Software Engineering 2

Project Game System

version 2.3.0

Michał Okulewicz
Modified by Bartłomiej Szymański

March 31, 2019

Contents

Glossary

The Project Game - a real-time board game played by two competing teams of cooperating Players

Player - an agent playing the game, holds its own view of the state of the game

Team - group of *Players* who cooperate in order to achieve the goal of the game

Piece - a token representing a project resource which is initially located in the *Tasks Area* of the board by the *Game Master*,

Goal Field - a type of field modelling a single project objective

Goal Area - a part of the board where the *Players* of a *Team* have exclusive access and *Goal Fields* are located (not all fields in the *Goal Area* are *Goal Fields*)

Tasks Area - a part of the board from which the *Players* may collect the *pieces*

Game Goal - discovering the all the *Goal Fields* by placing pieces in the fields of the *Goal Area*

Project Game System - a distributed IT system for managing multiple *Project Games* and the agents participating in them

Game Master - an agent responsible for generating the board and location of the *Goal Fields*, holds the whole state of the game and generates new pieces in the *Tasks Area*,

Communications Server - a module responsible for passing messages between *Game Master* and *Players*

Changelog

1.1.0 2018-04-09 Destroy piece action added

1.2.0 2018-04-09 Knowledge exchange communication protocol changed

1.2.1 2018-04-09 A more explicit statement about message queue on Game Master

1.2.2 2018-04-15 System extended with an error message

1.2.3 2018-04-15 Fixed leader—member issues

1.3.0 2018-04-16 Added ability to suggest actions

2.0.0 2019-03-08 Issue #10
Updated the documentation to work with the 2019 Software Engineering 2 project

2.1.0 2019-03-19 Issues #12-13, #14-25
Addressed a ton of consistency issues created by moving to JSONs, listed all possibilities for each of the messages, added three more messages and introduced the concept of GUID, changed diagrams and revamped communication aspects

2.2.0 2019-03-26 Issues #26-35, #37
Added GUID to every message, clarified JSONs further, changed values of manhattanDistance fields. Introdution of timestamping, reworking KnowledgeExchange. Gave information on wha information does Player need to keep track of. Solved a ton of ambiguities, make rules clearer, tweaked log, set Board Team locations parameters in stone

Introduction

The purpose of this document is to present the rules and the goal of The Project Game and specify the requirements for multi-agent system for playing the game.
The purpose of The Project Game itself is to create an environment in which computer and human agents might interact and whose behaviour can be observed, quantified and simulated. The scope of this particular system is focused on the message passing system and computer agents only.
The system can be seen from the following perspectives:

- as a network system,
- as a distributed database system,
- as a multi-agent system.

Network system perspective

The Players are separated from each other and from Game Master by a network connection. In the network system perspective, the goal of the project is to implement transparent communication between Players and Game Master agents. The technical part of the communication should be properly separated from the logic through the message objects management and communication management layers.

Distributed database perspective

Game Master holds the whole and most accurate state of the game (master database). Players keep partial local copies of the state of the game (partial mirror databases). They need to keep track of time they have received information about the state of the board with a certain set time accuracy (for example every 10 miliseconds increase the timer by one to keep the messages synchronized). Access to master database is limited to 9 fields per request, while access to the data in mirror databases is limited by the accessibility of those nodes and the state of their data. In the distributed system perspective, the goal of the project is to implement a best strategy for obtaining the most accurate and crucial data with as little cost as possible.

Multi-agent perspective

Each Player forms beliefs from the knowledge obtained from the Game Master and other Players. Each Player assumes certain intentions of the Players in his Team and the opposite Team. In the multi-agent perspective, the goal of the project is to implement the best strategy for obtaining, testing, placing the pieces and exchanging information resulting in the fastest possible winning the game.

Document contents

Part I of the document describes the game organization, the objects appearing in the game, the possible moves of the Players and their effects. Part II specifies the system technical requirements and defines the communication protocol.

Special thanks

Wojtek Zieliński, Siyana Ivanova and Jarosław Korbut for comprehensive help in getting this document in a format that would allow to produce an actual game from it. Seriously, I'm very grateful for that.

Part I

The Project Game

Chapter 1

Game description

1.1 Game concept

The idea of The Project Game is to simulate competitive project development by two teams of Players. The game simulates the following properties of projects development: (1) tasks are connected with risks (usually negative), (2) goals of the project are unclear (and need to be discovered), and (3) communication between members helps to speed up the process of the development.

1.2 Game rules

The rules of the game could be summarized in the following way:

1. The game is played by two teams of Players: red and blue.
2. The game is played on a rectangular board[1] (depicted in Figure 1.1).
3. The board is divided into 3 parts: a common tasks area, a red team goals area and a blue team goals area.
4. The game is controlled by a game master, who places (in secret and randomly) pieces on the tasks area.
5. The Player has ability to move around the board, discover the state of surrounding fields, handle the pieces and exchange information with other Players.
6. The game is played in real time with a time cost assigned to each of the possible Player's actions.
7. The objective of the team is to complete the project as fast as possible by discovering all the goal fields in the goals area.

