

**Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)**

---

УТВЕРЖДАЮ  
Заведующий кафедрой ИУ7  
(Индекс)  
И. В. Рудаков  
(И.О.Фамилия)  
«        »              20        г.

**ЗАДАНИЕ  
на выполнение курсового проекта**

по дисциплине Протоколы вычислительных сетей

Студент группы ИУ7И-31М

Диас Сергей Рамирович  
(Фамилия, имя, отчество)

Тема курсового проекта Разработка протокола для пошаговой многопользовательской игры "Морской бой" на 2 и более игроков.

Направленность КП (учебный, исследовательский, практический, производственный, др.)  
учебный

Источник тематики (кафедра, предприятие, НИР) Кафедра

График выполнения проекта: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

**Задание**

Разработать протокол для взаимодействия игроков в игре. Описать механизмы взаимодействия между клиентами и сервером, включая передачу игровых ходов, синхронизацию игрового состояния, обработку ошибок и завершение игры. Разработать структуру сообщений протокола, определить их содержание и правила обработки. Разработать игру с использованием протокола.

**Оформление курсового проекта:**

Расчетно-пояснительная записка на 20-30 листах формата А4.

Расчетно-пояснительная записка должна содержать постановку задачи, введение, аналитическую, конструкторскую, технологическую части, заключение и список литературы.

Дата выдачи задания «        » 2024 г.

**Руководитель курсового проекта**

А.М.Никульшин  
(И.О.Фамилия)

**Студент**

С.Р.Диас  
(И.О.Фамилия)

(Подпись, дата)



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное  
учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)  
(МГТУ им. Н.Э. Баумана)»

---

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА  
К КУРСОВОЙ РАБОТЕ  
НА ТЕМУ:**

**«Разработка протокола для пошаговой  
многопользовательской игры "Морской бой" на 2  
и более игроков»**

Студент группы ИУ7И-31М

\_\_\_\_\_

(Подпись, дата)

С.Р. Диас

(И.О. Фамилия)

Руководитель

\_\_\_\_\_

(Подпись, дата)

А.М. Никульшин

(И.О. Фамилия)

2024 г.

# СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ .....</b>	<b>4</b>
<b>Аналитическая часть .....</b>	<b>5</b>
<b>1.1 Типы многопользовательских игр .....</b>	<b>5</b>
1.1.1 Игры с пошаговым режимом .....	5
1.1.2 Игры в реальном времени .....	5
<b>1.2 Протоколы для сетевого взаимодействия .....</b>	<b>5</b>
1.2.1 Протоколы транспортного уровня.....	5
1.2.2 Протоколы прикладного уровня .....	6
<b>Выводы.....</b>	<b>7</b>
<b>Конструкторская часть.....</b>	<b>8</b>
<b>2.1. Основные этапы протокола для пошаговой многопользовательской игры "Морской бой".....</b>	<b>8</b>
<b>2.2. Создание комнаты.....</b>	<b>9</b>
<b>2.3. Присоединение к комнате .....</b>	<b>12</b>
<b>2.4. Запустить игру .....</b>	<b>15</b>
<b>2.5. Игра 0-го раунда.....</b>	<b>16</b>
<b>2.6. Игра 1-го или более раунда.....</b>	<b>19</b>
<b>Выводы.....</b>	<b>22</b>
<b>Технологическая часть.....</b>	<b>23</b>
<b>Выбор средств программной реализации.....</b>	<b>23</b>
Бэкенд.....	23
Фронтенд .....	23
<b>Разработка протокола:.....</b>	<b>23</b>
<b>Выводы.....</b>	<b>31</b>
<b>ЗАКЛЮЧЕНИЕ .....</b>	<b>32</b>
<b>СПИСОК ЛИТЕРАТУРЫ.....</b>	<b>33</b>
<b>Приложение А – Обработка сообщений WebSocket на сервере .....</b>	<b>34</b>
<b>Приложение Б - Обработка запросов на присоединение к игре на сервере .....</b>	<b>34</b>
<b>Приложение В - Обработка запроса на создание игры на сервере .....</b>	<b>35</b>
<b>Приложение Г - Обработка отключений игроков (сервер).....</b>	<b>36</b>
<b>Приложение Д - Обработка сообщения о начале игры (сервер) .....</b>	<b>38</b>
<b>Приложение Е - Обработка сообщения раунда 0 (сервер).....</b>	<b>39</b>
<b>Приложение Ё - Обработка сообщения раунда 1 (сервер).....</b>	<b>41</b>
<b>Приложение Ж - Обработка HTTP-запросов (клиент) .....</b>	<b>44</b>
<b>Приложение З - Обработка сообщений WebSocket (клиент).....</b>	<b>46</b>

## **ВВЕДЕНИЕ**

Современный рынок многопользовательских игр продолжает стремительно развиваться, предлагая пользователям все более сложные и увлекательные формы взаимодействия. Согласно отчету Statista [1], по состоянию на 2023 год в мире насчитывается около 1,1 миллиарда онлайн-игроков. Эта цифра подчеркивает значительное и растущее число людей, занимающихся онлайн-играми по всему миру.

Одним из интересных примеров многопользовательских игр является классическая игра "Морской бой", которая адаптирована для цифровой среды и обеспечивает возможность соревноваться с другими игроками в реальном времени. Учитывая требования к удобству и скорости взаимодействия между игроками, эффективный сетевой протокол играет ключевую роль в успешной реализации таких игр.

Цель работы – разработать протокол для пошаговой многопользовательской игры "Морской бой" на 2 и более игроков.

# **Аналитическая часть**

## **1.1 Типы многопользовательских игр**

Многопользовательские игры можно классифицировать на два основных типа в зависимости от характера взаимодействия между игроками:

### **1.1.1 Игры с пошаговым режимом**

Игры с пошаговым режимом (turn-based games) предполагают, что каждый игрок выполняет свои действия поочередно. После завершения хода одного игрока данные передаются другим участникам. Примеры таких игр: шахматы, пошаговые стратегии или карточные игры. Ключевые характеристики:

- **Низкие требования к времени отклика.** Небольшие задержки в передаче данных практически не влияют на игровой процесс.
- **Четкая последовательность действий.** Логика игры зависит от строгого соблюдения очередности ходов.

### **1.1.2 Игры в реальном времени**

Игры в реальном времени (real-time games) требуют синхронизации действий всех игроков в один момент. Примеры включают шутеры от первого лица, массовые многопользовательские онлайн-игры (MMORPG) и стратегии в реальном времени (RTS). Характерные особенности:

- **Высокие требования к задержке.** Любые задержки могут существенно повлиять на игровой процесс.
- **Одновременное выполнение действий.** Игроки взаимодействуют с игровым миром и друг с другом одновременно.

## **1.2 Протоколы для сетевого взаимодействия**

Современные сетевые системы используют различные протоколы передачи данных, работающие на разных уровнях модели TCP/IP. Выбор протокола зависит от требований к **надежности, безопасности, задержке и эффективности передачи данных**.

В данном разделе рассматриваются два уровня:

- **Транспортный уровень:** TCP и UDP.
- **Прикладной уровень:** WebSocket и HTTP.

### **1.2.1 Протоколы транспортного уровня**

#### **TCP (Transmission Control Protocol)**

TCP — это **надежный транспортный протокол**, гарантирующий доставку данных в **правильном порядке**. Он широко используется в приложениях, где важно избежать потерь информации.[2]

### **Основные характеристики TCP:**

- **Гарантированная доставка данных** – использует механизм подтверждения (ACK).
- **Контроль потока и перегрузки** – регулирует передачу в зависимости от состояния сети.
- **Более высокая нагрузка на сеть**, чем у UDP, из-за дополнительной обработки пакетов.

TCP идеально подходит для систем, где **важна надежность передачи данных**.

### **UDP (User Datagram Protocol)**

UDP — это **легковесный и быстрый** транспортный протокол, который **не гарантирует** доставку пакетов, но снижает задержку. Он применяется в системах, где важна **скорость обмена данными**.[3]

### **Основные характеристики UDP:**

- **Отсутствие подтверждений** – пакеты отправляются без ожидания ответа.
- **Меньшая нагрузка на сеть** – заголовки пакетов меньше, чем у TCP.
- **Нет механизма повторной передачи** – потерянные пакеты не восстанавливаются.

UDP идеально подходит для приложений, где **важна минимальная задержка** и допустима небольшая потеря данных.

