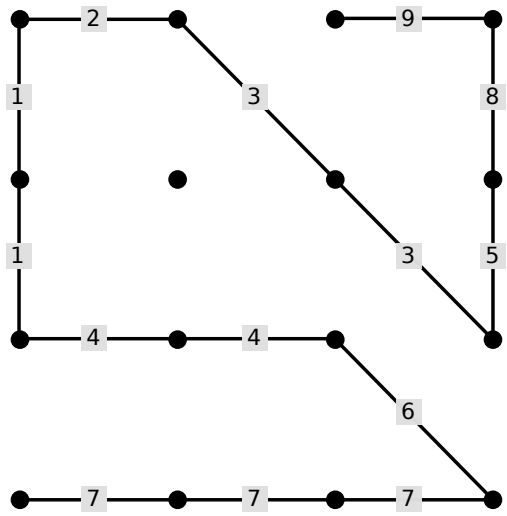

Figure 1: Example Game

Attribution

The game was designed by Sid Sackson.

The Client

To reiterate, the Client does *not* know the rules of the game. It is, effectively, a tool to plot lines on a grid. The grid is represented by quadrant IV of the

Cartesian plane, with the caveat that the indices of the y-axis are positive. Example:

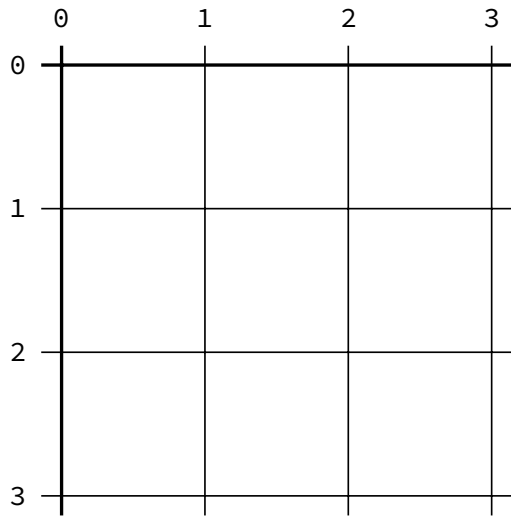

Figure 2: Cartesian Plane

The Client is configured for a 4x4 grid, but larger or smaller sizes are possible. Nodes on the grid are represented by a pair of Cartesian coordinates. Lines are represented by a starting node and an ending node. The Client has no conception of a continuous path; It blindly renders any line that it is given.

Additionally, the Client provides two text areas for displaying useful information to the user - a heading and a generic message field. The Client will never write any values to these text areas on its own; They are provided solely for your use.

The APIs

Use the following guide to help choose the right API for you.

- If you want to use JavaScript or a language that compiles to JavaScript: Use the Client API.
- If you want to use another language (or Node.js), and your language supports the WebSocket protocol: Use the WebSocket API.
- If your favorite language doesn't have WebSocket support: Use the HTTP API.

The APIs provide nearly identical functionality and only differ meaningfully in network latency.

- Client API : No network latency
- WebSocket API : Mitigated network latency

- HTTP API : Significant network latency (for a real-time game). Doesn't support HTTP/2 streaming.

Your API choice will have no bearing whatsoever on our judgement of your solution.

How to Get Started

Once you have decided which API you are going to use, you *must* edit the second argument of the `Elm.Main.embed` function in the `client/init.js` file included in this bundle to set the correct API. If you are using the WebSocket or HTTP APIs, you *must* provide the Client with the address of your server. There are three valid configuration options (listed below). The spelling and case of the `api` field *must* be exactly as specified below. If your API requires the `hostname` field, you *must* omit the trailing slash.

Client API

```
const app = Elm.Main.embed(node, {
    api: 'Client',
    hostname: '',
});
```

HTTP API

```
const app = Elm.Main.embed(node, {
    api: 'Http',
    hostname: 'http://localhost:8080',
});
```

WebSocket API

```
const app = Elm.Main.embed(node, {
    api: 'WebSocket',
    hostname: 'ws://localhost:8081',
});
```

If you are using the Client API, ensure that your code is loaded after `client/init.js`, and open `client/index.html` in your browser. If you are using the WebSocket or HTTP APIs, start your server, and open `client/index.html` in your browser.