[1] the size of the board may vary

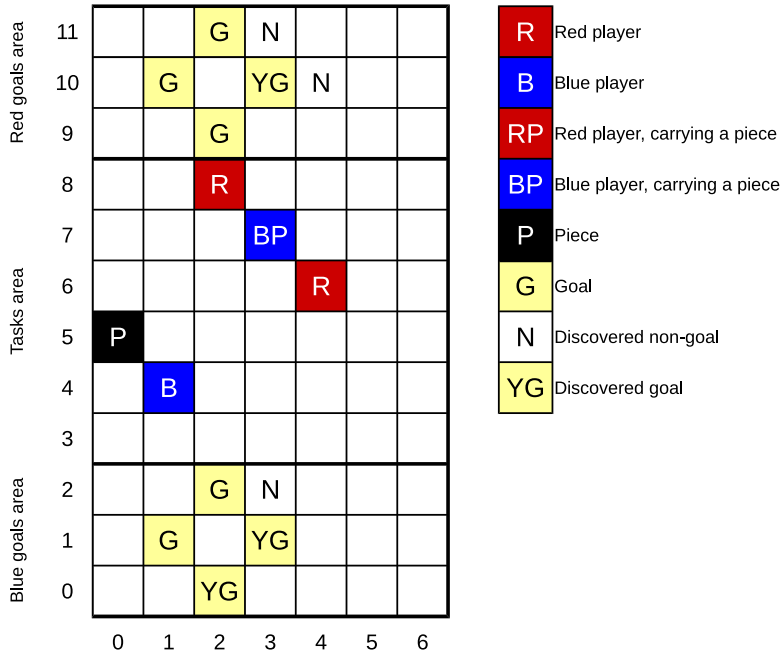

Figure 1.1: Global state of the game.

8. The goal fields are known only to the game master, a Player who discovered them and the Players with whom that information has been shared.

9. Player may carry only a single piece and a field in the tasks area may contain only a single piece[2].

1.3 Game state

The state of The Project Game consists of the following data:

- Players' locations
- Pieces' locations
- Pieces' state (sham or non-sham)
- Project's goals locations
- Project's goals discovery state

The true state of the game is known only to the Game Master. Figure 1.1 presents the state of the game as seen by the Game Master.

[2]Player cannot place a piece on a field containing a piece, but Game Master can do it and destroy the piece which previously has been in the field

1.4 Game actions

The Players discover the state of the game by interacting with the board, the pieces and each other. State of the game perceived by one of the blue Players before and after exchanging information with another blue Player is depicted in Figure 1.2.
Possible moves of the Player consist of:

- moving in one of 4 directions (up, down, right and left),
- discovering the contents of 8 neighbouring fields (and the currently occupied field),
- picking up a piece from a board,
- testing the picked piece for being a sham,
- destroying a piece held be a Player (usually because it is a sham),
- placing a piece in the field in the goal area, in hope of completing one of the project objectives (or leaving a sham piece on the board),
- request exchange of information with another Player,

The Player may discover the goals only by placing (using) a piece in a given field of the goal area:

- A correctly placed piece results in an information to the Player that one of the project's goals has been completed (field has been a goal field).
- Incorrectly placed piece or a sham piece on a goal area results in an information that the completed action (getting the piece from the tasks board and placing it in the goals area) has been meaningless, in the sense of project completion (field has not been a goal field).

Player actions have following ramifications and constrains:

- Player cannot move into a field occupied by another Player.
- Player cannot move into the goals area of the opposing team.
- The piece may be picked only by a Player which is located in the same field as that piece.
- Player can handle only one piece (he cannot pick up another piece).
- Player can place piece only on an empty field (with no other piece on it).
- Observing a field (either by discovering or entering it) results in receiving information about the Manhattan distance to the nearest piece.
- Team leader is one of the Players in the game