## Выбор транспортного протокола

Для разрабатываемого сетевого взаимодействия был выбран **TCP**, так как игра основана на пошаговом геймплее, где **важна надежная передача данных и строгий порядок их получения**

## 1.2.2 Протоколы прикладного уровня

### **WebSocket (WS)**

WebSocket (WS) — это протокол связи, обеспечивающий двустороннюю связь между клиентом и сервером через одно TCP-соединение. [4]

### **Основные характеристики WS:**

- **Постоянное соединение** – устанавливается один раз и поддерживается на протяжении всей сессии.
- **Двунаправленный обмен** – клиент и сервер могут отправлять данные друг другу в любое время без предварительных запросов.
- **Использует TCP** – WS работает поверх TCP, что гарантирует надежную доставку пакетов.

### **HTTP (HyperText Transfer Protocol)**

HTTP это протокол для передачи данных по схеме «запрос-ответ» [5]

### **Основные характеристики HTTP:**

- **Протокол с запросами-ответами** – клиент отправляет запрос на сервер и ожидает ответа.
- **Использует TCP** – обеспечивает надежную доставку данных.

### Различия между версиями HTTP:

Версия	Описание	Основные отличия
HTTP 1.1	Использует отдельное соединение для каждого запроса.	Медленнее из-за необходимости устанавливать новые TCP-соединения.
HTTP 2	Использует одноканальное соединение (multiplexing).	Позволяет загружать несколько ресурсов одновременно без ожидания.
HTTP 3	Основан на протоколе QUIC, а не TCP.	Уменьшает задержку, так как не требует установки TCP-соединения.

Таблица 1 Различия между версиями HTTP

### Выводы

Многопользовательские игры подразделяются на игры с пошаговым режимом и игры в реальном времени, каждая из которых предъявляет разные требования к сетевому взаимодействию. Игры с пошаговым режимом имеют низкие требования к задержкам и требуют четкой последовательности передачи данных.

Был выбран TCP, так как игра пошаговая и требует надежной доставки данных. Протокол WebSocket (WS) был выбран для двусторонней передачи данных. Для обеспечения масштабируемости было принято решение, что клиент должен делать HTTP-запрос для получения ссылки на WebSocket-сервер, что позволит распределять игроков по разным серверам для улучшения производительности. Для этого запроса был выбран HTTP 1.1, так как он выполняется только один раз и использование HTTP 2 или 3 потребовало бы дополнительной сложности в разработке, при этом не принесло бы значительного улучшения производительности.

## Конструкторская часть

### 2.1. Основные этапы протокола для пошаговой многопользовательской игры "Морской бой"

В разработке протокола для онлайн-игры "Морской бой" было принято решение использовать HTTP для первого запроса, который выполняет функцию создания игровой комнаты или присоединения к существующей. В ответе на этот запрос сервер возвращает персональный URL-адрес, который клиент использует для установления соединения по WebSocket. После установления WebSocket-соединения дальнейшая коммуникация между клиентом и сервером осуществляется через этот канал до завершения игры. Фреймы сообщений WebSocket имеют структуру, описанную в таблице 2. Кроме того, структура JSON, отправляемого в виде WebSocket payload от клиента к серверу, описана в таблице 3, а структура JSON, отправляемого от сервера к клиенту, представлена в таблице 4. Знак "\*" в таблицах обозначает атрибуты, специфичные для типа операции. В таблице 5 показаны все возможные значения для атрибута «operation» и их описание.

Поле заголовка	Значение
FIN	1
Opcode	1
Mask	1
Payload Length	...
Masking Key	...
Payload	Текст, содержащий JSON

Таблица 2 Структура фреймов сообщений WebSocket

Атрибут	Значение
operation	...
user_id	...
room_id	...
*	...

Таблица 3 Структура JSON от клиента к серверу

Атрибут	Значение
operation	...
*	...

Таблица 4 Структура JSON от сервера к клиенту

operation	описание
0	Используется сервером для оповещения всех игроков в комнате о том, что к их комнате подключился новый пользователь.
1	Используется клиентом для инициализации игры и сервером для информирования всех игроков о начале игры.

2	Используется клиентом для игры в раунд 0 и сервером для информирования всех игроков о том, сколько пользователей завершили этот раунд.
3	Используется сервером для указания того, что все игроки завершили раунд 0 и что раунд 1 начался.
4	Используется игроком для выполнения своего хода в раунде и сервером для информирования всех игроков о результате хода игрока.
5	Используется сервером для информирования всех игроков о том, что игра закончилась.
6	Используется сервером для информирования всех игроков о том, что игрок покинул игру.

*Таблица 5 Возможные значения для «operation» и их описание*

Основные этапы:

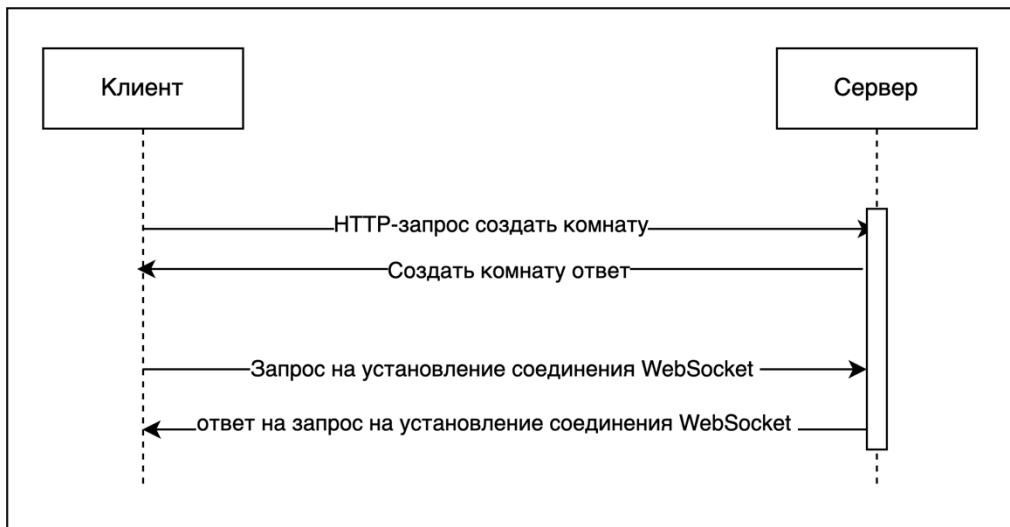
1. Игрок отправляет запрос на сервер для создания игровой комнаты, сервер отвечает с помощью созданного идентификатора комнаты.
2. Другие игроки присоединяются к комнате через запрос на сервер. Запрос содержит идентификатор комнаты, который игрок получил вне протокола (например, один игрок создал игровую комнату и отправил идентификатор комнаты своему другу через какое-то чат-приложение).
3. Игрок, создавший комнату, отправляет запрос на начало игры, затем сервер отправляет всем игрокам в комнате сообщения. Сообщения указывает на начало игры и содержит список игроков и порядок игры.
4. Игрок отправляет свой ход на сервер, а сервер отправляет ход и его результат всем игрокам. Этот процесс повторяется для всех игроков до конца игры.

Далее каждый шаг будет описан более подробно.

## 2.2. Создание комнаты

Клиент отправляет HTTP-запрос на сервер для создания комнаты. Клиент отправляет в качестве параметра максимальное количество игроков. Сервер создает комнату с требуемыми параметрами, генерирует идентификатор пользователя для клиента (который представляет его в этой комнате) и добавляет его в список игроков в созданной комнате, а также генерирует URL для установления соединения websocket с этим клиентом. Клиент получает в качестве ответа свой идентификатор пользователя, идентификатор комнаты и URL для установления соединения websocket для дальнейшего общения. Далее представлена

схема создания комнаты, таблицы атрибутов, таблица с возможными ошибками и структура запросов и ответов, задействованных на этом этапе:



*Rис. 1 Схема создание комнаты*

Параметр	Тип	Описание
number_of_players	Int	Максимально допустимое количество игроков в игре

*Таблица 6 Атрибутивный состав запроса создание комнаты*

Параметр	Тип	Описание
user_id	String	Уникальный идентификатор пользователя
room_id	String	Уникальный идентификатор комнаты
websocket_url	String	URL для установки соединения через веб-сокет

*Таблица 7 Атрибутивный состав ответа создания комнаты*

Параметр	Тип	Описание
message	String	Описание ошибки
error_code	Int	Идентификатор ошибки

*Таблица 8 Атрибутивный состав ответа на ошибку создания комнаты*

Error Code	Message	Описание
1001	The number of players need to be 2 or more	Возвращается, когда параметр "number_of_players" меньше 2
1002	Missing parameter number_of_players	Возвращается, когда отправленный запрос не имеет параметра "number_of_players"
1000	Unknown error	Возвращается, когда произошла другая ошибка

*Таблица 9 Возможные ошибки*

**HTTP-запрос (от клиента к серверу) создания комнаты:**

```
POST /create-room HTTP/1.1
Host: ...
Accept: application/json
Content-Length: ...
{
  "number_of_players": ...
}
```