Client API Protocol

Types

The API uses JSON as a messaging format. Because JSON does not provide a means of specifying compound types, the following ad hoc type definitions are

provided.

The pipe character | represents logical `OR`.

JSON Primitives

```
string : STRING
number : INTEGER | FLOAT
boolean : BOOLEAN
null : NULL
```

POINT

```
{
    "x": "INTEGER",
    "y": "INTEGER"
}
```

LINE

```
{
    "start": "POINT",
    "end": "POINT"
}
```

STATE_UPDATE

```
{
    "newLine": "LINE | NULL",
    "heading": "STRING | NULL",
    "message": "STRING | NULL"
}
```

PAYLOAD

```
{
    "msg": "STRING",
    "body": "STATE_UPDATE | POINT | STRING"
}
```

Overview

Despite its name, the Client API still follows the Client/Server model. The Client exposes two functions to the Server (one to subscribe to requests and another to send responses). The usage of each is briefly demonstrated below:

```
app.ports.request.subscribe((message) => {
    message = JSON.parse(message);
```

```
    // Parse the message to determine a response, then respond:
    app.ports.response.send({ some: 'object' });
});
```

The `app.ports.request.subscribe` function takes as its argument a single-argument, message-handling function. Each time an action occurs in the Client (usually as a response of user activity), the provided message-handling function will be called. Accordingly, `app.ports.request.subscribe` should only be called once to avoid providing duplicate responses to the Client. The Server should call `app.ports.response.send` whenever a response is available. The function accepts a single JS object argument (`PAYLOAD`) and has no return value. Hindley-Milner types for the Client functions are provided below (the symbol `()` is used to represent an undefined value, and it should be understood in all cases that String is `PAYLOAD`):

```
app.ports.request.subscribe :: (String -> ()) -> ()
app.ports.response.send :: Object -> ()
```

Message Format

All RX/TX messages are JSON strings. To simplify message handling, all messages have a common container format (`PAYLOAD`):

```
{
    "msg": "STRING",
    "body": "STATE_UPDATE | POINT | STRING"
}
```

Incoming Messages (Request)

All incoming messages use the `msg` field of `PAYLOAD` to indicate a requested action from the Client. There are three valid values for the `msg` field in Client requests (`INITIALIZE`, `NODE_CLICKED`, `ERROR`). Not all requests expect a corresponding response, but most do. Valid responses for each request are outlined below.

NODE_CLICKED

This message is sent each time a node is clicked. Drawing a line requires two successful clicks. After each click, the Server should send a response indicating whether or not the user clicked a valid node. On the second successful click, the Server should additionally include the `newLine`. The Client does not store the first click. Therefore, if the second click is unsuccessful, it is *not* possible to re-attempt a second click; The click following a failed second click will always be a new first click.

Request The value of `body` will always be a `POINT`.

Example Request Payload

```
{
    "msg": "NODE_CLICKED",
    "body": {
        "x": 0,
        "y": 2
    }
}
```

Response Required. There are five acceptable values for `msg` in the response, representing the four possible states for a clicked node (`VALID_START_NODE`, `INVALID_START_NODE`, `VALID_END_NODE`, `INVALID_END_NODE`) and the game over state (`GAME_OVER`). A `VALID_END_NODE` *must* contain a `LINE` in the `newLine` field. If a `VALID_END_NODE` also constitutes the last move in the game, the Server should send `GAME_OVER` in the `msg` field instead. All other states should contain `NULL` in the `newLine` field. The `heading` and `message` fields can be used to provide feedback to the user; Their value must be `STRING | NULL`, but the content is left unspecified.