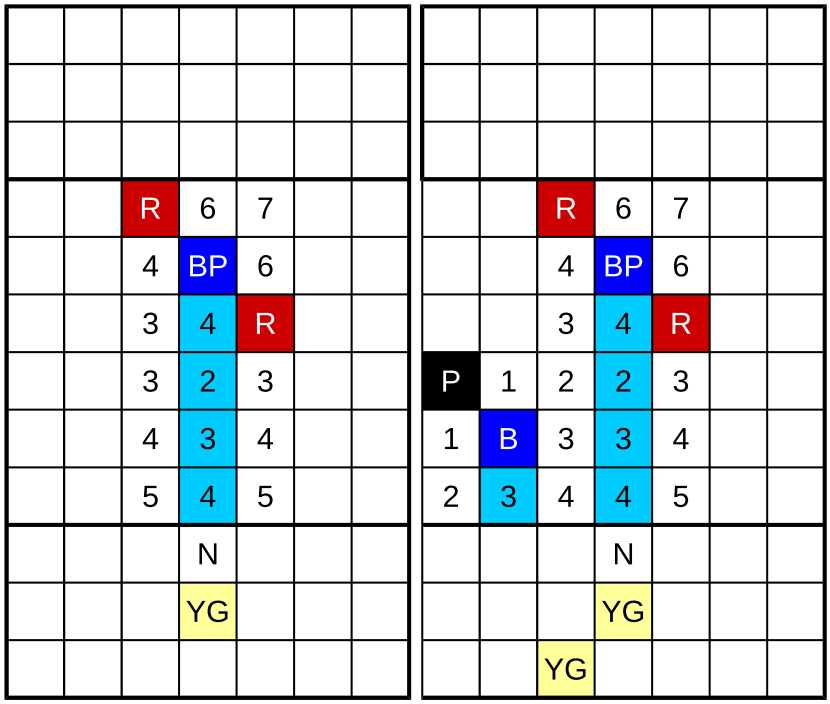

Figure 1.2: Personal game state perceived by one of the blue Players. The left one is before he exchanged knowledge with the other blue Player, while the right one after the exchange. Light blue color marks the traces of each of the clients. Visible piece has been added by Game Master during Player movement. Keep in mind that the Discover command works like a sonar and the information Discovered by Player becomes obsolete the moment a new piece joins the game or a piece becomes picked up by another Player.

Part II

The Project Game System

Chapter 2

Requirements

The system organizes matches between teams of cooperating agents. The game is played in real time on a rectangular board, consisting of the tasks area and the goals area. Red team's goal area is located on top of the board (with lower Y values), while the Blue team's goal area is located at the bottom (with higher Y values). Each team of agents needs to complete an identical project, consisting of placing a set of pieces, picked from the tasks area, in the project goals area. The pieces appear at random within the tasks area. Pieces cannot spawn on top of fields that already contain pieces. They can spawn on top of fields occupied by Players however. The agent's view of the tasks area is limited and the shape of the goals needs to be discovered. The game is won by the team of agents which is first to complete the project.

2.1 Features

All communication passes through Communication Server and uses TCP sockets. Game Master serves one game at a time. After completing one game Game Master and Players after 10 seconds sign up for another game (with the same settings). Game Master logs requests metadata and a victory or defeat event for each of the Players. Displays their ranking and average response times after each game on a text console.

Each Player displays the whole state of the board on a text console after completing the game.

All the components should be able to run in a text-mode only. Graphical User Interface is unnecessary, but may be additionally developed as a means of presenting the game state for development or testing purpose.

2.2 Supportability

Game Master should log the following data after receiving any type of request from a Player:

- Request type (name of a root element in the JSON message)
- Player GUID

- Player colour
- Player role

Game Master should also place an event winning/losing game event in the same log (marked as Victory or Defeat for each of the Players). The log should be stored in an UTF-8 encoded CSV file located in the same directory as the binary files of the Game Master.

A sample part of the event log should look as follows:

Type	Player GUID	Colour	Role
TestPiece	c094...	blue	member
Move	c094...	blue	member
Move	c094...	blue	member
Move	c179...	red	leader
PlacePiece	c094...	blue	member
Victory	c094...	blue	member
Defeat	c179...	red	leader

2.3 Performance

Communication Server and Game Master are able to serve smoothly at least 16 Players. Theoretical number of Players is unlimited. Communication Server is allowed to host only one independent game.
The size and number of messages should be minimal.

2.4 Security

Player should be unable to get information available only to other Players. Player should not try to influence other Players' decisions with false information.

2.5 Technical settings

All the technical settings are set through a JSON configuration file.
The following parameters are set in all the components:

- Interval between keep alive bytes (or expected keep alive)

The following parameters are set in the Player agent:

- Interval between retries for joining the game.

2.6 Game settings

All the game settings are set through an JSON configuration file.
In order to create various types of games the following elements of the game are configurable as the parameters of the Game Master agent (with proposed values inside brackets):

- Probability of piece being a sham (0.5)
- Frequency of placing new pieces on board (2 every 5 seconds)
- Initial number of pieces on board (3)
- Width of the board, length of the tasks area, length of a single goals area (10, 10, 3)
- Number of Players in each of the teams (4)
- Goal definition (the same amount of goals for both teams) (an array of 5 goals per team)

In order to balance different actions of the Players, response delay for each of the actions is configured as the parameters of the Game Master agent (with default values given in parenthesis):

- Move delay (100ms)
- Discovery delay (450ms)
- Test delay (500ms)[1]
- Destroy delay (100ms)
- Pick-up delay (100ms)
- Placing delay (100ms)
- Knowledge exchange (1200ms)[2]

Delays should be introduced in such a way, that the Player's actions affect the state of the game only after all preceding delay times have already passed. The Players action might be activated at any moment during its delay, but a given Player must not receive results before the delay time has passed.

2.7 Running the application

Apart from the configuration files the following parameters are passed while starting the application and a proper shell script / batch file accepting them, and running certain modules of the system, must be dispatched with the project.

- YY - the current year (17)
- XX - group identifier

Communication Server runtime parameters:
```
YY-XX-cs --port [port number binded on all interfaces] --conf [configuration filename]
```

Player and Game Master runtime parameters:
```
YY-XX-[pl|gm] --address [server IPv4 address or IPv6 address or host name] --port
```

[1] The idea: should take as long as crossing half of the board.

[2] The idea: should take as long as discovering number of the board task fields divided by number of Players in team.

```
[server port number] --conf [configuration filename]
```

In order to join a particular type of a game in a particular role the following elements are configurable as the parameters of the Player agent:

- Preferred colour
- Preferred role

The Player will play only the game named exactly as requested, but the colour and the role might be assigned by Game Master as needed.

