
Logo by Laura Leclerc

# Requirements Document

**Valentin Bonnet** **Pier-Luc Gagnon** **Etienne Perot**

# Table of Contents

***Note**: It is highly recommended to read this document in colors online:*
**http://tinyurl.com/everVoidRequirements**

# 1. Introduction

everVoid is a space based video game in which player(s) vie for control of the galaxy; it is a turn-based strategy game win which each player submits his/her turn simultaneously. The project is being developed by Etienne Perot, Valentin Bonnet and Pier-Luc Gagnon for McGill's Software Engineering Project course (COMP 361), spanning both the fall 2010 and winter 2011 semesters.

## 1.1 Purpose

This Software Requirements Specification (SRS) provides a description of all the functions and specifications of the everVoid multi-player turn-based space game; it is based on the IEEE 830 Standard.

## 1.2 Scope

The project is composed of two main components: a *Client* and a *Server*. *Clients* connect to everVoid *Server* in order to play a game. The *Server* is tasked with all game calculations while the *Client* simply relays information between the *Player* and the *Server*. For the purpose of this document, the *Client* and *Server* are considered a single *System*, which provides the user with the game experience. Interactions specific to *Host* - *Server* or *Player* - *Client* are discussed and marked as such.

## 1.3 Definitions, acronyms and abbreviations

### 1.3.1 Definitions

- **Player:** User playing an everVoid game by using the everVoid Graphical interface.
- **Host:** User hosting an everVoid server, has access to a console and can issue commands to the server. Can be a Player as well.
- **Client:** The everVoid Graphical Interface, capable of displaying a game state, connecting to an everVoid Server, and sending moves to the Server.
- **Server:** an everVoid server, accepting lobby creations, capable of accepting player move submissions and calculating the next turn.
- **Lobby:** Small virtual room where players wait until a game is launched.
- **Action:** a type of play that players can perform during a game

### 1.3.2 Glossary

| Term | Definition |
|---|---|
| Ack | Acknowledgement |
| A.I. or AI | Artificial Intelligence (also denotes robots) |
| API | Application Programming Interface |
| (G)UI | (Graphical) User Interface |
| IEEE | Institute of Electrical and Electronics Engineers |
| JVM | Java Virtual Machine |
| LAN | Local Area Network |
| SRS | Software Requirements Specification |
| TCP | Transfer Control Protocol |
| UML | Unified Modelling Language |

## 1.4 References


- IEEE Recommended Practice for Software Requirements Specifications. New York: Institute of Electrical and Electronics Engineers, 1998.


## 1.5 Overview

Chapter 2 provides a description of the everVoid project, its purpose, and flow. It is mostly textual and provides information as to what interactions a user could have with the everVoid game. Chapter 3 contains diagrams detailing the structure and state of the everVoid System, which were created following UML 2.0 specifications. This section details the process involved in each interaction mentioned in section 2.

# 2. Overall description

This section describes the overall scope, functionality, and constraints of the everVoid project.

## 2.1 Product perspective

everVoid will use external libraries for certain tasks (i.e. for graphics and communication). The Game logic and state representation will however be entirely constructed by the everVoid development team.

everVoid will feature a number of Graphical User Interfaces to allow the Player to communicate his intent to the the game:

- A root **Main Menu** will appear first when the game is launched, giving the player the possibility to join a lobby, change settings (volume, player name, screen resolution, etc.) and quit.
- A **Lobby interface** which allows the players to communicate with one another while waiting for other players. This interface also allows the Lobby Master to choose a map, add additional players/AIs, load a game from a save file, and start the game.
- A game **User Interface** resembling the interface of common mainstream real-time-strategy games (for familiarity and intuitiveness) allowing the player to issue commands to units, control camera movement, access in-game menus (such as the research menu, planet menu, etc.) and communicate with other players.
- **Game Menu** that enables saving, enabling or disabling sound and/or music and leaving the game (i.e. returning to root main menu).
- Multiple in-game views such as Research view, Planet view, Solar System view, and Galaxy view. These will allow the user to change state of their respective game elements.

Because everVoid will be written in Java, it will make heavy use of the Java Virtual Machine and thus Java must be installed on the machine to play. For this same reason, everVoid will work properly on operating systems with Java support as it will not interface with them directly.

## 2.2 Product functions

This section provides a summary of the different functions available to users and hosts to allow them to control the flow of the everVoid software. For convenience, a list of the use cases is provided, followed by the actual use cases diagram in section 2.2.1. The full use cases can be found later in section 3.3.1.

Summary level:

- Play a Game
- Run a Server

User Goal level:

- Create a Server
- Set up a Lobby
- Join a lobby
- Start a Game
- Save a Game
- Load a Game
- Play a Turn
- Leave a Game

Subfunction level:

- Send a chat message
- Kick a player
- Change Server Status

### 2.2.1 Use Cases

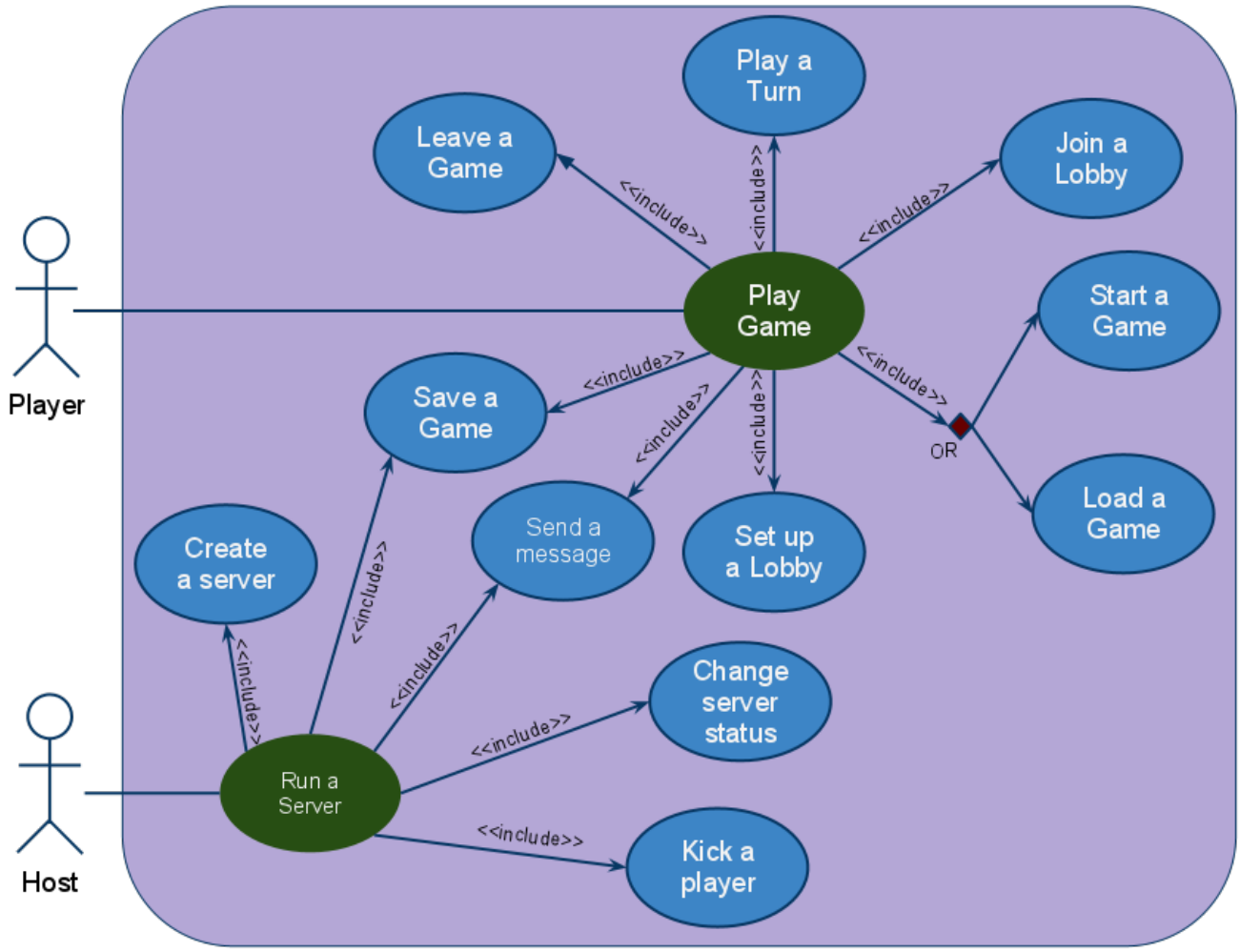



Figure 1: everVoid use case diagram

This use case diagrams represents different goals that different users might have during the execution of the software. The Players' main success scenario is playing a game, while the Hosts' is running an everVoid server; both involves multiple possible sub-cases.

## 2.3 User Characteristics

Players are expected to be knowledgeable about how the Internet works and know what a computer address is (in order to play multi-player). They are also expected to have some basic game knowledge. The interface will use elements established in many games, players are assumed to be comfortable with these features.

The hosts are expected to be very knowledgeable about how the Internet and client/ server models work, as well and know how to open, edit and save server configuration files (text files). They are also expected to know how to use a console to control a server.

## 2.4 Constraints

Because memory and processing powers are limited, certain aspects of the game might have to be limited in order to provide the user with a pleasant game experience. For instance, the number of units a player can possess will have to be limited to prevent overloading the computer.

To prevent cheating, the validation of player actions will be done on the server side. If cheating is caught the Server will save the previous game state, warn all clients, and proceed to end the game.

Certain factors are outside of the control of the everVoid game (ie. disconnection from the Internet, computer shuts down...). To mitigate the risk of untimely shutdown, the game will save state to a temporary file periodically.

## 2.5 Assumptions and Dependencies

It is assumed, and vital for the application, that the Java Virtual Machine is installed on the computer trying to run everVoid. The final product will bundle the JVM with the installer, avoiding this issue.