Example Response Payloads

```
{
    "msg": "VALID_START_NODE",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Select a second node to complete the line."
    }
}
```

```
{
    "msg": "INVALID_START_NODE",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Not a valid starting position."
    }
}
```

```
{
    "msg": "VALID_END_NODE",
    "body": {
        "newLine": {
            "start": {
                "x": 0,
                "y": 0
```

```
            },
            "end": {
                "x": 0,
                "y": 2
            }
        },
        "heading": "Player 1",
        "message": null
    }
}
```

```
{
    "msg": "INVALID_END_NODE",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Invalid move!"
    }
}
```

```
{
    "msg": "GAME_OVER",
    "body": {
        "newLine": {
            "start": {
                "x": 0,
                "y": 0
            },
            "end": {
                "x": 0,
                "y": 2
            }
        },
        "heading": "Game Over",
        "message": "Player 2 Wins!"
    }
}
```

ERROR

This is a general mechanism for the Client to report errors. In the current implementation, it is only used to indicate that the Server sent a response with an invalid message format. This message does require (or expect) a response.

Example Message The `body` field will always be a `STRING`.

```
{
    "msg": "ERROR",
    "body": "Invalid type for `id`: Expected INT but got a STRING"
}
```

Outgoing Messages (PubSub)

In addition to request/response messages, the Server can publish a message without an initial request from the Client. These PubSub messages are used to begin the game and to trigger unprompted updates to the `heading` or `message` properties.

INITIALIZE

To initialize the Client and begin the game, send the `INITIALIZE` message. This is always the first message and can also be used to reset the game. This message indicates that the application is set (or reset) to its initial state. The `newLine` property must be `null`, but `heading` and `message` may be set to an arbitrary initial value.

Example Payload

```
{
    "msg": "INITIALIZE",
    "body": {
        "newLine": null,
        "heading": "Player 1",
        "message": "Awaiting Player 1's Move"
    }
}
```

UPDATE_TEXT

This message is completely optional. It could be used, for instance, to clear the `heading` or `message` after a certain period of time or to nag the user after a period of inactivity. The Client is dumb and will *not* ignore `newLine` if it is included, but doing so would clearly corrupt the Client's representation of the game state.

Example Payload

```
{
    "msg": "UPDATE_TEXT",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Are you asleep?"
```

```
    }
}
```

HTTP API Protocol

Types

The API uses JSON for request/response bodies. Because JSON does not provide a means of specifying compound types, the following ad hoc type definitions are provided.

The pipe character <|> represents logical `OR`.

JSON Primitives

```
string : STRING
number : INTEGER | FLOAT
boolean : BOOLEAN
null : NULL
```

POINT

```
{
    "x": "INTEGER",
    "y": "INTEGER"
}
```

LINE

```
{
    "start": "POINT",
    "end": "POINT"
}
```

STATE_UPDATE

```
{
    "newLine": "LINE | NULL",
    "heading": "STRING | NULL",
    "message": "STRING | NULL"
}
```

PAYLOAD

```
{
    "msg": "STRING",
    "body": "STATE_UPDATE | POINT | STRING"
}
```

Overview

The game begins when the Client makes a request to `/initialize`. To simplify the implementation, the server should not maintain session state for multiple players. To reiterate, it is expected that the server will maintain state for only a single active game. If an `/initialize` request is received before the current game is over, assume that the player has reloaded the Client and that a new game should begin.

Message Format

All request/response bodies are JSON strings. To simplify response handling on the Client, all responses have a common container format (`PAYLOAD`):

```
{
    "msg": "STRING",
    "body": "STATE_UPDATE | POINT | STRING"
}
```

API Endpoints

The Server must provide the following endpoints. The Server can ignore query parameters and HTTP headers - they have no meaning in the provided protocol.

GET /initialize

This is always the first request, sent immediately after the Client loads. It can also be used to reset the game. This message indicates that the application should be set (or reset) to its initial state. The response is required, and can be used to pre-populate `heading` or `message`.

Request The request has no body.

Response

Example Response Payload

```
{
    "msg": "INITIALIZE",
    "body": {
        "newLine": null,
        "heading": "Player 1",
        "message": "Awaiting Player 1's Move"
    }
}
```

POST /node-clicked

This request is made each time a node is clicked. Drawing a line requires two successful clicks. After each click, the Server's response should indicate whether or not the user clicked a valid node. On the second successful click, the Server should additionally include the `newLine`. The Client does not store the first click. Therefore, if the second click is unsuccessful, it is *not* possible to re-attempt a second click; The click following a failed second click will always be a new first click.