For example, if the game has already all the necessary blue Players, and there are new Players coming, who prefer to be blue, they would still be given a vacant role in a red team.

The choice of the the role is done by the Game Master.
Therefore, apart from the technical setting the Player has additional parameters:
`--team [red|blue]`

Chapter 3

Communication protocol

The communication is maintained through the TCP/IP protocol. All messages are passed through the Communication Server which acts as the service for adding people and removing people from the games and as a proxy for passing all other messages.

The specification presents the exchange of messages between the components for the following typical scenarios: setting up and starting a game, normal gameplay, information exchange between the Players.

All messages are text JSON messages. The messages are separated by a bytecode 23 (ETB - End transmission blocks). The code is present after each message and is also used as a connection keep alive data. The TCP/IP connections are maintained during the whole time of system operations, until the process, of at least one of the components in that particular connection, is finished. Keep alive byte is sent from Player and Game Master to the Communication Server and responded with a keep alive byte. Sending a normal message also acts as keep alive.

In order for the Communication Server to run the game, all messages sent from Players intended for a Game Master are marked with the `userGuid`.

3.1 Starting a game

In order to start a new game Game Master must set up the game on the Communication Server by `SetUpGame` message.

After completing all the Players, Game Master broadcasts the `BeginGame` message to all Players, containing information about the size of the board, their teams and their initial location. The exchange of those messages and their responses is presented in Fig. 3.1.

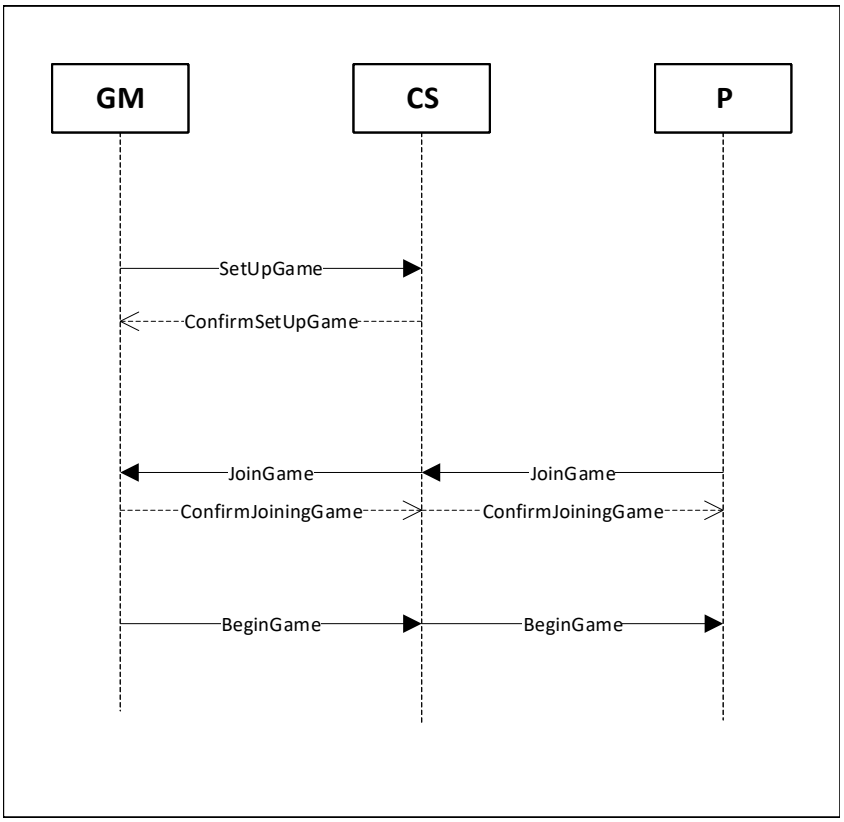

Figure 3.1: The messages passed between a sample (P)layer, (G)ame (M)aster and (C)ommunication (S)erver when a new game is organized in the system.

```
{
    "action": "start"
}
```

Figure 3.2: An example of `SetUpGame` message.

```
{
    "action": "start",
    "result": "<OK|denied>"
}
```

Figure 3.3: An example of `ConfirmSetUpGame` message starting the game.

```
{
    "action": "connect",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "preferredTeam": "<red|blue>",
    "type": "player"
}
```

Figure 3.4: A `JoinGame` message with Player trying to join, as a Player of one of the teams.

```
{
    "action": "connect",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "OK",
    "type": "player"
}
```

Figure 3.5: A `ConfirmJoiningGame` message.

```
{
    "action": "connect",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "denied",
    "type": "player"
}
```

Figure 3.6: A `RejectJoiningGame` message informing that Game Master rejects Player.

```
{
    "action": "start",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "team": "<blue|red>",
    "role": "<member|leader>",
    "teamSize": <1|2|3|4|...|maximal number specified in the config JSON>,
    "teamGuids": [
        "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
        "0a041a66-01b7-40cf-a06a-a47550f0e6c1",
        "0a041a66-01b7-40cf-a06a-a47550f0e6c2",
        ...
    ]
    "location": {
        "x": <0|1|2|3|4|...|board width specified in the config JSON -1>,
        "y": <0|1|2|3|4|...|board height specified in the config JSON -1>
    },
    "board": {
        "width": <1|2|3|...|board width specified in the config JSON>,
        "tasksHeight": <1|2|3|...|tasks height specified in the config JSON>,
        "goalsHeight": <1|2|3|...|goals height specified in the config JSON>
    }
}
```

Figure 3.7: A `BeginGame` message sent to a Player of a given team. The "teamGuids" array contains GUIDs of each of the team members, with the first one being assigned to a leader and every following one assigned to members of the team in order.