The product relies on the jMonkeyEngine library, which will also be bundled with the product.

The ability to connect to a live everVoid Server is also assumed for the Client, i.e. no firewall or other applications are preventing the client and the server from communicating.

## 2.6 Risks and Apportioning of requirements

Lack of experience of the development team is certainly a big factor in the development of everVoid. Time constraints might lead to certain features getting delayed until future versions of the game. Because the project is also planned by this same inexperienced team, project expectations might have been set too high, leading to features having to be removed, delayed or simplified.

In particular, non-gameplay-critical features such as sounds might not be ready in time, or compromises might be made regarding varying AI personalities. Additionally, certain units or buildings with more complex logic (transport ships, movable base station) might also not be implemented in the first release.

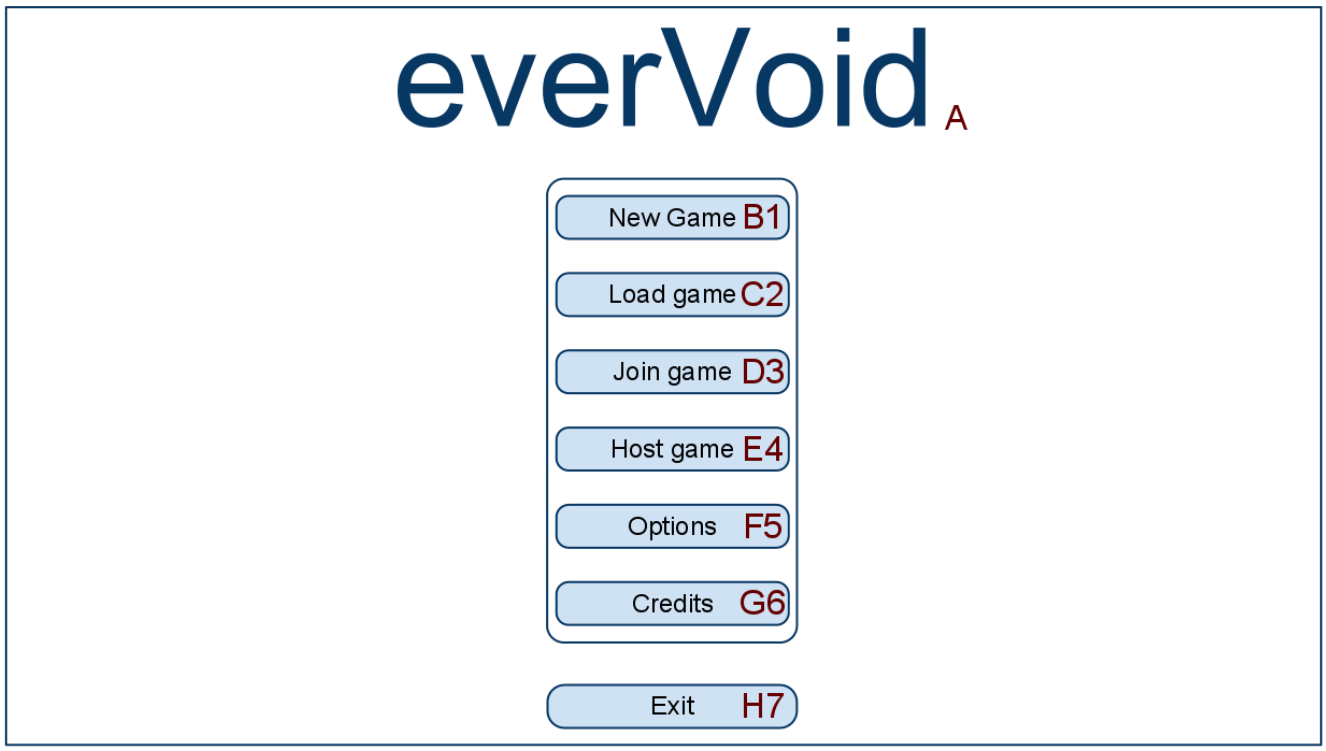
If necessary, the Galaxy View might also not be implemented, as all other views feature a minimal version of the galaxy view, bypassing the need for a full-screen one without reducing user functionality.

# 3. Specific requirements

This section describes some implementation requirements about the everVoid System.

## 3.1 User Interface requirements

### Main Menu




A: everVoid logo
B: New Game: Starts a new single-player game lobby
C: Load a game from a save file
D: Join an online or a LAN game
E: Create a new multiplayer lobby
F: Adjust options (screen resolution, audio volume, player name, etc)
G: Show everVoid credits
H: Quit the game

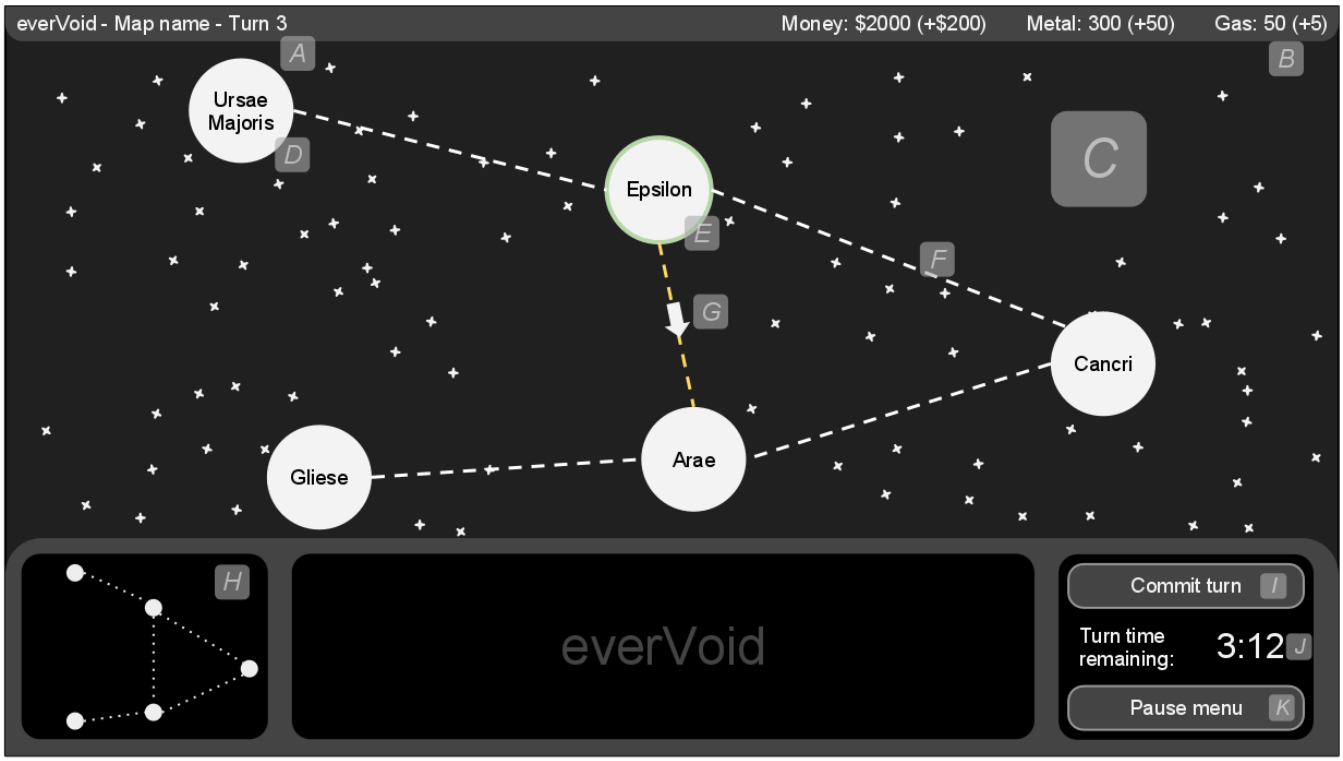
Figure 2: everVoid interface mockup: Main Menu

## Lobby Interface


A: Displays server info (lobby name, player name)
B: Leave lobby button: Makes the player exit the lobby
C: Lobby menu, where lobby parameters are set. All players may see this menu, but only the lobby master can modify it
D: The lobby's name
E: The map that will be played once the game is started
F: The number of teams in the lobby
G: The number of available player slots (limited by the map)
H: Player slots area. There is a row for each player slot
I: Slot number
J: Player name
K: The player may choose his faction from a dropdown menu
L: The player may choose his team from a dropdown menu
M: This symbol indicates that this player is the lobby master
N: The player may annoucne that he is ready to play by checking this checkbox
O: Another human player slot. The lobby master may not modify this row.
P: This checkbox indicates Player2's readiness. The lobby master cannot check this
Q: An AI slot. The lobby master may select the AI profile from the dropdown menu
R: The AI player's faction. The lobby master may select this from the dropdown menu
S: The AI player's team. The lobby master may select this from the dropdown men
T: An open player slot. An additional human player may join and will be assigned to this slot, or the lobby master may choose an AI profile to fill this slot from the dropdown menu
U: Chat area. Player messages show up here
V: Message input box; the player may type a message here
W: Send button: Sends the typed message to the chat area. The player may also press Enter instead of pressing this button
X: Load game: Prompts the lobby master for a save file, which overwrites all lobby parameters to those specified in the save file
Y: Start game: Starts the game. Only available when all players are ready.