Request The request body will always be a `POINT`.

Example Request Payload

```
{
    "x": 0,
    "y": 2
}
```

Response There are five acceptable values for `msg` in the response, representing the four possible states for a clicked node (`VALID_START_NODE`, `INVALID_START_NODE`, `VALID_END_NODE`, `INVALID_END_NODE`) and the game over state (`GAME_OVER`). A `VALID_END_NODE` *must* contain a `LINE` in the `newLine` field. If a `VALID_END_NODE` also constitutes the last move in the game, the Server should send `GAME_OVER` in the `msg` field instead. All other states should contain `NULL` in the `newLine` field. The `heading` and `message` fields can be used to provide feedback to the user; Their value must be `STRING | NULL`, but the content is left unspecified.

Example Response Payloads

```
{
    "msg": "VALID_START_NODE",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Select a second node to complete the line."
    }
}
```

```
{
    "msg": "INVALID_START_NODE",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Not a valid starting position."
```

```
    }
}
```

```
{
    "msg": "VALID_END_NODE",
    "body": {
        "newLine": {
            "start": {
                "x": 0,
                "y": 0
            },
            "end": {
                "x": 0,
                "y": 2
            }
        },
        "heading": "Player 1",
        "message": null
    }
}
```

```
{
    "msg": "INVALID_END_NODE",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Invalid move!"
    }
}
```

```
{
    "msg": "GAME_OVER",
    "body": {
        "newLine": {
            "start": {
                "x": 0,
                "y": 0
            },
            "end": {
                "x": 0,
                "y": 2
            }
        },
        "heading": "Game Over",
        "message": "Player 2 Wins!"
    }
}
```

POST /error

This is a general mechanism for the Client to report errors. The request is solely intended to aid debugging, and the Client ignores any response.

Example Message

```
{
    "error": "Invalid type for `id`: Expected INT but got a STRING"
}
```

WebSocket API Protocol

Types

The API uses JSON as a messaging format. Because JSON does not provide a means of specifying compound types, the following ad hoc type definitions are provided.

The pipe character <|> represents logical `OR`.

JSON Primitives

```
string : STRING
number : INTEGER | FLOAT
boolean : BOOLEAN
null : NULL
```

POINT

```
{
    "x": "INTEGER",
    "y": "INTEGER"
}
```

LINE

```
{
    "start": "POINT",
    "end": "POINT"
}
```

STATE_UPDATE

```
{
    "newLine": "LINE | NULL",
    "heading": "STRING | NULL",
```

```
    "message": "STRING | NULL"
}
```

PAYLOAD

```
{
    "id": "INTEGER",
    "msg": "STRING",
    "body": "STATE_UPDATE | POINT | STRING"
}
```

Overview

The WebSocket API is an extension of the Client API. Unlike most WebSocket APIs, which are typically PubSub, this API is primarily Request/Response (exceptions are clearly marked). As such, each RX/TX message contains an `id` as a means of preserving message order. If a request contains an `id` of `2`, then the corresponding response should also contain an `id` of `2`. If the Client receives a response with an `id` that does not match the most recent request, or ten seconds pass without receiving a response, then the Client will re-send the most recent request. Accordingly, if two successive requests are received with the same message `id`, the Client is indicating that it did not receive the most recent response, and the response should be re-sent. There is one exception: PubSub messages are indicated by a message `id` of `0`. In this case, both Client and Server should infer that the message is unrelated to any previous message and does not require a response.

The game begins when the Client sends the `INITIALIZE` message. To simplify the implementation, the server should not maintain session state for multiple players. To reiterate, it is expected that the server will maintain state for only a single active game. If an `INITIALIZE` message is received before the current game is over, assume that the player has reloaded the Client and that a new game should begin.

Message Format

All RX/TX messages are JSON strings. To simplify message parsing on both the Client and the Server, all messages have a common container format (`PAYLOAD`):

```
{
    "id": "INTEGER",
    "msg": "STRING",
    "body": "STATE_UPDATE | POINT | STRING"
}
```

Incoming Messages (Request)

All incoming messages use the `msg` field of `PAYLOAD` to indicate a requested action from the Client. There are three valid values for the `msg` field in Client requests (`INITIALIZE`, `NODE_CLICKED`, `ERROR`). Not all requests expect a corresponding response, but most do. Valid responses for each request are outlined below.