**HTTP-ответ (от сервера к клиенту) создания комнаты:**

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: ...
{
  "user_id": ...,
  "room_id": ...,
  "websocket_url": ...
}
```

**HTTP-ответ (от сервера к клиенту) создания комнаты при ошибке:**

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: ...
{
  "message": ...,
  "error_code": ...
}
```

**Запрос (от клиента к серверу) на установление соединения WebSocket (рукопожатие):**

```
GET WebSocket_URL(после "/") HTTP/1.1
Host: WebSocket_URL(до "/")
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: ...
Sec-WebSocket-Version: 13
```

**Ответ (от сервера к клиенту) на запрос на установление соединения WebSocket (рукопожатие):**

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: ...
```

## 2.3. Присоединение к комнате

Клиент отправляет HTTP-запрос на сервер для присоединения к комнате. Клиент отправляет в качестве параметра идентификатор комнаты, к которой хочет присоединиться. Сервер генерирует идентификатор пользователя, который будет представлять этого клиента в комнате и добавлять его в список игроков соответствующей комнаты, также сервер генерирует URL для соединения WebSocket с клиентом. Клиент получает в качестве ответа свой идентификатор пользователя, идентификатор присоединенной комнаты, URL для соединения WebSocket, максимальное количество разрешенных игроков в комнате и текущее количество игроков в комнате. Клиенты, которые уже находятся в комнате, также получают количество текущих игроков в комнате в виде сообщения через свои соединения WebSocket. Далее представлена схема присоединения к комнате, таблицы атрибутов, таблица с возможными ошибками и структура запросов и ответов, задействованных на этом этапе:

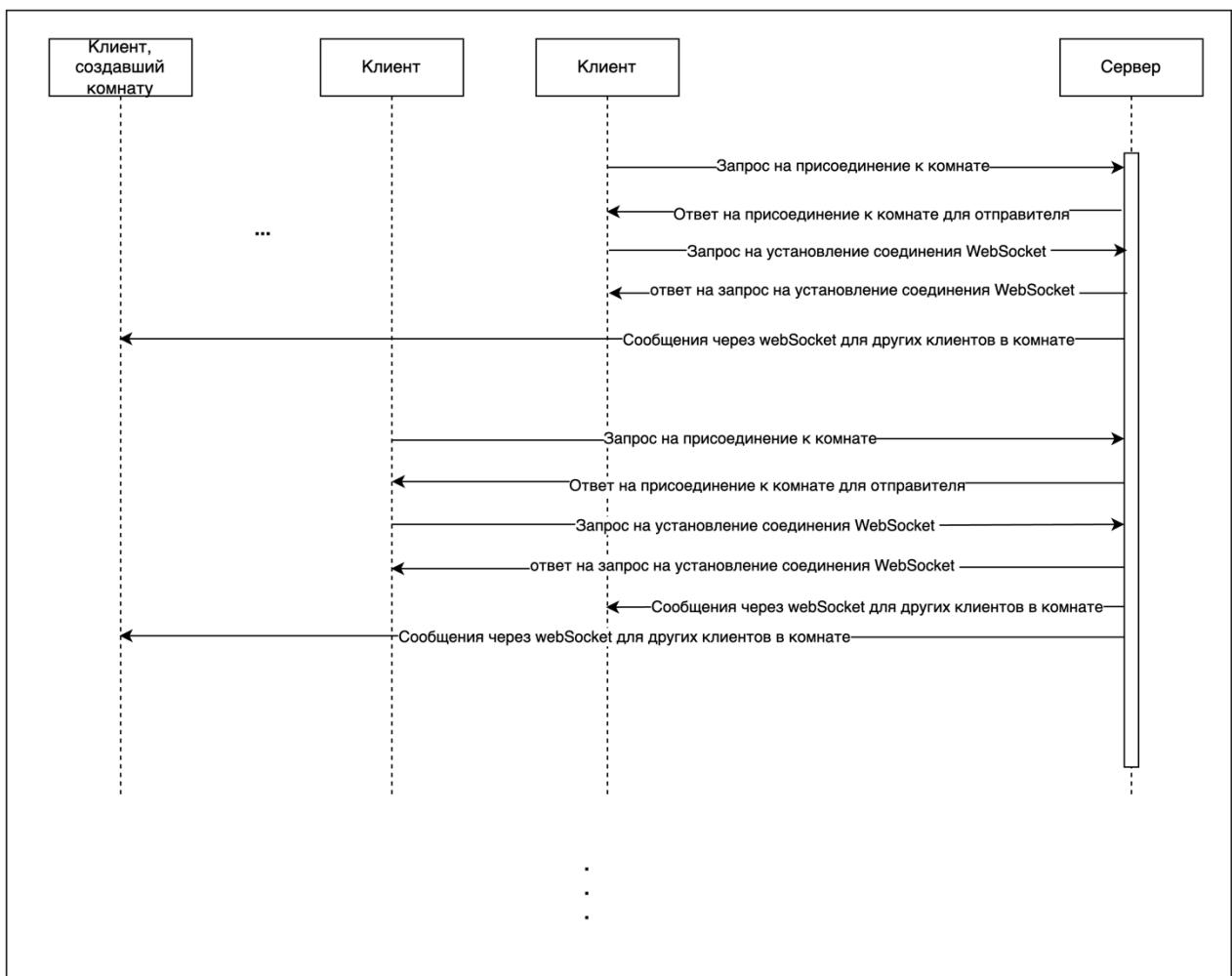


Рис. 2 Схема присоединения к комнате

Параметр	Тип	Описание
room_id	String	Уникальный идентификатор комнаты

Таблица 10 Атрибутивный состав запроса присоединение к комнате

Параметр	Тип	Описание
user_id	String	Уникальный идентификатор пользователя

room_id	String	Уникальный идентификатор комнаты
websocket_url	String	URL для установки соединения через веб-сокет
number_of_players	Int	Максимально допустимое количество игроков в игре
number_of_current_players	Int	Количество игроков, присоединившихся к комнате

Таблица 11 Атрибутивный состав ответа отправителю при подключении к комнате

Параметр	Тип	Описание
message	String	Описание ошибки
error_code	Int	Идентификатор ошибки

Таблица 12 Атрибутивный состав ответа отправителю на ошибку подключения к комнате

Параметр	Тип	Описание
operation	Int	Тип операции (в данном случае присоединение к комнате)
number_of_current_players	Int	Количество игроков, присоединившихся к комнате

Таблица 13 Атрибутивный состав ответа другим игрокам при подключении к комнате

Error Code	Message	Описание
2001	Invalid room ID	Возвращается, когда комната со вставленным параметром room_id не существует.
2002	Missing parameter room_id	Возвращается, когда отправленный запрос не имеет параметра "room_id"
2003	The room is already full	Возвращается, когда комната уже заполнена
2004	The game in the room are already started	Возвращается, когда игра в комнате уже началась
2000	Unknown error	Возвращается, когда произошла другая ошибка

Таблица 14 Возможные ошибки

**HTTP-запрос присоединения к комнате:**

```
POST /join-room HTTP/1.1
Host: ...
Accept: application/json
Content-Length: ...
{
  "room_id": ...
}
```

**HTTP-ответ присоединения к комнате:**

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: ...
{
  "user_id": ...,
  "room_id": ...,
  "websocket_url": ...,
  "number_of_players": ...,
  "number_of_current_players": ...
}
```

**HTTP-ответ присоединения к комнате при ошибке:**

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: ...
{
  "message": ...,
  "error_code": ...
}
```

**Запрос на установление соединения WebSocket Secure (рукопожатие):**

```
GET WebSocket_URL(после "/") HTTP/1.1
Host: WebSocket_URL(до "/")
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: ...
Sec-WebSocket-Version: 13
```

**Ответ на запрос на установление соединения WebSocket Secure (рукопожатие):**

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: ...
```

**Payload сообщения WebSocket с сервера всем остальным клиентам, уже подключенными к комнате (при изменении количества игроков, подключенных к комнате):**

```
{
  "operation": 0,
  "number_of_current_players": ...
}
```

## 2.4. Запустить игру

Клиент, создавший комнату, отправляет сообщение через веб-сокет на сервер, чтобы указать начало игры. Клиент помещает в сообщение идентификатор игровой комнаты, идентификатор своего пользователя и тип операции (начать игру). Сервер устанавливает атрибут раунда комнаты на 0 и назначает имя цвета каждому пользователю в комнате. Каждый клиент, подключенный к комнате, получает сообщение через веб-сокет с номером раунда и соответствующим цветом, который представляет клиента. Далее представлена схема запустить игру, таблицы атрибутов, таблица с возможными ошибками и структура сообщений, задействованных на этом этапе:

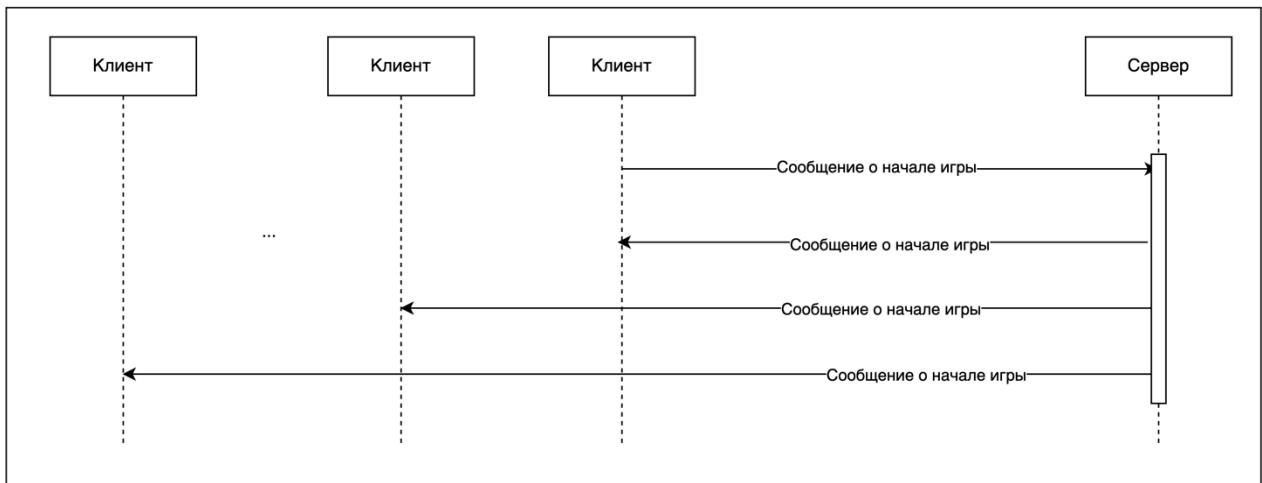


Рис. 3 Схема запустить игру

Параметр	Тип	Описание
<u>operation</u>	Int	Тип действия, которое пользователь хочет выполнить (в данном случае начать игру)
<u>user_id</u>	String	Уникальный идентификатор пользователя
<u>room_id</u>	String	Уникальный идентификатор комнаты

Таблица 15 Атрибутивный состав запроса запустить игру

Параметр	Тип	Описание
<u>message</u>	String	Описание ошибки
<u>error_code</u>	Int	Идентификатор ошибки

Таблица 16 Атрибутивный состав ответа отправителю на ошибку запустить игру

Параметр	Тип	Описание
<u>operation</u>	Int	Тип операции (в данном случае начать игру)
<u>round</u>	Int	Текущий раунд игры
<u>user_color</u>	String	Цвет пользователя в игре

Таблица 17 Атрибутивный состав ответа всем игрокам при запуске игры.

Error Code	Message	Описание
3001	The number of players in the room need to be at least 2	Возвращается, когда в комнате меньше 2 игроков
3002	Invalid room _id	Возвращается, когда идентификатор комнаты не существует.
3003	The game can only be started by the creator of the room	Возвращается, когда идентификатор пользователя не принадлежит создателю комнаты
3004	The game in the room are already started	Возвращается, когда игра в комнате уже началась
3000	Unknown error	Возвращается, когда произошла другая ошибка

Таблица 18 Возможные ошибки

**Payload сообщения WebSocket от клиента, создавшего комнату, к серверу:**

```
{
  "operation": 1,
  "user_id": ...,
  "room_id": ...
}
```

**Payload сообщения WebSocket с сервера клиентам подключенным к комнате:**

```
{
  "operation": 1,
  "round": ...,
  "user_color": ...
}
```

**Payload сообщения WebSocket с сервера клиенту, который инициирует процесс запуска игры (в случае ошибки):**

```
{
  "message": ...,
  "error_code": ...
}
```

## 2.5. Игра 0-го раунда

Каждый клиент, подключенный к комнате, отправляет сообщение через соединение webSocket. Клиент помещает в сообщение тип операции (игра раунда 0), идентификатор своего пользователя, идентификатор комнаты и матрицу целых чисел, представляющую его стол с кораблями. Сервер, получив сообщение такого типа, помещает матрицу пользователя в словарь, который представляет столы пользователей в комнате, также сервер подсчитывает, сколько пользователей завершили раунд 0, и если все завершили его, сервер присваивает атрибуту раунда комнаты значение 1. Каждый раз, когда клиент играет свой ход раунда 0, все клиенты в этой игре получают сообщение через webSocket, это сообщение содержит количество игроков, которые завершили раунд 0, и если все-таки его раунд 0 завершается, все игроки получают еще одно сообщение, которое содержит новый номер

текущего раунда, порядок действий и цвет игрока, которому принадлежит текущий ход. Матрица, представляющая доску игрока с кораблями, должна иметь следующий формат:

- Иметь размер 10x10
- Пустые ячейки должны быть представлены значением «0»
- Ячейки, представляющие корабль, должны быть представлены значением «2» (если размер корабля составляет 2 ячейки), «3» (если размер корабля составляет 3 ячейки), «4» (если размер корабля составляет 4 ячейки) или «5» (если размер корабля составляет 5 ячеек)
- Корабли на доске могут иметь только горизонтальную или вертикальную ориентацию (не могут иметь диагональную ориентацию)
- Доска должна содержать 1 корабль размером 2 клетки, 2 корабля размером 3 клетки, 1 корабль размером 4 клетки и 1 корабль размером 5 клеток.

Далее представлена схема игры 0-го раунда, таблицы атрибутов, таблица с возможными ошибками и структура сообщений, задействованных на этом этапе:

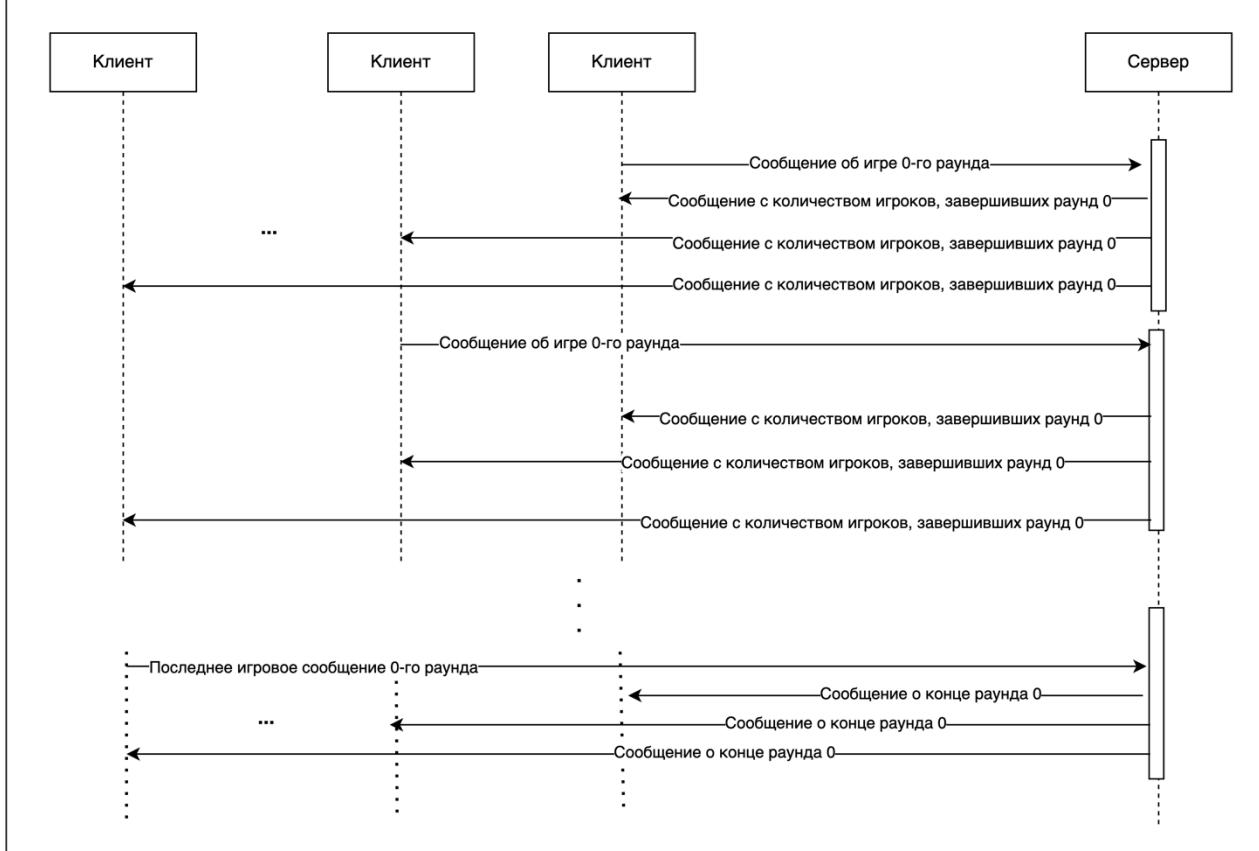


Рис. 4 Схема игры 0-го раунда

Параметр	Тип	Описание
operation	Int	Тип действия, которое пользователь хочет выполнить (в данном случае раунд 0)
user_id	String	Уникальный идентификатор пользователя
room_id	String	Уникальный идентификатор комнаты
player_table	[[Int]]	Матрица с позициями кораблей игроков