Figure 3: everVoid interface mockup: Lobby

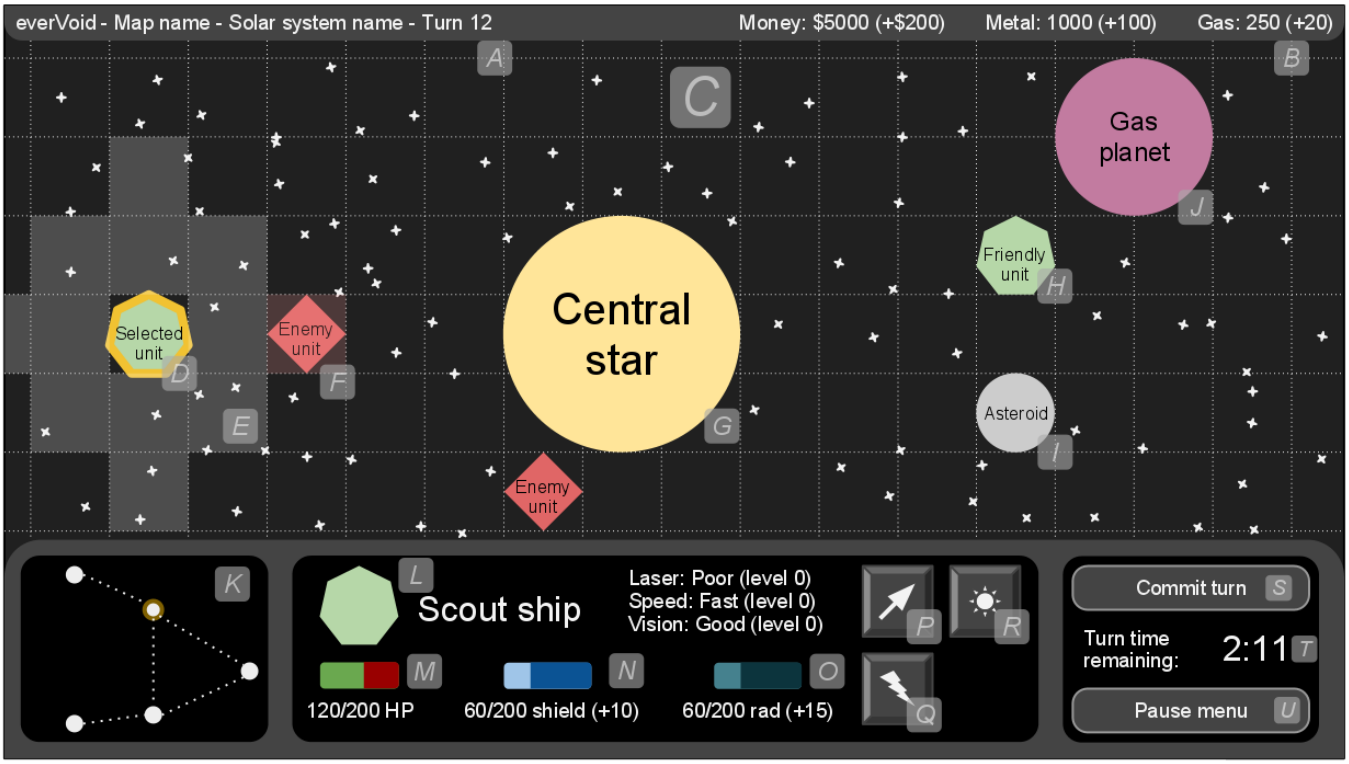
## Galaxy View




A: Displays game info (map name, current turn)
B: Displays the player's resources, and income per turn in parentheses
C: Galaxy view, where all the solar systems are displayed
D: A solar system. The player may click on it to switch to Solar System view.
E: A solar system in which the player has buildings or units (solar system is outlined)
F: A wormhole from a solar system to another
G: A wormhole in which units belonging to the player are going through.
H: A minimal solar system view, representing solar systems and wormholes.
I: Commit turn button: Ends the player's turn and sends the moves to the server.
J: Remaining time for the player to finish his turn. If the time is elapsed, his turn is automatically committed.
K: Pause menu button: Brings up the pause menu.

Figure 4: everVoid interface mockup: Galaxy view

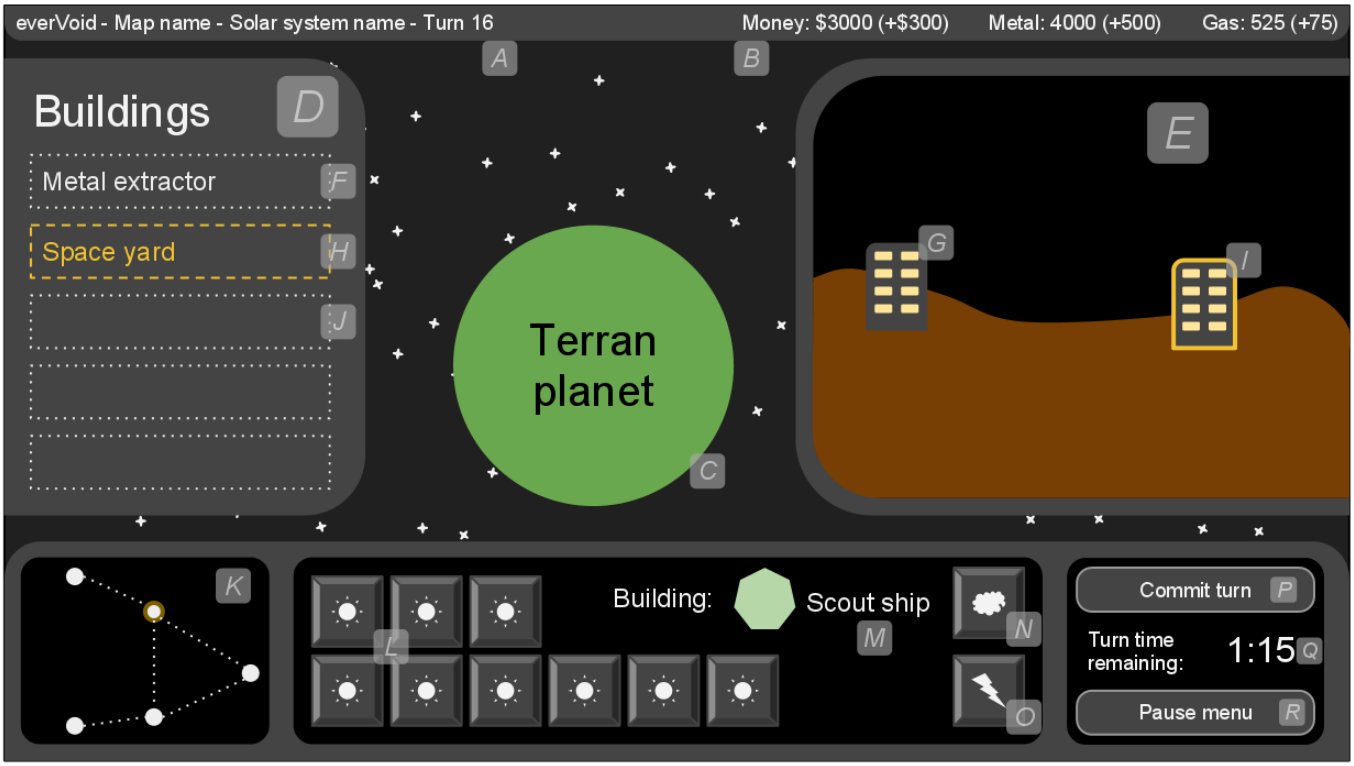
## Solar System View




A: Displays game info (map name, solar system being viewed, current turn)
B: Displays the player's resources, and income per turn in parentheses
C: Solar system grid, where the action takes place
D: A selected friendly unit, with a border around it to indicate that it is being selected
E: The areas where the ship can move to are highlighted. The units that the ship can attack are also highlighted.
F: An enemy unit.
G: The solar system's central star, from which radiation emanates
H: A non-selected friendly unit
I: An asteroid (decoration, non-free space for ships)
J: A gas planet. The player may click it to switch to Planet view.
K: A minimal solar system view, representing solar systems and wormholes. The solar system being viewed is highlighted.
L: Selected unit view. Shows unit image and type
M: Selected unit's health
N: Selected unit's shields and shield regeneration rate
O: Selected unit's radiation level and radiation regeneration rate
P: Move button. The player may press this button and select a square in which to move in order to initiate a move to that square.
Q: Delete unit button: Destroys this unit.
R: Deploy probe button (special Scout ship ability). Ships may have special abilities, with associated buttons represented here.
S: Commit turn button: Ends the player's turn and sends the moves to the server.
T: Remaining time for the player to finish his turn. If the time is elapsed, his turn is automatically committed.
U: Pause menu button: Brings up the pause menu.