3.2 Standard gameplay

During a standard gameplay Players send `Discover`, `Move`, `PickUp`, `TestPiece`, `DestroyPiece` and `PlacePiece` messages to the Game Master. The messages specify only the information necessary for the proxy server to be dispatched to a Game Master and the data allowing to safely identify the Player (`userGuid` field) and decide about the Player's action (direction in the case of movement). Most the other details (like the ID of the piece being picked up) are maintained by the Game Master only. The things that Player needs to keep track involve:

- His GUID
- His current position
- The GUIDs of his team members (and if any of them had become permanently rejected from communicating with)
- The size of teams
- The team that he belongs to
- His role within a team
- A basic storage container for the state of the board (width, tasksHeight, goalsHeight), generated after receiving the `BeginGame` message.
- A basic, list-alike storage for each of the fields of the board that the Player has any information about containing timestamps for each of the fields (applied when receiving messages from Game Master), their x and y coordinates, their "manhattanDistance", "contains" and "userGuid" properties
  - "manhattanDistance" can be either a null or a numeric value not bigger than the Board Width + Board Length
  - "contains" can be one of the six values:
    - "empty", if a field below is empty
    - "goal", if a field below is a goal
    - "piece", if a field below contains an unchecked piece
    - "sham", if a field below contains a piece that was checked to be a sham and dropped by the Player that did it or Knowledge Exchanged with the Player that has this data
    - "valid", if a field below contains a piece that was checked to be a valid and dropped by the Player that did it or Knowledge Exchanged with the Player that has this data
    - "unknown", if a field is located within the Goals area and has not been Discovered by the Player or the Player that sent the Knowledge Exchange message

    Please note that it is not possible for "contains" to get changed to "sham" or "valid" through any means other than Knowledge Exchange or checking the Piece while holding it.

- "userGuid" contains GUID of a Player standing on top of it it and only if there was a Player standing on top of it in a given "timestamp"

To each request Game Master responses with a `Data` message containing the information obtained by the Player, concerning part of the game state.

After the move finishing the game Game Master broadcasts `GameOver` message containing the winning team and the piece of information that the game has finished to all the Players. The `Data` messages include the information allowing the server to dispatch the data to a proper Player (`userGuid` field) and the data about the game state aggregated in the `TaskFields`, `GoalFields`, `Pieces` and `PlayerLocation` elements.

Please note, that the Manhattan distance to the nearest piece is calculated to a nearest piece currently placed on a field in the Tasks Area, regardless of its status (sham or not-sham). If no such piece is currently in the game, the distance value is equal to null.

If a Player is standing on a piece, the value of Manhattan distance is set to 0. Player might rightfully assume that the spot is populated by a piece without having to use Discovery command for that. The distance is calculated only for the fields in the Tasks Area, but can be observed from a Goals Area if the discover action is applied while standing on a field in the Goals Area neighbouring fields in the Tasks Area. For discovering fields located within the Goals area, the Manhattan distance returned is always equal to null.

A `timestamp` field prevalent across multiple messages gives information on how many units of time have passed since the Game has started. Game Master is the only one entity that keeps updating how much time has passed since the game has started, Players need to store the data that they obtain from Game Master with `Move`, `Discover`, `Place` and `KnowledgeExchange` messages.

Figure 3.8 presents the messages exchange between Players and Game Master. Please note, that the Communication Server **always acts as a proxy** in this communication, but is not depicted. Knowledge exchange between the Players, which is also a part of a gameplay, is explained in the next section.