INITIALIZE

This is always the first request, sent immediately after the Client loads. It can also be used to reset the game. This message indicates that the application should be set (or reset) to its initial state. The response is required, and can be used to pre-populate `heading` or `message`.

Request The value of `body` will always be `null`.

Example Request Payload

```
{
    "id": 1,
    "msg": "INITIALIZE",
    "body": null
}
```

Response Required.

Example Response Payload

```
{
    "id": 1,
    "msg": "INITIALIZE",
    "body": {
        "newLine": null,
        "heading": "Player 1",
        "message": "Awaiting Player 1's Move"
    }
}
```

NODE_CLICKED

This message is sent each time a node is clicked. Drawing a line requires two successful clicks. After each click, the Server should send a response indicating whether or not the user clicked a valid node. On the second successful click, the Server should additionally include the `newLine`. The Client does not store the first click. Therefore, if the second click is unsuccessful, it is *not* possible to

re-attempt a second click; The click following a failed second click will always be a new first click.

Request The value of `body` will always be a `POINT`.

Example Request Payload

```
{
    "id": 3,
    "msg": "NODE_CLICKED",
    "body": {
        "x": 0,
        "y": 2
    }
}
```

Response Required. There are five acceptable values for `msg` in the response, representing the four possible states for a clicked node (`VALID_START_NODE`, `INVALID_START_NODE`, `VALID_END_NODE`, `INVALID_END_NODE`) and the game over state (`GAME_OVER`). A `VALID_END_NODE` *must* contain a `LINE` in the `newLine` field. If a `VALID_END_NODE` also constitutes the last move in the game, the Server should send `GAME_OVER` in the `msg` field instead. All other states should contain `NULL` in the `newLine` field. The `heading` and `message` fields can be used to provide feedback to the user; Their value must be `STRING | NULL`, but the content is left unspecified.

Example Response Payloads

```
{
    "id": 3,
    "msg": "VALID_START_NODE",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Select a second node to complete the line."
    }
}
```

```
{
    "id": 3,
    "msg": "INVALID_START_NODE",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Not a valid starting position."
    }
}
```

```
{
    "id": 3,
    "msg": "VALID_END_NODE",
    "body": {
        "newLine": {
            "start": {
                "x": 0,
                "y": 0
            },
            "end": {
                "x": 0,
                "y": 2
            }
        },
        "heading": "Player 1",
        "message": null
    }
}
```

```
{
    "id": 3,
    "msg": "INVALID_END_NODE",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Invalid move!"
    }
}
```

```
{
    "id": 3,
    "msg": "GAME_OVER",
    "body": {
        "newLine": {
            "start": {
                "x": 0,
                "y": 0
            },
            "end": {
                "x": 0,
                "y": 2
            }
        },
        "heading": "Game Over",
        "message": "Player 2 Wins!"
    }
}
```

ERROR

This is a general mechanism for the Client to report errors. In the current implementation, it is only used to indicate that the Server sent a response with an invalid message format. This is a PubSub message and does not expect a response.

Example Message The `body` field will always be a `STRING`.

```
{
    "id": 0,
    "msg": "ERROR",
    "body": "Invalid type for `id`: Expected INT but got a STRING"
}
```

Outgoing Messages (PubSub)

All response messages are covered above, along with the corresponding request. Additionally, the Server can publish a message to update `heading` or `message`.

UPDATE_TEXT

This message is completely optional. It could be used, for instance, to clear the `heading` or `message` after a certain period of time or to nag the user after a period of inactivity. The Client is dumb and will *not* ignore `newLine` if it is included, but doing so would clearly corrupt the Client's representation of the game state. The value of `id` must be `0`.

Example Payload

```
{
    "id": 0,
    "msg": "UPDATE_TEXT",
    "body": {
        "newLine": null,
        "heading": "Player 2",
        "message": "Are you asleep?"
    }
}
```