Figure 5: everVoid interface mockup: Solar System view

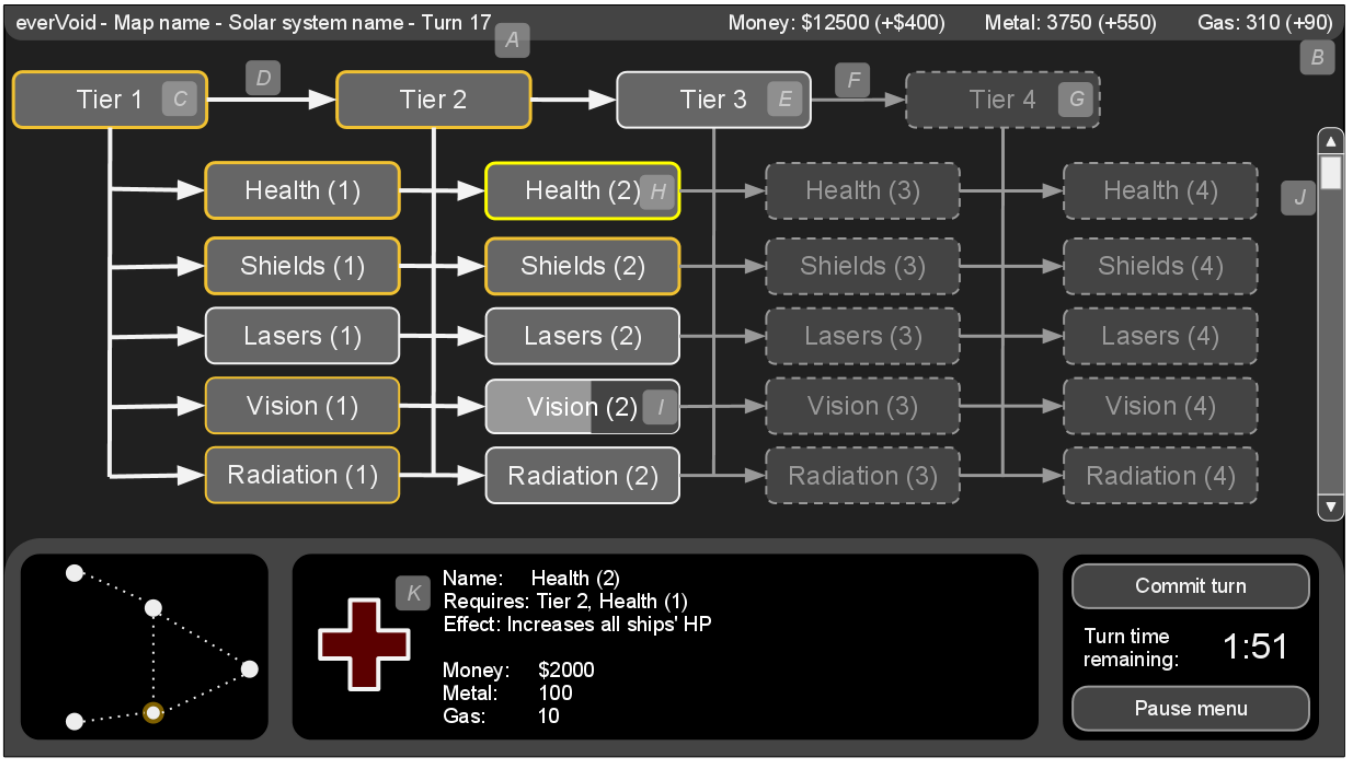
## Planet View




A: Displays game info (map name, solar system in which the planet is, current turn)
B: Displays the player's resources, and income per turn in parentheses
C: Close-up of the planet being viewed
D: A panel containing the list of building slots the planet has, and the buildings built on the planet.
E: A visual representation of the planet's surface.
F: An unselected building in a building slot
G: Visual representation of the unselected building
H: A selected building in a building slot
I: Visual representation of the selected building
J: An empty building slot. The player may click it and select a building to build in that slot using the bottom panel.
K: A minimal solar system view, representing solar systems and wormholes. The solar system being viewed is highlighted.
L: Buttons corresponding to the selected building. For a ship-constructing building, this is a list of ships that the building may build.
M: The ship currently being constructed by the building.
N: Cancel ship construction button: Cancels the construction of the ship being built. Refunds resources, and allows the player to build another ship.
O: Delete building button: deletes the building.
P: Commit turn button: Ends the player's turn and sends the moves to the server.
Q: Remaining time for the player to finish his turn. If the time is elapsed, his turn is automatically committed.
R: Pause menu button: Brings up the pause menu.

Figure 6: everVoid interface mockup: Planet view

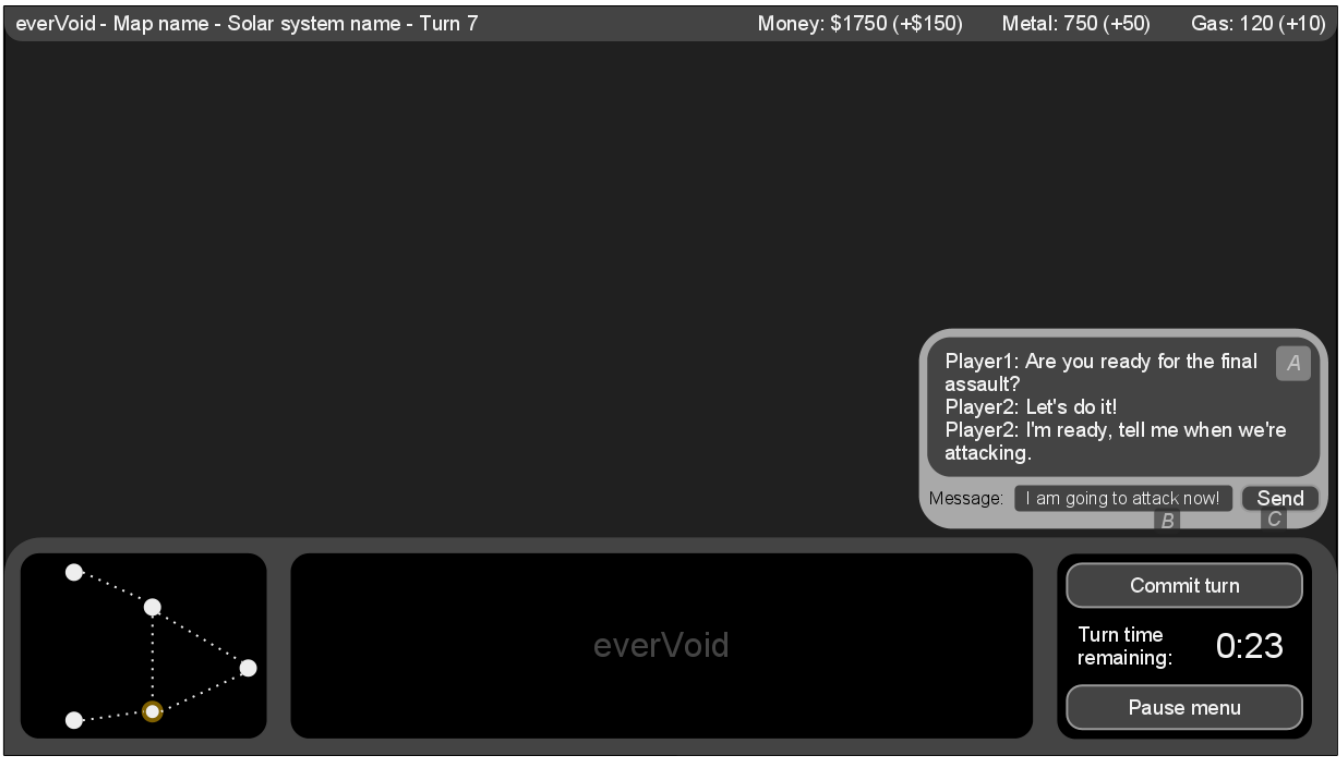
## Research View




A: Top bar displaying game information (turn number, etc.)
B: Displays player resources and income on next turn (in parentheses)
C: An already-researched upgrade has a special outline
D: An available upgrade path shows a full arrow
E: An available, not-researched upgrade shows as a regular button. The user may click to research it.
F: An unavailable upgrade path. The user must research other things before being able to access this path
G: An unavailable upgrade. The user must satisfy all linked upgrade paths before being able to research this upgrade.
H: An upgrade being hovered by the cursor shows a special outline, and corresponding information is displayed in the bottom area.
I: An upgrade currently being researched. The partial upgrade progress is represented by the progress bar.
J: Scrollbar that the player may use to view additional research paths at the bottom
K: Information about the selected upgrade is displayed (Name, research requirements, effect, resource cost)

Figure 7: everVoid interface mockup: Research view

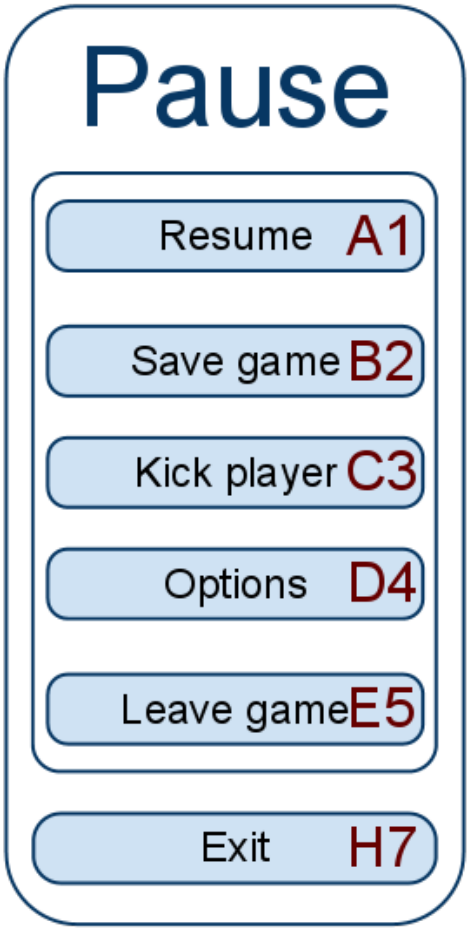
## In-game Chat Message




A: The main in-game chat window. Shows up as an overlay when the user presses a key.
Messages show up in the central box. When the player doesn't have the chat box displayed but another player sends a message, the received message is displayed without an overlay, directly on the game board.
B: Message input box. The player may type his message here.
C: Send button. This sends the typed message to the other players and displays it in the chat box. The player may also use the Enter key to send a message.