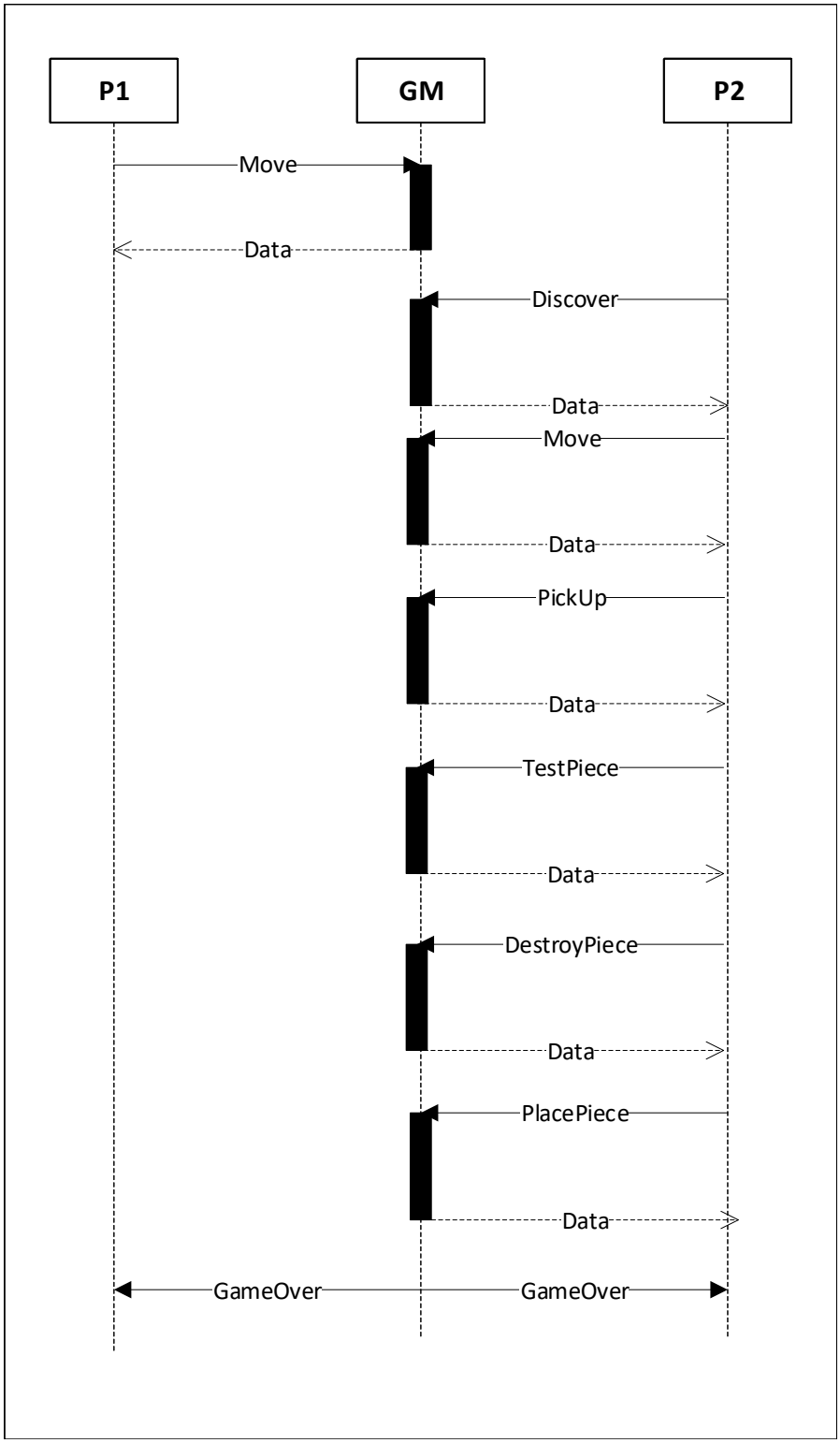

Figure 3.8: The messages passed between sample Players P1,P2 and a (G)ame (M)aster during normal gameplay. The Communication Server is not depicted but acts as a proxy for communication.

```
{

    "action": "state",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "location": {
        "x": <0|1|2|3|4|...|board width specified in the config JSON -1>,
        "y": <0|1|2|3|4|...|board height specified in the config JSON -1>
    }
}
```

Figure 3.9: A `Discover` message from Player.

```
{
    "action": "state",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "<OK|denied>",
    "location": {
        "x": <0|1|2|3|4|...|board width specified in the config JSON -1>,
        "y": <0|1|2|3|4|...|board height specified in the config JSON -1>
    },
    "fields": [
        {
            "x": <0|1|2|3|4|...|board width specified in the config JSON -1>,
            "y": <0|1|2|3|4|...|board height specified in the config JSON -1>
            "value": {
                "manhattanDistance": <null|0|1|2|...|Nearest piece M. dist.>,
                "contains": "<goal|piece|empty>",
                "userGuid": "fa3f3cb5-742d-478f-9e02-230679032777"
            }
        },
        {
            "x": <0|1|2|3|4|...|board width specified in the config JSON -1>,
            "y": <0|1|2|3|4|...|board height specified in the config JSON -1>
            "value": {
                "manhattanDistance": <null|0|1|2|...|Nearest piece M. dist.>,
                "contains": "<goal|piece|empty>",
                "userGuid": null
            }
        }
    ]
}
```

Figure 3.10: A `Data` message response for the discover action. The "fields" field is empty if the message request becomes denied. If a Player stands on a field, "userGuid" field is not null. Otherwise, "userGuid" is null. Keep in mind that "contains" does not return information on if a piece is a valid piece or a sham piece, even if one of the Players has checked it. Only Game Master and the Player who checked it has this knowledge and the information can be returned to other Players by knowledge exchange from the Player who has checked the piece. The maximal amount of fields in the "fields" array is 9. As soon as a Player grabs a piece, the piece stops being recognised on the board, "manhattanDistance" does not lead to it anymore and the spot that used to contain a piece returns "empty" until a piece is dropped.

```
{
    "action": "move",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "direction": "<N|S|W|E>"
}
```

Figure 3.11: A `Move` message from Player.

```
{
    "action": "move",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "OK",
    "manhattanDistance": <null|0|1|2|...|Nearest piece M. dist.>
}
```