Таблица 19 Атрибутивный состав запроса игры 0-го раунда

Параметр	Тип	Описание
operation	Int	Тип операции
completed_round0_players	[String]	Игроки, завершившие раунд 0

Таблица 20 Атрибутивный состав ответа всем игрокам при игре 0-го раунда.

Параметр	Тип	Описание
operation	Int	Тип операции
players_order	[{String, Int}]	Массив с порядком игры каждого цвета
round	Int	Текущий раунд игры
current_player_turn	String	Цвет пользователя, которому нужно сделать ход

Таблица 21 Атрибутивный состав ответа всем игрокам при последней игре 0-го раунда.

Параметр	Тип	Описание
message	String	Описание ошибки
error_code	Int	Идентификатор ошибки

Таблица 22 Атрибутивный состав ответа отправителю на ошибку при игре 0-го раунда

Error Code	Message	Описание
4001	Invalid table format	Возвращается, когда отправленная таблица не имеет требуемого формата.
4002	Invalid room_id	Возвращается, когда идентификатор комнаты не существует
4003	Missing parameters	Возвращается, если отправленное сообщение не содержит 1 или более обязательных параметров.
4004	The round 0 has already ended or not started yet	Возвращается, когда игра не находится в раунде 0
4005	Invalid user_id	Возвращается, когда идентификатор пользователя не принадлежит комнате
4006	The user has already played his round 0	Возвращается, когда пользователь пытается отправить свою игру раунда 0 во второй раз.
4000	Unknown error	Возвращается, когда произошла другая ошибка

Таблица 23 Возможные ошибки

**Payload сообщения WebSocket от клиента к серверу:**

```
{
  "operation": 2,
  "user_id": ...,
  "room_id": ...,
  "player_table": ...
}
```

**Payload сообщения WebSocket с сервера клиентам подключенным к комнате:**

```
{
  "operation": 2,
  "round": ...,
  "completed_round0_players": ...
}
```

**Payload сообщения WebSocket с сервера клиентам подключенным к комнате когда все игроки завершили раунд 0:**

```
{
  "operation": 3,
  "players_order": ...,
```

```
    "round": ...,
    "current_player_turn": ...
}
```

**Payload сообщения WebSocket с сервера клиенту, который отправил сообщение, генерирующее ошибку (в случае ошибки):**

```
{
  "message": ...,
  "error_code": ...
}
```

## 2.6. Игра 1-го или более раунда

Клиент, которому принадлежит текущий ход, отправляет сообщение на сервер через соединение websocket. Сообщение содержит тип операции (раунд 1+ игра), идентификатор пользователя клиента, идентификатор комнаты, цвет атакованного игрока и координаты атаки. Сервер проверяет, действительна ли атака, и в случае ее наличия изменяет таблицу атакованного пользователя и проверяет, жив ли пользователь. В случае смерти пользователя сервер удаляет его из списка порядка хода. После этого, если в этом списке только один пользователь, этот пользователь является победителем, если нет, сервер проверяет, закончился ли раунд, и обновляет номер раунда. Каждый клиент получает сообщение от сервера через соединение websocket. Если игра еще не закончилась, сообщение содержит цвет атакующего, цвет атакованного пользователя, координаты атаки, результат атаки (попадание в корабль или нет), текущий раунд, цвет текущего игрока для игры и обновленный список порядка ходов. Если игра закончилась, сообщение содержит цвет победителя. Также, если какой-либо клиент не отправляет сообщение в течение 60 секунд, сервер удаляет его из списка порядка хода и отправляет сообщение всем клиентам, чтобы сообщить, что этот клиент больше не играет. Далее представлена схема игры 1-го или более раунда, таблицы атрибутов, таблица с возможными ошибками и структура сообщений, задействованных на этом этапе:

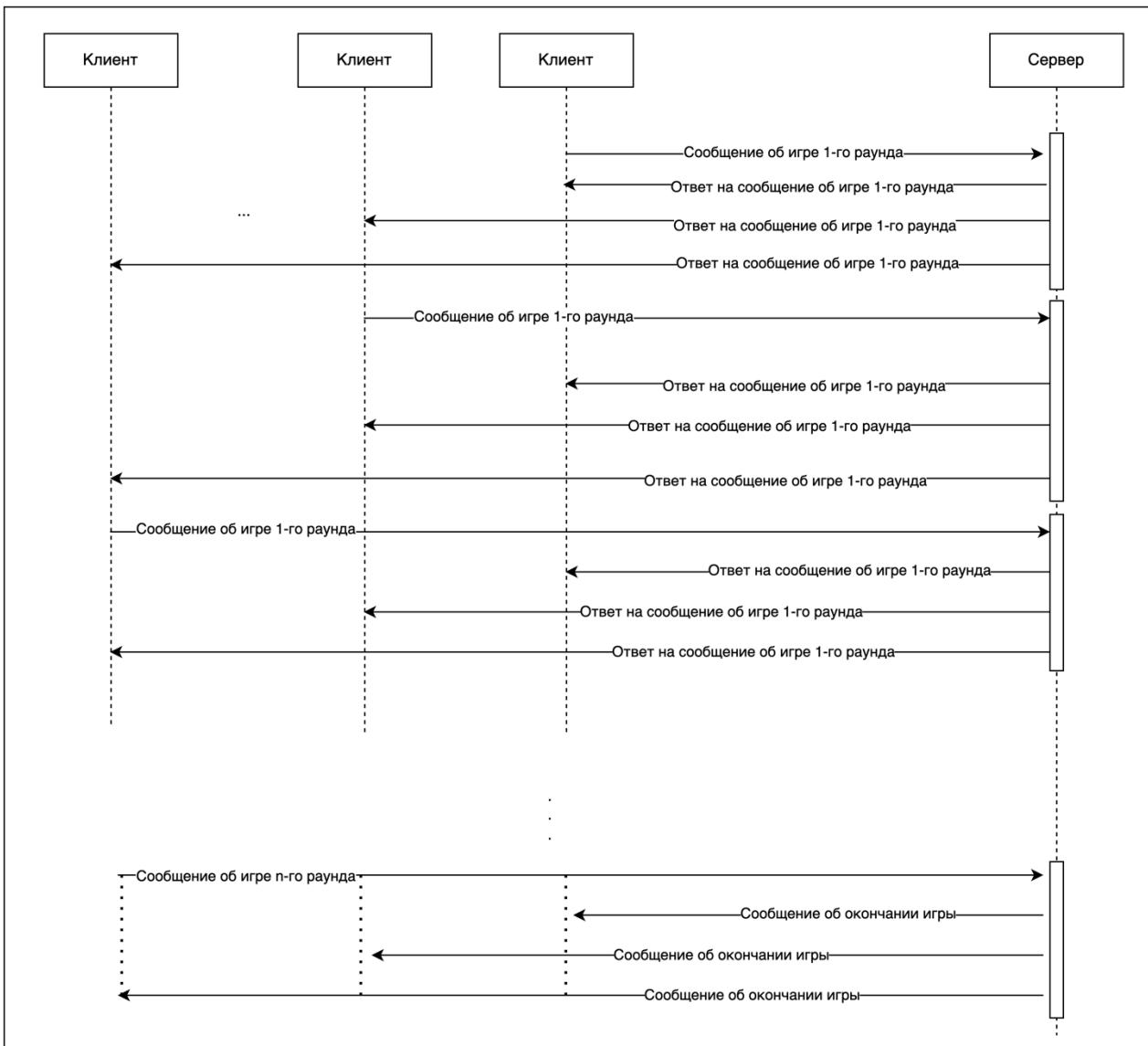


Рис. 5 Схема игры 1-го или более раунда

Параметр	Тип	Описание
operation	Int	Тип действия, которое пользователь хочет выполнить (в данном случае раунд 1+ игра)
user_id	String	Уникальный идентификатор пользователя
room_id	String	Уникальный идентификатор комнаты
player_color	String	Цвет, который игрок хочет атаковать.
coordinates	{Int, Int}	Координаты атаки