Figure 8: everVoid interface mockup: In-game chatting

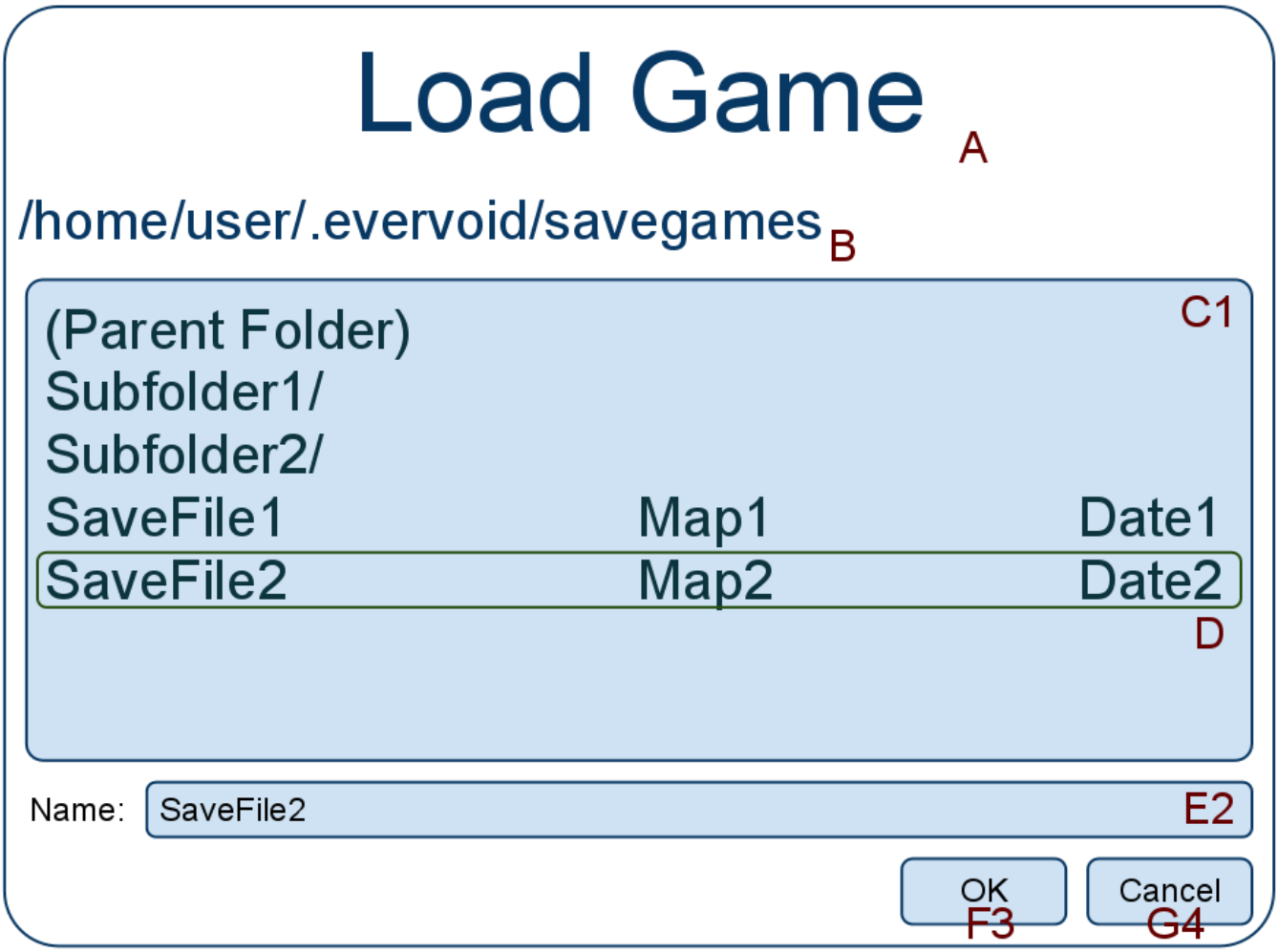
**Pause Menu**




A: Resume game (hides the pause menu)
B: Save game (requests a save file from the server, and opens the save file dialog to save it)
C: Select a player to kick from the game (Only available if the player is the lobby "master")
D: Adjust options (screen resolution, audio volume, player name, etc)
E: Leave game (return to lobby)
F: Quit everVoid completely

Figure 9: everVoid interface mockup: Pause menu

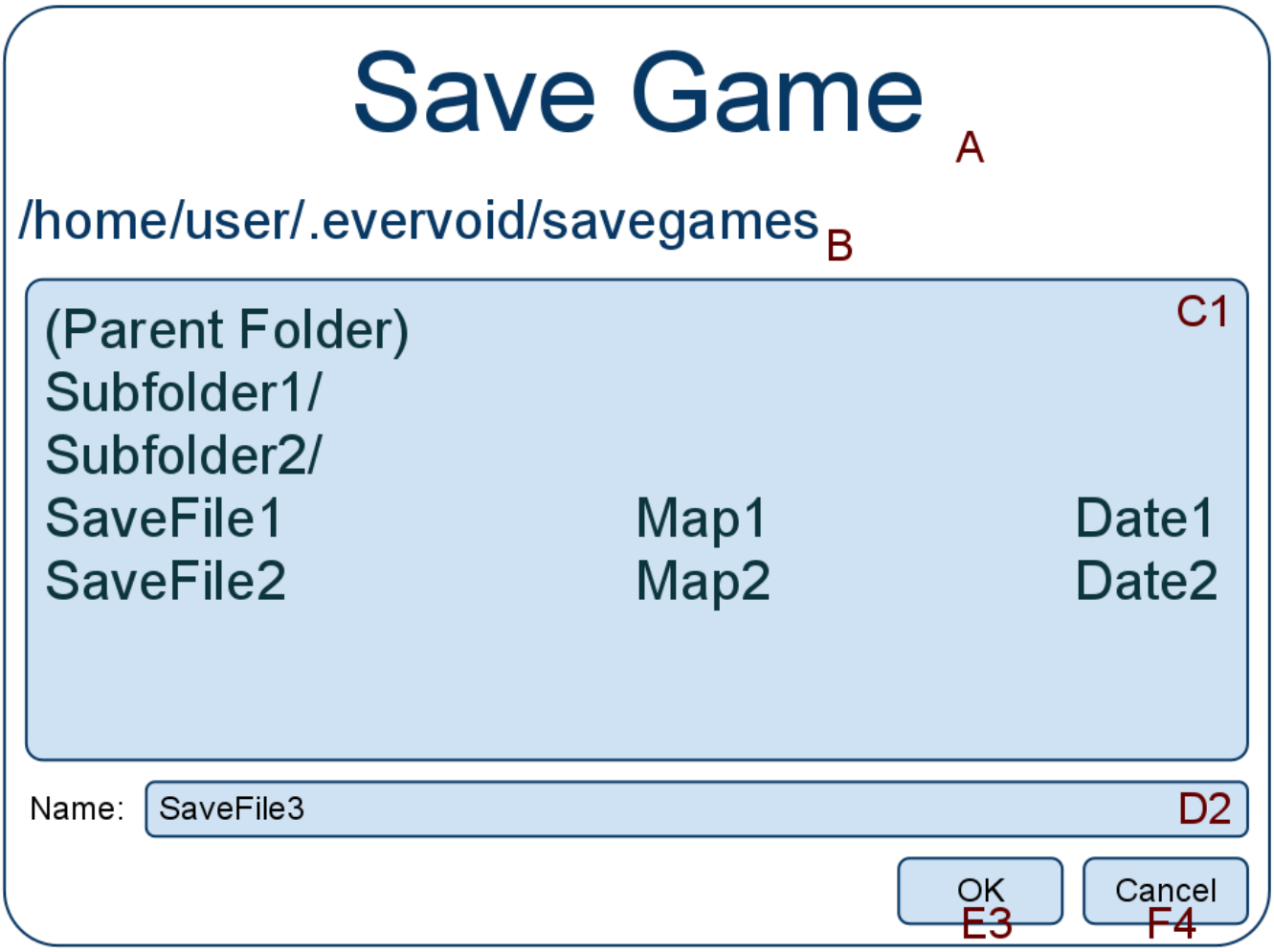
**Load game**




A: Dialog title
B: Directory in which to look for save files
C: File manager; shows save files in selected directory, and allows filesystem exploration
D: Selected file is highlighted
E: Filename of the desired file to load
F: Confirm loading of selected file
G: Cancel loading dialog

Figure 10: everVoid interface mockup: Load game dialog

**Save Game**




A: Dialog title
B: Directory where save files will be placed
C: File manager; shows save files in selected directory, and allows filesystem exploration
D: Filename of the desired save file
E: Confirm name
F: Cancel saving dialog

Figure 11: everVoid interface mockup: Save game dialog

## 3.2 Structural requirements

### 3.2.1 Distributed Architecture

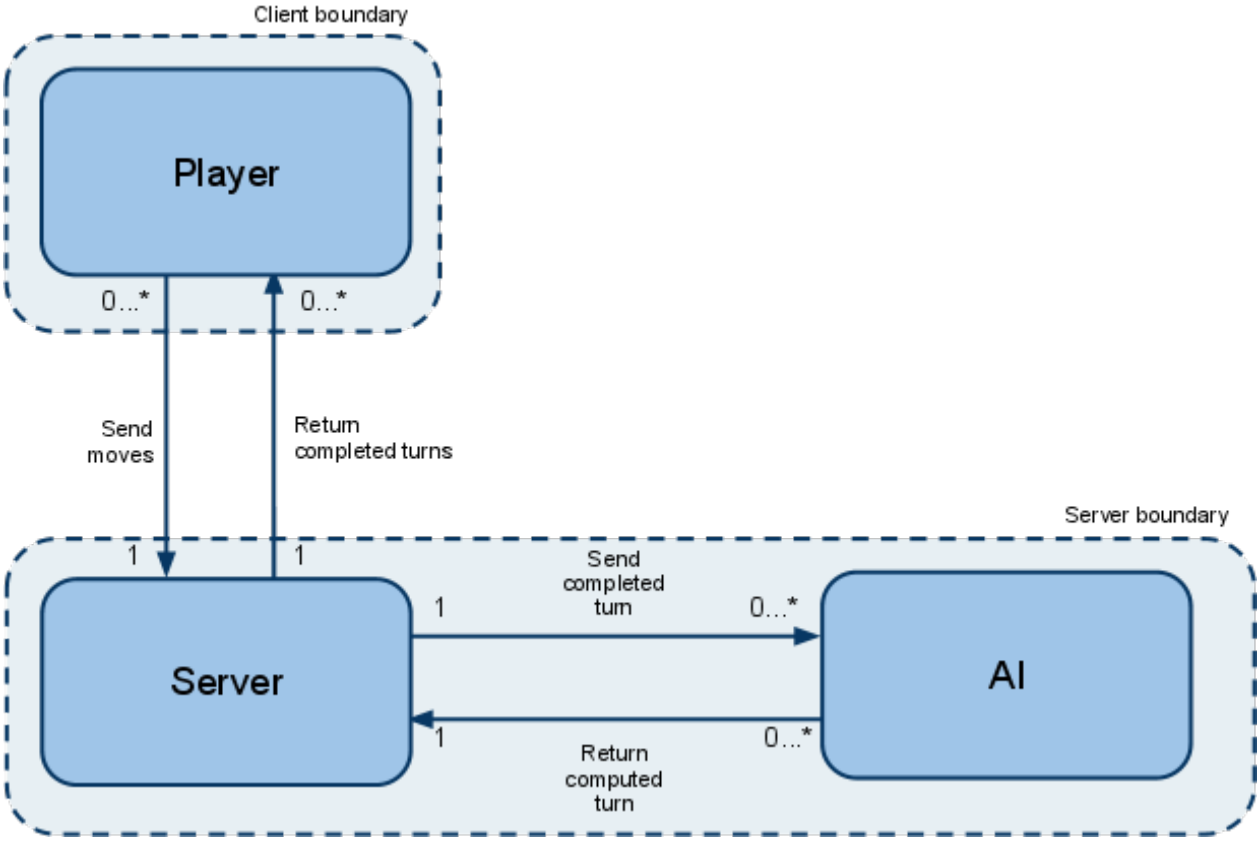



Figure 12: everVoid client/server boundaries
This diagram highlights the structural boundaries between the client-server communication.