Figure 3.12: A `Data` message response for the proper move action.

```
{
    "action": "move",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "denied",
    "manhattanDistance": null
}
```

Figure 3.13: A `Data` message response for the move action, when trying to enter an occupied field or field away from the board.

```
{
    "action": "pickup",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0"
}
```

Figure 3.14: A `PickUp` message from a Player.

```
{
    "action": "pickup",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "<OK|denied>",
}
```

Figure 3.15: A `Data` message response for the piece pick up action. ”result” set to ”denied” if attempting to pick up from a place that does not have a piece on it.

```
{
    "action": "test",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0"
}
```

Figure 3.16: A `TestPiece` message from a Player.

```
{
    "action": "test",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "<OK|denied>",
    "test": <true|false|null>
}
```

Figure 3.17: A `Data` message response for the testing of a piece action. "test" is null if "result" is "denied" (for example, agent does not hold a piece. If Player believes he is holding a piece at that time, he has to revoke his thinking and acknowledge that he does not, Game Master is the ultimate entity when comparing different outcomes). "true" indicates that the piece is a sham, "false" indicates that it is, indeed, a valid piece.

```
{
    "action": "place",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0"
}
```

Figure 3.18: A `PlacePiece` message from a Player.

```
{
    "action": "place",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "<OK|denied>",
    "consequence": "<correct|meaningless"|null>,
}
```

Figure 3.19: A `Data` message response for the placing of a piece in a goals area action. "result" is "denied" if attempting to place a piece on a spot that is not legitimate for placing the piece, for example on top of a piece. "consequence" set to "correct" if a valid piece is being placed on a valid goal, "meaningless" if a valid/sham piece is placed on any other spot or a sham piece is placed on one of the goals, "null" if the "result" is "denied". As a result of the "correct" consequence, the "goal" field becomes "empty" and both Game Master needs to acknowledge it and Player needs to update his mini database on a spot underneath him with the new timestamp and the new "contains" value.

```
{
    "action": "destroy",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0"
}
```

Figure 3.20: A `DestroyPiece` message to indicate the intent to destroy the held piece by the Player.

```
{
    "action": "destroy",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "<OK|denied>"
}
```

Figure 3.21: A `Data` message response to indicate that the request has been acknowledged. If the piece had been destroyed, "result" is set to "OK", if a Player for example was not holding a piece to begin with, "result" is set to "denied".

```
{
    "action": "end",
    "result": "<red|blue>"
}
```

Figure 3.22: A `GameOver` message to indicate that the game was over and one of the teams has won. It does not require a response from Players, hence it doesn't need GUIDs in its body.

3.2.1 Knowledge exchange

Knowledge exchange is a special type of action during the gameplay as it involves not only a single Player and a Game Master, but also another Player. Therefore, it needs special handling as the cost of all actions need to be controlled and imposed by the Game Master.
The knowledge exchange is depicted in Fig. 3.23, with omission of the Communication Server as a proxy, and performed in a following way:

1. `AuthorizeKnowledgeExchange` message is sent from Player 1 to the Game Master and relayed to the intended Player 2.
2. Player 1 sends also `Data` message **immediately** after `AuthorizeKnowledgeExchange` message
3. Player 2 might reject the offer (without further delay) by sending a `RejectKnowledgeExchange` to Player 1 (through Game Master, without further delay)
4. Player 2 might accept the offer by sending a `AcceptKnowledgeExchange` message to Player 1 (through Game Master)
5. Game Master relays the `AcceptKnowledgeExchange` message from Player 2 to Player 1, and after a delay `Data` message from Player 1 to Player 2.

3.2.2 Player handling of updating the messages

If a message on Knowledge Exchange becomes accepted by one of the Players and the Data message becomes transmitted, that player updates their own fields storage accordingly with the message obtained by the second Player.

If he already has the data on a given field that is newer than the data obtained from the Knowledge Exchange process (checked by comparing "timestamp" of the same field), he does not update the "manhattanDistance", "contains" and "userGuid" fields in his database, even if "contains" used to be "piece" inside the database and the older Knowledge Exchanged message contains "valid" or "sham" in it.

The reason being that other players can move valid piece somewhere else and leave a sham piece on its place. This is why Knowledge Exchange works better when is done directly after placing pieces, either on empty spot on on goal spots.