To support multi-player interactions, everVoid will be built using a client/server distributed architecture. The client executable contains the full game engine and the graphical user interface. The server executable will not contain a graphical user interface, all the interaction are made using text commands through a console. In order to allow the server to verify turn validity, it will contain a copy of the game engine. Checking that the local player's turn is valid is not necessary on the client side, because the Graphical User Interface will not allow players to make invalid moves. However, each client will check other players' turns. Both client and server can save and load the game, but only the server is able to load a game in the current lobby (player will be able to load the game offline). Although the server is intended to run on a dedicated computer, it is entirely possible to run both client and server programs on the same machine, to allow offline play or simply host a server and be able to play on it using the same computer.

### 3.2.2 Environment Model

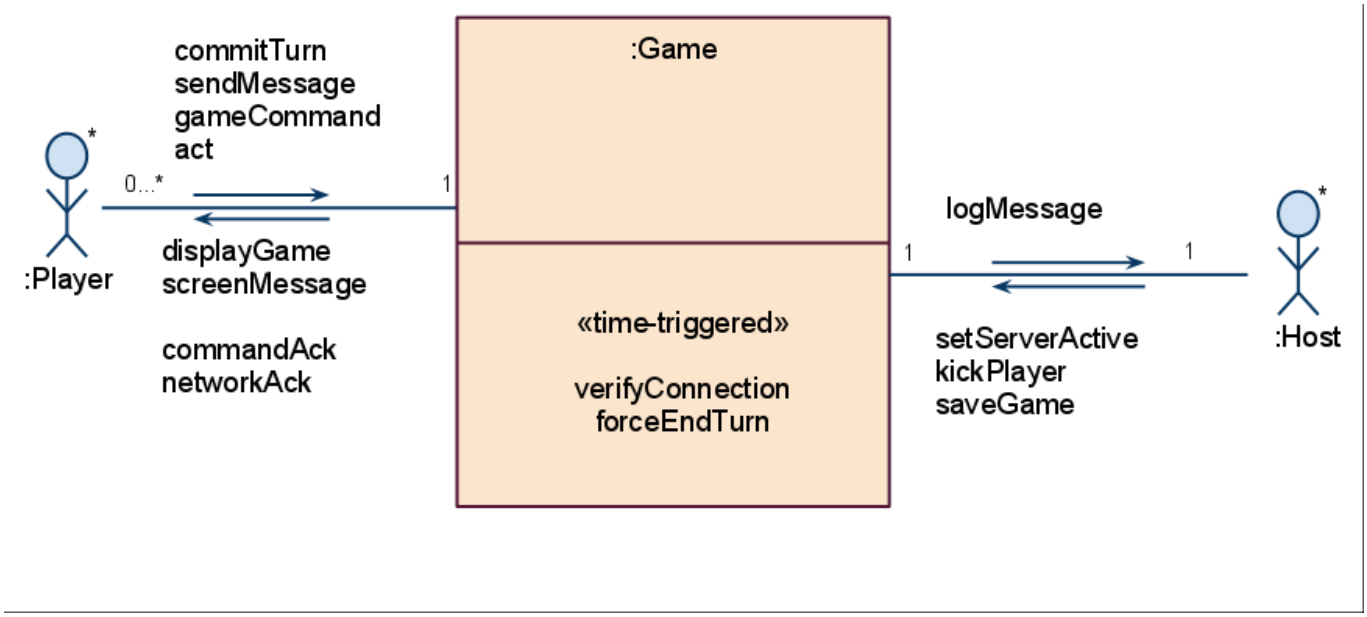



Figure 13: everVoid environment model

This diagram describes the possible interactions between the System and external actors.

The different commands and their parameters are described below:

**Input Messages**

- `commitTurn(actions: List<Action>)`
- `sendMessage(message: String, player: Player)`
- `gameCommand(command: Command, parameter: Parameter)` (This is an abstraction for different possible game commands, such as etc.)
- `act(action: Action)`
- `setServerActive(active: Boolean)`
- `kickPlayer(player: Player)`
- `saveGame(location: String)`

**Output Messages**

- `displayGame()` (Update the User Interface to reflect what is currently happening)
- `screenMessage(message: String)`
- `commandAck()` (Acknowledges the user's command visually)
- `networkAck()`
- `logMessage(message: String)` (Any message to be shown in the console)

**Message types**

- `Action`: An action that a player may perform during a turn that is related to

the actual game world.

Action Types: {move ship, attack, initiate research, construct a building, construct a ship, use a ship ability, colonize a planet, use hyperjump}

- `Command`: A task that a player may execute while in everVoid.

  Command Types: {create a server, change server status, join a lobby, leave a lobby, start a game, end a game, save a game, load a game, quit}

### 3.2.3 Concept Model

The following UML diagram (figure 14) describes the internal representation of the everVoid game state. The Game object represents the game itself, and contains Players and Solar Systems, themselves containing references to other objects, representing the whole game state. ShipContainer is an interface which lets Solar Systems and Wormholes contain Ships. The entire Game State can be serialized and sent over the network from Client to Server. **Note** that Enumeration types (ship types, resource types, building types, research types) are defined statically, thus not part of the game state.

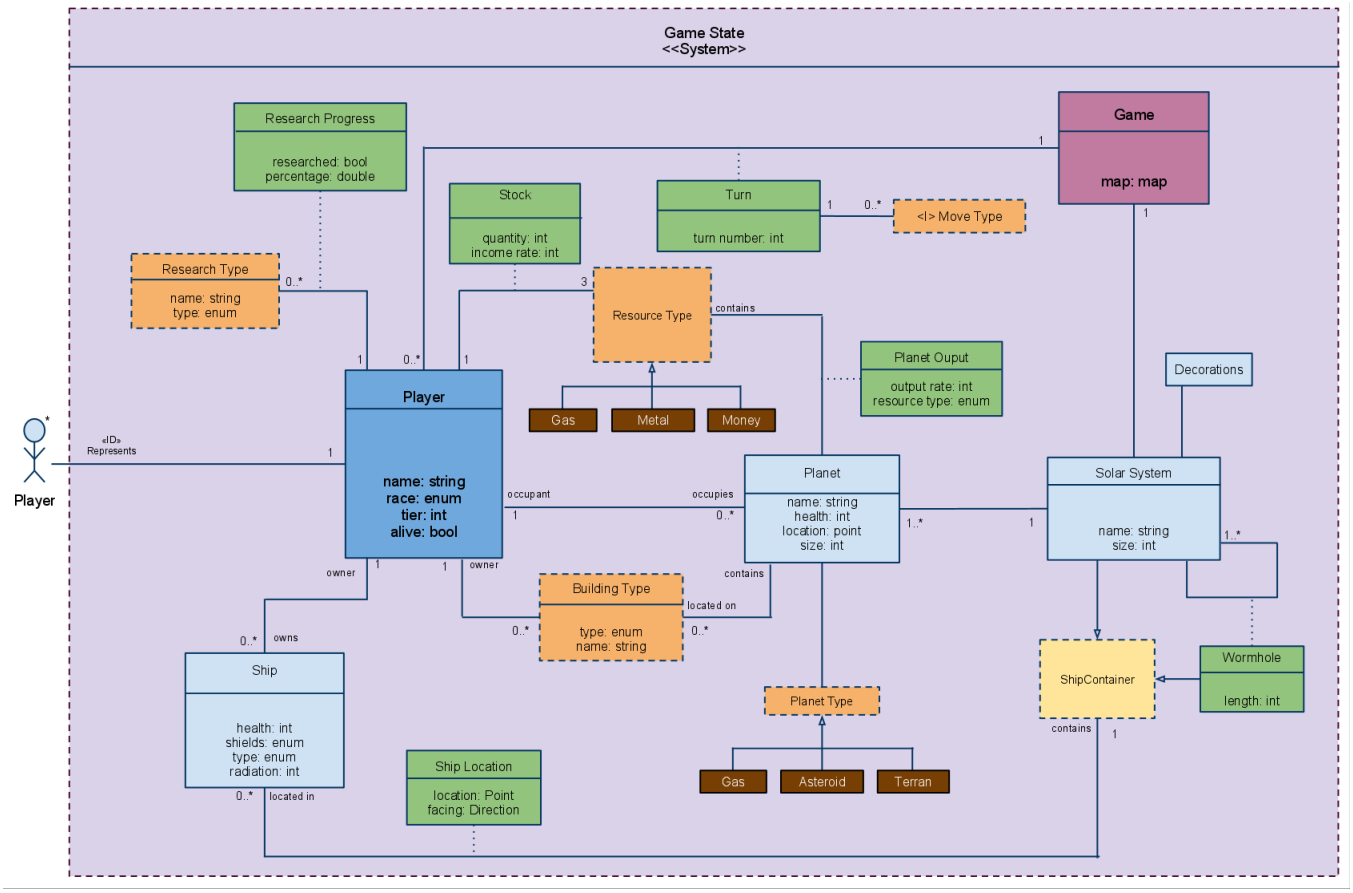



Figure 14: everVoid concept model

A bigger version of the diagram can be found at: [http://tinyurl.com/everVoidConceptModel](http://tinyurl.com/everVoidConceptModel)

## 3.3 Behavioural requirements

### 3.3.1 Use Cases

**Use Case:** Play a Game
**Scope**: Entire System
**Level**: Summary Level
**Intention**: The Player wants to play a game of everVoid.
**Multiplicity**: Can only play one Game
**Primary actor**: Player
**Secondary actors**:
**Precondition**: Game is installed on Player's machine.
**Main success scenario**:

Steps 4, *5, 6 can be repeated any number of times and can be done in any order*

1. *Player* launches the Client.
2. *Player* joins a Lobby.
3. *Player* starts the Game.
4. *Player* plays a turn.
5. Player sends a message.
6. Player saves the Game.
7. Player leaves the game.

**Extensions**:

2a. *Player* does not want to join an existing Lobby, go to set up a Lobby and continue with step 3.
3a. *Player* informs the *System* that he wants to load a game, go to the Loading a Game use case. Return to step 4.

**Use Case:** Run a Server
**Scope**: Server Subsystem
**Level**: Summary Level
**Intention**: A Host wishes to create a server on which players can connect to play a game
**Multiplicity**: Only one server per machine
**Primary actor**: Host
**Secondary actors**:
**Precondition**: Host is connected to the Internet and has properly configured the Server files
**Main success scenario**:
Steps 2, 3, 4, 5 *can be repeated any number of times and can be done in any order*
1. *Host* creates a Server
2. *Host* saves the game
3. *Host* changes server status
4. *Host* sends a message to one or all of the players
5. *Host* kicks a player

**Extensions**:
1a. Server is already running on the machine, use case ends in failure.

---

**Use Case:** Create a Server
**Scope**: Server-side
**Level**: User Goal
**Intention**: A host wishes to create a dedicated everVoid server.
**Multiplicity**: Can only create one Server
**Primary actor:** Host
**Secondary actors**: Console
**Precondition**: None
**Main success scenario**:
*The Host wishes to create a standalone, dedicated everVoid Server.*
1. The *Host* creates a configuration file for the *System* (server name, server port, maximum number of players, etc.)
2. The *System* launches the *server*.
3. The *System* loads the configuration file.
4. The *Host* informs the *System* that he wants to activate the server (make it public).
5. *Players* may now connect to the *Host's* dedicated server.

**Extensions**:
3a Server refuses to start if the configuration file is malformed or contains invalid values. Use case ends in failure.

**Use Case:** Set up a Lobby
**Scope**: Client-side: Lobby
**Level**: User Goal
**Intention**: The user wants to create a lobby on a server.
**Multiplicity**: Can only create one Game at a time
**Primary actor**: Player
**Secondary actors**: Lobby Interface
**Precondition**: Server must be active; no game in progress
**Main success scenario**:

*Player requests server to create a new game given some information. The new Game is then returned and can be played.*

1. *Player* joins a Lobby as "lobby master".
2. *Player* sets Lobby variables (map, number of players, name, etc).
3. *Player* waits for Lobby to fill (players may join the lobby, and Player may add AIs in the player slots).

**Extensions**:

2a. *System* ascertains that the provided game name is invalid.
2a.1. *Game Engine* informs the player and changes back the lobby name (step 2).
2b. *Player* informs the *system* that he wants to load a game, go to Loading a Game use case

---

**Use Case:** Join a Lobby
**Scope**: Client-side: Lobby
**Level**: User Goal
**Intention**: The player wishes to join a lobby that is already hosted.
**Multiplicity**: Multiple users per server.
**Primary actor**: Player
**Secondary actors**: Lobby interface
**Precondition**: User is connected to the Internet; no Game is running
**Main success scenario**:

1. *Player* informs the *System* that he wants to join a lobby.
2. *Player* provides the *System* with the address of a server.
3. *System* attempts to connect to the specified *Server.*
4. *Server* accepts the *System's* request, and sends back information about the lobby.
5. The *System* switches to the lobby view, and displays the information about the lobby (map name, players connected).

**Extensions**:

3a. Connection might fail (server is offline, wrong address, firewall blocking the connection, etc.), the *Player* is notified and the use case ends in failure.
4a. The server may reject the client's request for various reasons: The server and the client are not running the same version of the game, the server is full, or a game is already taking place on the server. An appropriate error message is displayed. Use case ends in failure.

**Use Case:** Start a Game
**Scope**: Client-side
**Level**: User Goal
**Intention**: The user wants to start the game once his lobby is ready.
**Multiplicity**: Can only start one Game
**Primary actor**: Player
**Secondary actors**: Lobby Interface
**Precondition**: Player must be in a lobby as master; all human players must be ready.
**Main success scenario**:

1. *Player* requests *System* to start the game with the specified parameters.
2. *System* verifies that the specified parameters are valid and that the lobby slots are full.
3. *System* locks the lobby.
4. *System* starts the game.

**Extensions**:

2a. Some lobby slots are empty: *System* fills them with the default AI.
2b. *System* ascertains that the specified parameters are incorrect: *System* warns the player and use case ends in failure.

---

**Use Case:** Save a Game
**Scope**: Client-side
**Level**: User Goal
**Intention**: The user wishes to save the game in order to resume it at a later time.
**Multiplicity**: The Client can only save one game state at a time.
**Primary actor**: User
**Secondary actors**: Save/Load interface or Console
**Precondition**: There is an unfinished game in progress.
**Main success scenario**:

*User saves the current game state to a file.*

1. *User* informs the *System* that he wants to save.
2. *User* provides the *System* with a name for the save file.
3. *System* saves game to disk.

**Extensions**:

2a. Name is not valid: User is prompted for another name. Return to step 2.
3a. Saving fails: User is warned. Use case ends in failure.

**Use Case:** Load a Game
**Scope**: Entire system
**Level**: User Goal
**Intention**: User wishes to load an existing game from save file
**Multiplicity**: Can only load one game
**Primary actor**: User
**Secondary actors**: Save/Load Interface
**Precondition**: Save file exists; save file is a valid game; User is in the lobby as master
**Main success scenario**:

*User loads a saved game state from file and starts a game from that state.*

1. *User* informs the *System* that he wants to load a game.
2. *System* displays a list of saved games.
3. *User* informs the *System* of which save file he wants to load.
4. *System* loads the selected save file and loads a Lobby.
5. *System* waits for all players to join the Lobby and be ready, then *System* loads the game.

**Extensions**:

1a. There are no saved games - *System informs* the *user*. Use case ends in failure.
4a. File is not a valid save file - *System* informs the *user*, return to step 2.

---

**Use Case:** Play a Turn
**Scope**: Client-Side
**Level**: User Goal
**Intention**: The Player wishes to play a turn
**Multiplicity**: Multiple users per server.
**Primary actor**: Player
**Secondary actors**: Graphical User Interface
**Precondition**: A game is currently running, enough players are connected
**Main success scenario**:

1. *Player* informs *System* of Actions he wishes to preform.
2. System approves Action list.
3. System calculates new turn, and displays it to the Player.

**Extensions**:

1a. The time allotted to make a turn has expired, *System* sends incomplete list of actions and use case resumes from step 2.
2a The Action list contains an invalid move: *Player* is warned, *Player*'s turn is discarded. Use case ends in failure.

**Use Case:** Leave a Game
**Scope**: System
**Level**: User Goal
**Intention**: The player wishes to leave the current game.
**Multiplicity**: Multiple players per server; can only quit once
**Primary actor**: User
**Precondition**: User is connected to a server
**Main success scenario**:
1. *Player* informs the *System* of his desire to leave the game.
2. *System* asks Player for confirmation.
3. *System* removes Player from the Game and notifies remaining players.

**Extensions**:
2a. *Player* cancels, use case ends in failure
3a. No more players, return to lobby.

---

**Use Case:** Kick a player
**Scope**: Server-Side
**Level**: Subfunction
**Intention**: The host wishes to kick a player
**Multiplicity**: The host is unique.
**Primary actor**: Host
**Secondary actors**: Console
**Precondition**: A game is currently running, players are connected
**Main success scenario**:
1. *Host* informs *System* that he would like to kick a player.
2. *Server* kicks the player, and notifies all connected parties.

**Extensions**:
1a. The specified player is not on the server, the *Host* is notified and the use case ends in failure.

**Use Case:** Send a Message
**Scope**: Entire System
**Level**: Subfunction
**Intention**: A user wishes to send a message to another users
**Multiplicity**: Multiple users per server.
**Primary actor**: User
**Secondary actors**: Graphical User Interface or Console
**Precondition**: A game is currently running, at least 2 players are connected
**Main success scenario**:
1. *User* informs *System* he would like to send a message to a player.
2. *User* informs *System* about the message he'd like to send.
3. *System* relays the message to the other player(s).

**Extensions**:
3a Player is not connected, use case ends in failure.
3b Message formatting is wrong, use case ends in failure.

---

**Use Case:** Changing Server Status
**Scope**: Server
**Level**: Subfunction
**Intention**: The host wants to change the status of the server (activate or deactivate it).
**Primary actor**: Host
**Secondary actors**: Console
**Precondition**: The server is currently running
**Main success scenario**:
1. *Host* informs the system that he wants to change the status of the server.
2. *System* changes the status of the *server* to the specified status.

**Extensions**:
2a *Server* is currently in the specified states, nothing happens and the *Host* is informed. Use case ends in failure.