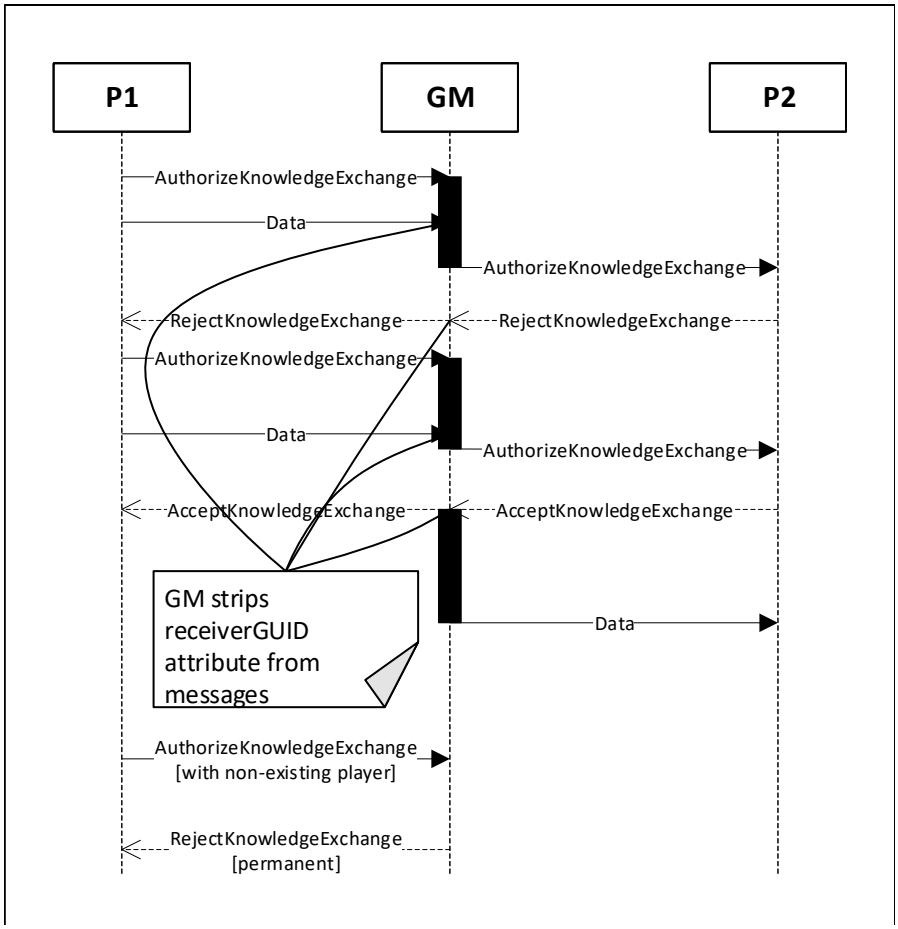

Figure 3.23: The messages passed between sample Players P1,P2 and a (G)ame (M)aster with possible scenarios for data exchange. The Communication Server is not depicted but acts as a proxy for communication.

```
{
    "action": "exchange",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "receiverGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c2"
}
```

Figure 3.24: An `AuthorizeKnowledgeExchange` message to communicate with Player denoted by "receiverGuid".

```
{
    "action": "exchange",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "denied",
    "rejectDuration": "<single|permanent>"
}
```

Figure 3.25: A `RejectKnowledgeExchange` message. Player can be banned permanently from talking with non-existing Players.

```
{
    "action": "exchange",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "result": "OK"
}
```

Figure 3.26: An `AcceptKnowledgeExchange` message.

```
{
    "action": "send",
    "userGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c0",
    "receiverGuid": "0a041a66-01b7-40cf-a06a-a47550f0e6c2",
    "fields": [
        {
            "x": <0|1|2|3|4|...|board width specified in the config JSON -1>,
            "y": <0|1|2|3|4|...|board height specified in the config JSON -1>
            "value": {
                "manhattanDistance": <null|0|1|2|...|Nearest piece M. dist.>,
                "contains": "<goal|sham|valid|piece|empty|unknown>",
                "timestamp": 1337,
                "userGuid": "fa3f3cb5-742d-478f-9e02-230679032777"
            }
        },
        {
            "x": <0|1|2|3|4|...|board width specified in the config JSON -1>,
            "y": <0|1|2|3|4|...|board height specified in the config JSON -1>
            "value": {
                "manhattanDistance": <null|0|1|2|...|Nearest piece M. dist.>,
                "contains": "<goal|sham|valid|piece|empty|unknown>",
                "timestamp": 1337,
                "userGuid": null
            }
        }
    ]
}
```

Figure 3.27: A `Data` message with a knowledge exchange/accept exchange response data. This time the "contains" field can contain "sham" and "valid" as a value. This is the only way information about a piece having been checked is being transferred. The maximal value of fields in the "fields" array is equal to the amount of fields in the neighborhood discovered by the Player (the amount of fields is only capped by a determined game implementation, it can be capped at 9, or go up to 20, but it's always at least 1)

Chapter 4

Recommended Unit Tests

The following Unit Tests should serve as the bare minimal amount of Unit Tests for the Game to make sure that everyone will have at least the same properties working. In order to determine that a Unit Test is successful, test it on a basic configuration of the board, that is: a board with default config JSON settings and default delays between actions.

The examples of Unit Tests involve:

- Testing the "Move" messages on Game Master's database, to make sure that:
  - A Player will not be able to get on a field occupied by another Player
  - A Player will not be able to get on a field that is outside of the boundaries of the Board,
- Testing if the Board was initialized with identital parameters to the ones specified inside the config JSON file,
- Testing if the config JSON file has been successfully loaded and it has a valid structure
- Testing how the game behaves if more than a specified amount of Players tries to join the game,
- Testing how many fields are going to be returned with Discover message if a Player stands on a corner, next to an edge and away from edges (4, 6 and 9 respectively),
- Testing if Discover message returns manhattanDistance for fields in Tasks Area or if it returns null for fields in Goals Area.