### 3.3.2 Operation Model

Here is a summary list of the different operations that can be performed:

- `commitTurn(actions: List<Action>)`
- `sendMessage(message: String, player: Player)`
- `gameCommand(command: Command, parameter: Parameter)`
- `act(action: Action)`
- `setServerActive(active: Boolean)`
- `kickPlayer(player: Player)`
- `saveGame(location: String)`

**Operation**: `Player::commitTurn(actions: List<Action>);`
**Scope**: Player; Game; Server;
**Messages**: Player::{commandAck}; Server::{networkCommand}; Player::{networkAck}
**Pre**: Player is in the Game and has a valid move to make, time is left to the turn
**Description**: The Player submits list of actions to the Client. The Client formats the actions and sends them to the Server. The Server verifies and then acknowledges all Actions in the list as valid.
**Use cases**: Playing a Turn

**Operation**: `Player::sendMessage(message: String; player: Player);`
**Scope**: Player; Server;
**Messages**: Player::{commandAck}; Server::{networkCommand}; Player::{networkAck}
**Pre**: Target player is in the game, Message is of correct format
**Description**: The player sends a text message and a player name to the server which is then relayed to the appropriate player.
**Use cases**: Playing a Turn

**Operation**: `Player::gameCommand(command: Command, parameter: Parameter);`
**Scope**: Player; Game;
**Messages**: Player::{commandAck}; Player::{screenMessage}
**Pre**: Command is valid
**Description**: The player sends a command to the software, telling the system to perform an action such as saving, loading a game, starting the game, changing a setting, etc.
**Use cases**: Setting up a Lobby, Joining a Lobby, Starting a Game, Saving a Game, Leaving a Game

**Operation**: `Player::act(action: Action)`
**Scope**: Player; Game;
**Messages**: Player::{commandAck}; Player::{displayGame}
**Pre**: Action is valid
**Description**: The player sends an action to the software, telling the system to perform a game-related action such as moving a unit, attacking, building, etc.
**Use cases**: Playing a Turn

**Operation**: `Host::setServerActive(active: Boolean)`
**Scope**: Host; Server;
**Messages**: Host::{logMessage};
**Pre**: Server has been properly configured, server is running
**Description**: The host activates or deactivates the server, meaning that the server can accept incoming connections or refuse all connections.
**Use cases**: Changing Server Status

**Operation**: `Host::kickPlayer(player: Player)`
**Scope**: Host; Server; Player
**Messages**: Host::{logMessage}; Player::{screenMessage}
**Pre**: Player exists and is in the game
**Description**: The host removes a player from the server.
**Use cases**: Kicking a player

**Operation**: `Host::saveGame(location: String)`
**Scope**: Host; Server; Game;
**Messages**: Host::{logMessage}
**Pre**: Game is currently running and is not finished.
**Description**: The host copies the current game state to disk at the specified location as backup or for later use.
**Use cases**: Saving a Game

**Operation**: `Player::saveGame(location: String)`
**Scope**: Player; Game;
**Messages**: Player::{commandAck}
**Pre**: Game is currently running and is not finished.
**Description**: The Player copies the current game state to disk at the specified location in order to resume the game from this state at a later time.
**Use cases**: Saving a Game

### 3.3.3 Protocol Model

The following diagram represents the different states the *Client* can be in, and which actions will make the system change state. It assumes an active everVoid *Server* exists.

A full-size version of the model can be found at: http://tinyurl.com/everVoid-Protocol

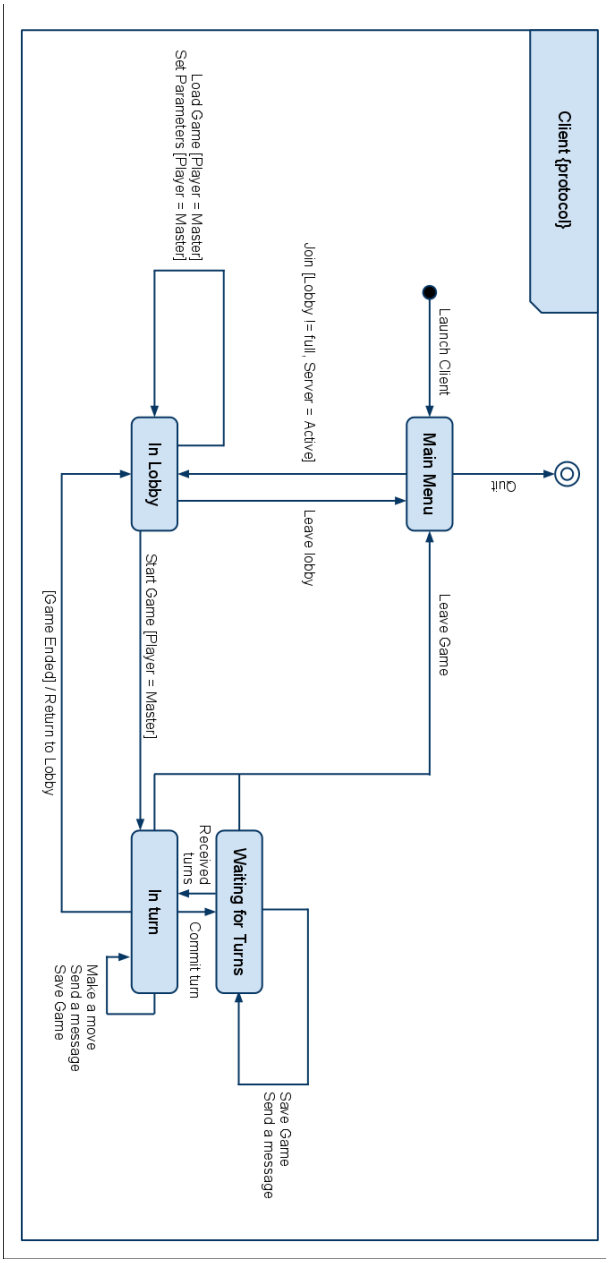



Figure 15: Protocol Model (Client)

The following diagram represents the different states the *Server* can be in, and which actions performed by the host will make the system change state.

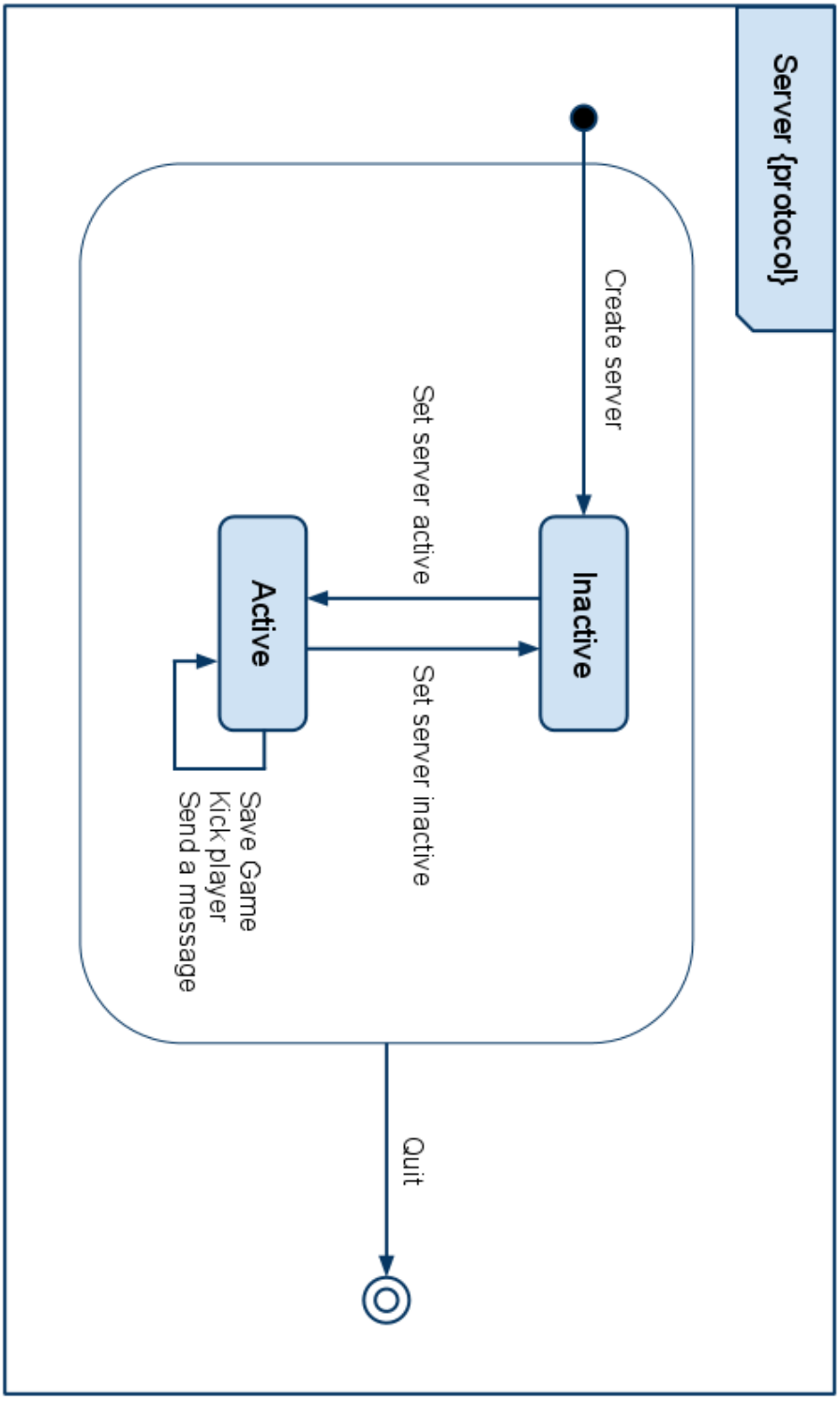



Figure 16: Protocol Model (Server)