Таблица 24 Атрибутивный состав запроса игра 1-го или более раунда

Параметр	Тип	Описание
operation	Int	Тип операции
attack_emisor_color	String	Цвет игрока, который атакует
attack_receptor_color	String	Цвет игрока, который принимает атаку.
coordinates	{Int, Int}	Координаты атаки
result	Int	Указывает, поразила ли атака какой-либо корабль (result = -2) или нет (result = -1)

round	Int	Текущий раунд игры
current_player_turn	String	Цвет пользователя, которому нужно сделать ход
players_order	[{String, Int}]	Массив с порядком игры каждого цвета (без мертвых)

Таблица 25 Атрибутивный состав ответа всем игрокам при игре 1-го или более раунда

Параметр	Тип	Описание
operation	Int	Тип операции
round	Int	Текущий раунд игры
current_player_turn	String	Цвет пользователя, которому нужно сделать ход
players_order	[{String, Int}]	Массив с порядком игры каждого цвета (без мертвых)

Таблица 26 Атрибутивный состав ответа, когда игрок не сделал хода в течение 60 секунд.

Параметр	Тип	Описание
operation	Int	Тип операции
winner_color	String	Цвет победителя

Таблица 27 Атрибутивный состав ответа при завершении игры

Параметр	Тип	Описание
message	String	Описание ошибки
error_code	Int	Идентификатор ошибки

Таблица 28 Атрибутивный состав ответа отправителю на ошибку при игре 1-го или более раунда

Error Code	Message	Описание
5001	Invalid coordinates	Возвращается, когда координаты находятся за пределами таблицы или имеют отрицательные числа.
5002	Invalid room_id	Возвращается, когда идентификатор комнаты не существует
5003	Missing parameters	Возвращается, если отправленное сообщение не содержит 1 или более обязательных параметров.
5004	The game current round is 0 or below	Возвращается, когда текущий раунд игры равен 0 или ниже
5005	The attacked player is not valid	Возвращается, когда цвет атакуемого игрока не существует или если он уже умер или покинул игру или если игрок пытается атаковать сам себя
5006	Some player already attacked that cell of the player	Возвращается, когда ячейка атакованного игрока уже была атакована каким-либо игроком
5007	Is the turn of another player	Возвращается, когда текущий ход не принадлежит игроку, который пытается играть.
5008	Invalid user_id	Возвращается, когда идентификатор пользователя не принадлежит комнате
5000	Unknown error	Возвращается, когда произошла другая ошибка

Таблица 29 Возможные ошибки

**Payload сообщения WebSocket от клиента к серверу:**

```
{
  "operation": 4,
  "user_id": ...,
```

```
    "room_id": ...,
    "player_color": ...,
    "coordinates": {..., ...}
}
```

**Payload сообщения WebSocket с сервера клиентам подключенным к комнате (Когда кто-то играет свою очередь):**

```
{
    "operation": 4,
    "attack_emisor_color": ...,
    "attack_receptor_color": ...,
    "coordinates": {..., ...},
    "result": ...,
    "round": ...,
    "current_player_turn": ...,
    "players_order": ...
}
```

**Payload сообщения WebSocket с сервера клиентам подключенным к комнате (Когда кто-то выходит из игры):**

```
{
    "operation": 6,
    "round": ...,
    "current_player_turn": ...,
    "players_order": ...
}
```

**Payload сообщения WebSocket с сервера клиентам подключенным к комнате (Когда игра заканчивается):**

```
{
    "operation": 5,
    "winner": ...
}
```

**Payload сообщения WebSocket с сервера клиенту, который отправил сообщение, генерирующее ошибку (в случае ошибки):**

```
{
    "message": ...,
    "error_code": ...
}
```

## Выводы

В этом разделе были формализованы сообщения с помощью схем и атрибутивного состава.

# Технологическая часть

## Выбор средств программной реализации

### Бэкенд

Бэкенд был разработан с использованием nodeJS и модулей «express» и «ws».

### Фронтенд

Фронтенд был разработан для выполнения в macOS и был разработан с использованием языка программирования Swift и фреймворка SwiftUI. Также для сетевого подключения использовались модули «NSURLSession» и «NSURLSessionWebSocketTask»

### Разработка протокола:

Основные функции реализации серверной части представлены в приложении А - Ё, а основные функции реализации клиентской части представлены в приложении Ж - З. Далее представлен пример протокола в работе с 3 игроками:

- Игрок 1 создает комнату:

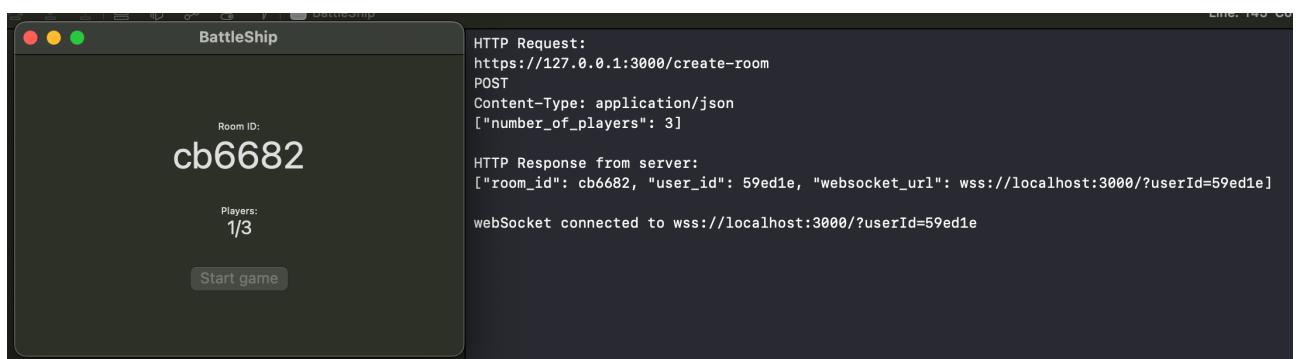


Рис. 6 Игрок 1 создает комнату

- Игрок 2 присоединяется к созданной комнате:

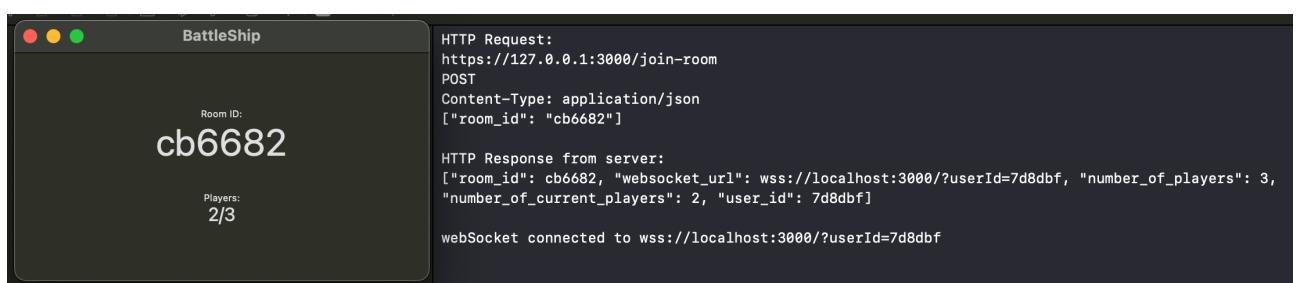


Рис. 7 Игрок 2 присоединяется к созданной комнате

- Игрок 1 получает сообщение с количеством подключенных игроков:

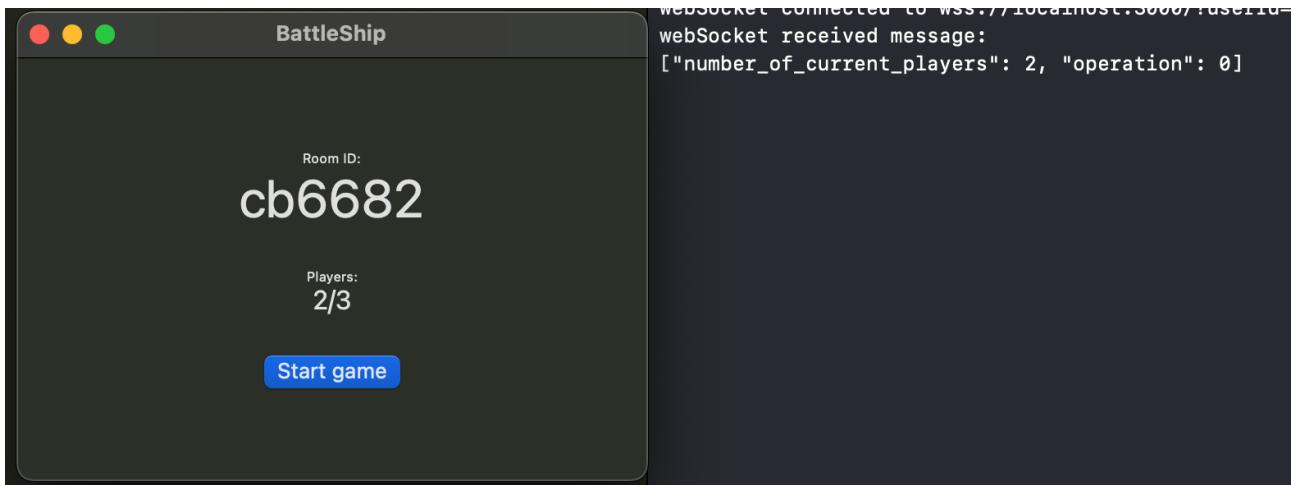


Рис. 8 Игрок 1 получает сообщение с количеством подключенных игроков

- Игрок 3 присоединяется к созданной комнате:

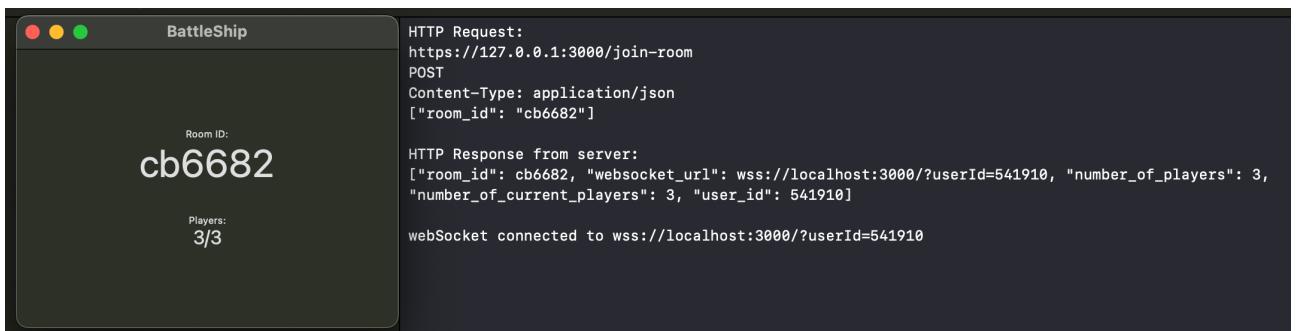


Рис. 9 Игрок 3 присоединяется к созданной комнате

- Игроки 1 и 2 получают сообщение с количеством подключенных игроков:

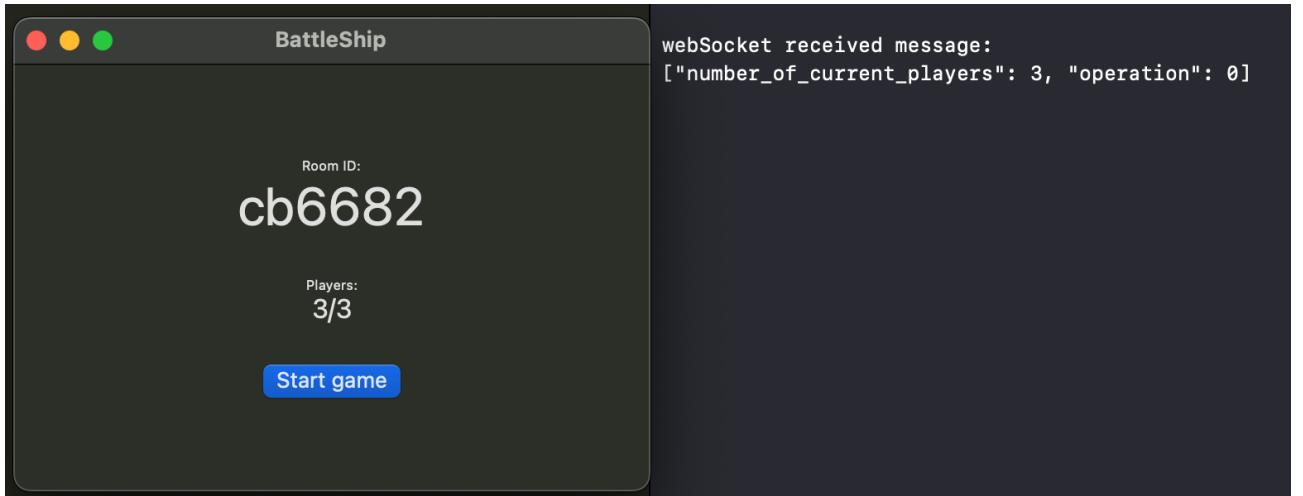


Рис. 10 Игрошки 1 и 2 получают сообщение с количеством подключенных игроков

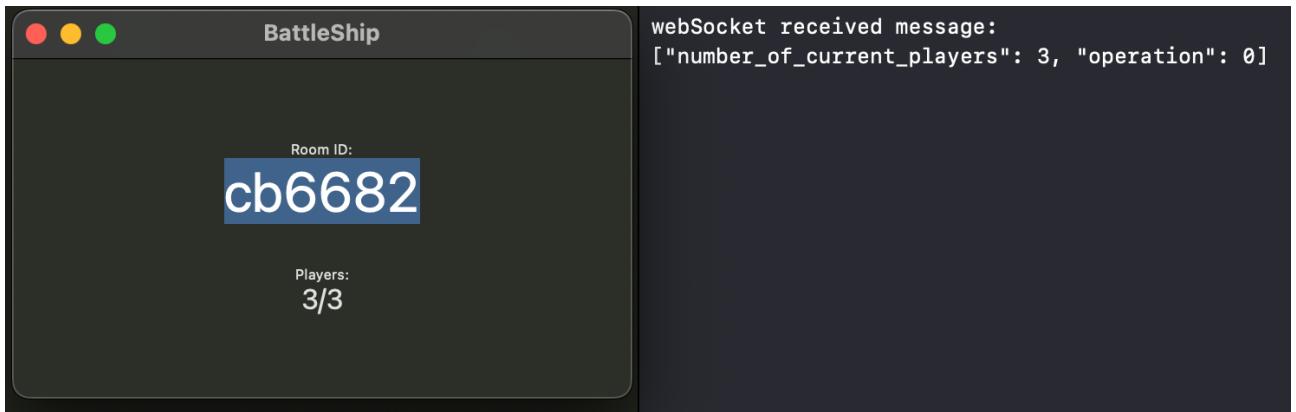


Рис. 11 Игроки 1 и 2 получают сообщение с количеством подключенных игроков

- Игрок 1 начинает игру:

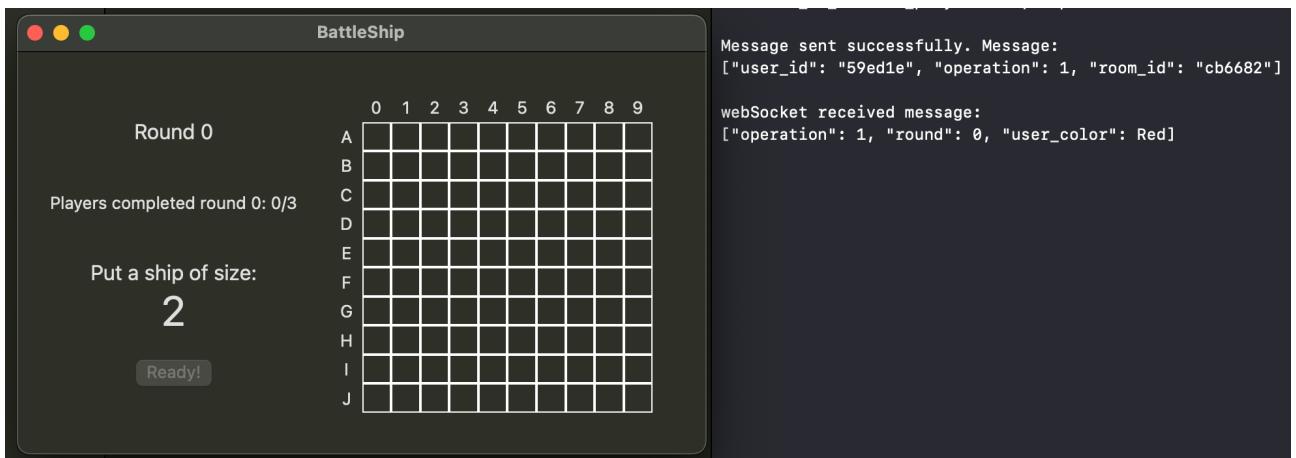


Рис. 12 Игровой 1 начинает игру

- Игроки 2 и 3 получают сообщение о начале игры:

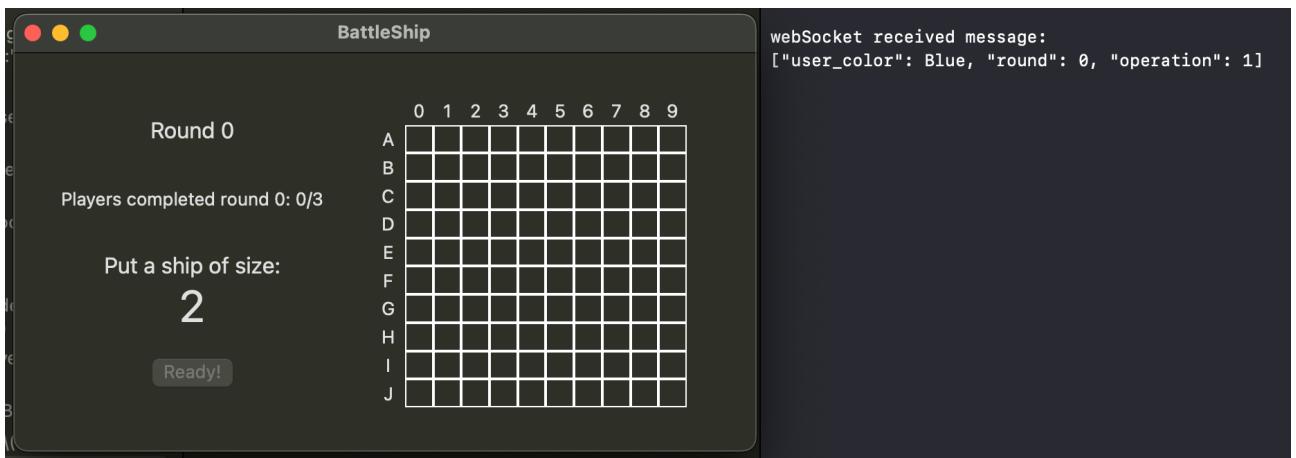


Рис. 13 Игровые 2 и 3 получают сообщение о начале игры

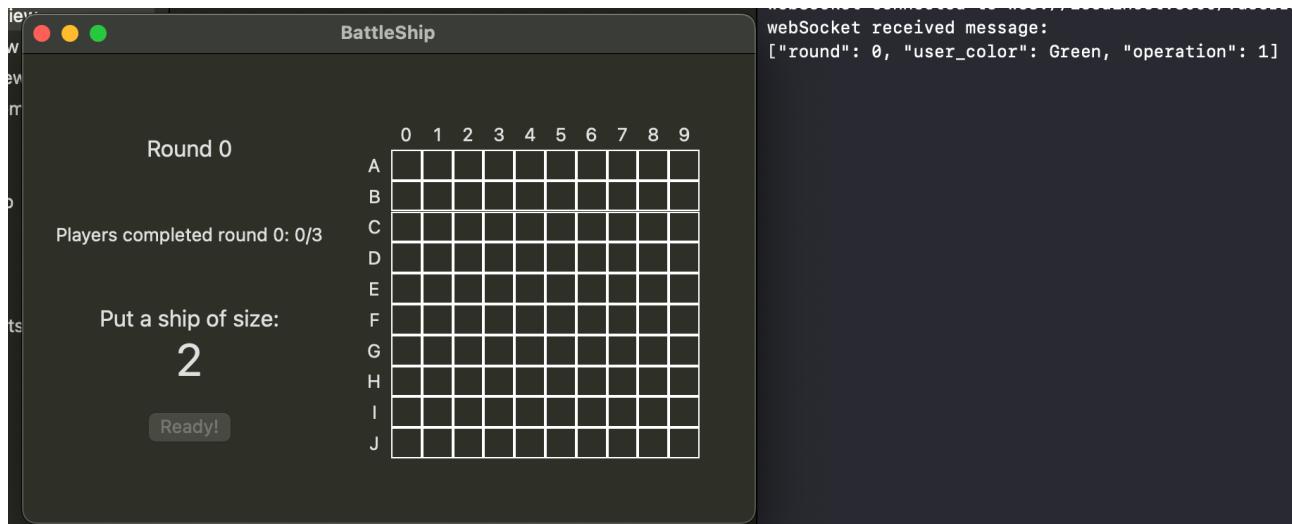


Рис. 14 Игроки 2 и 3 получают сообщение о начале игры

- Игроки играют раунд 0:

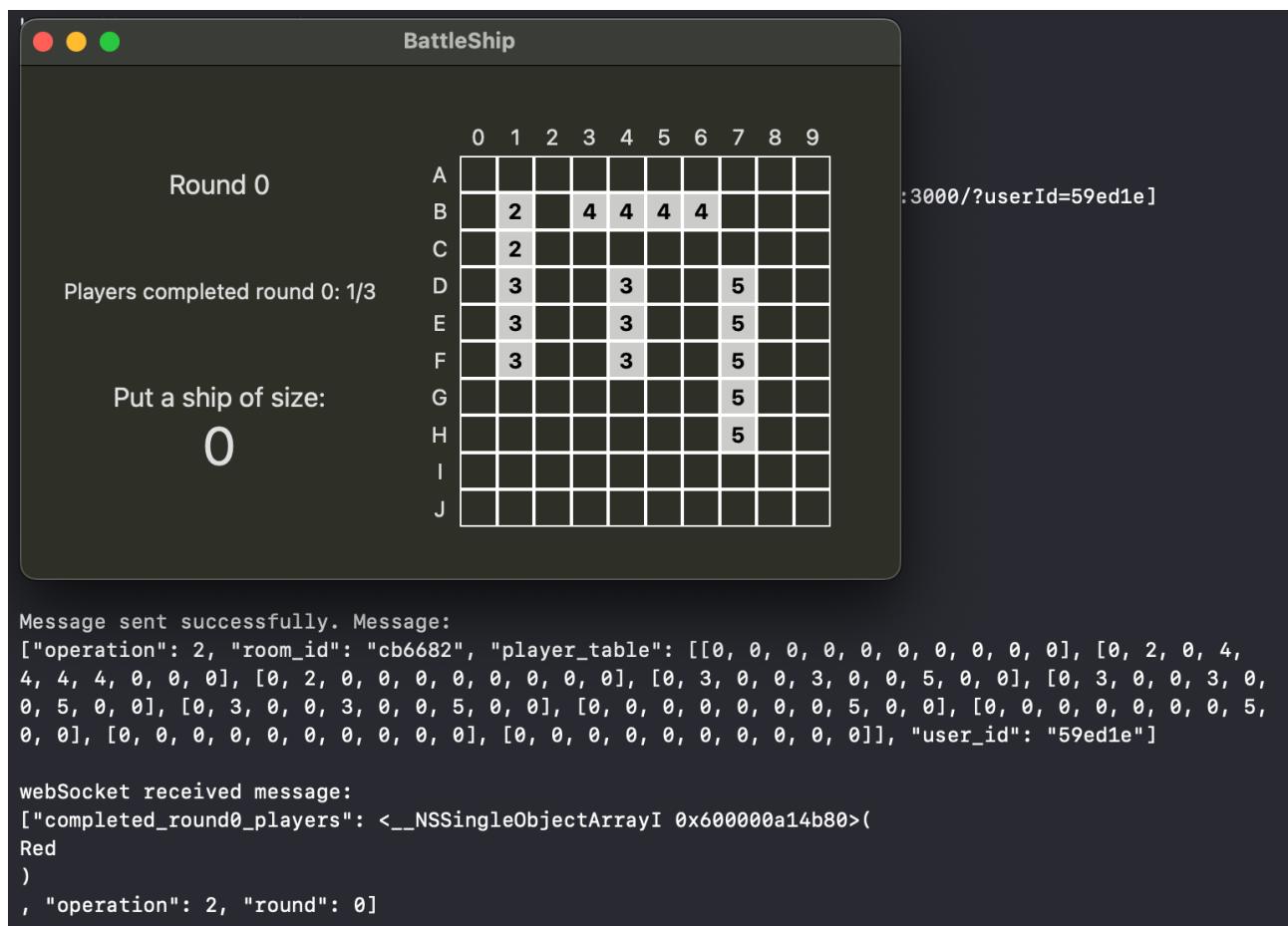


Рис. 15 Игроки играют раунд 0

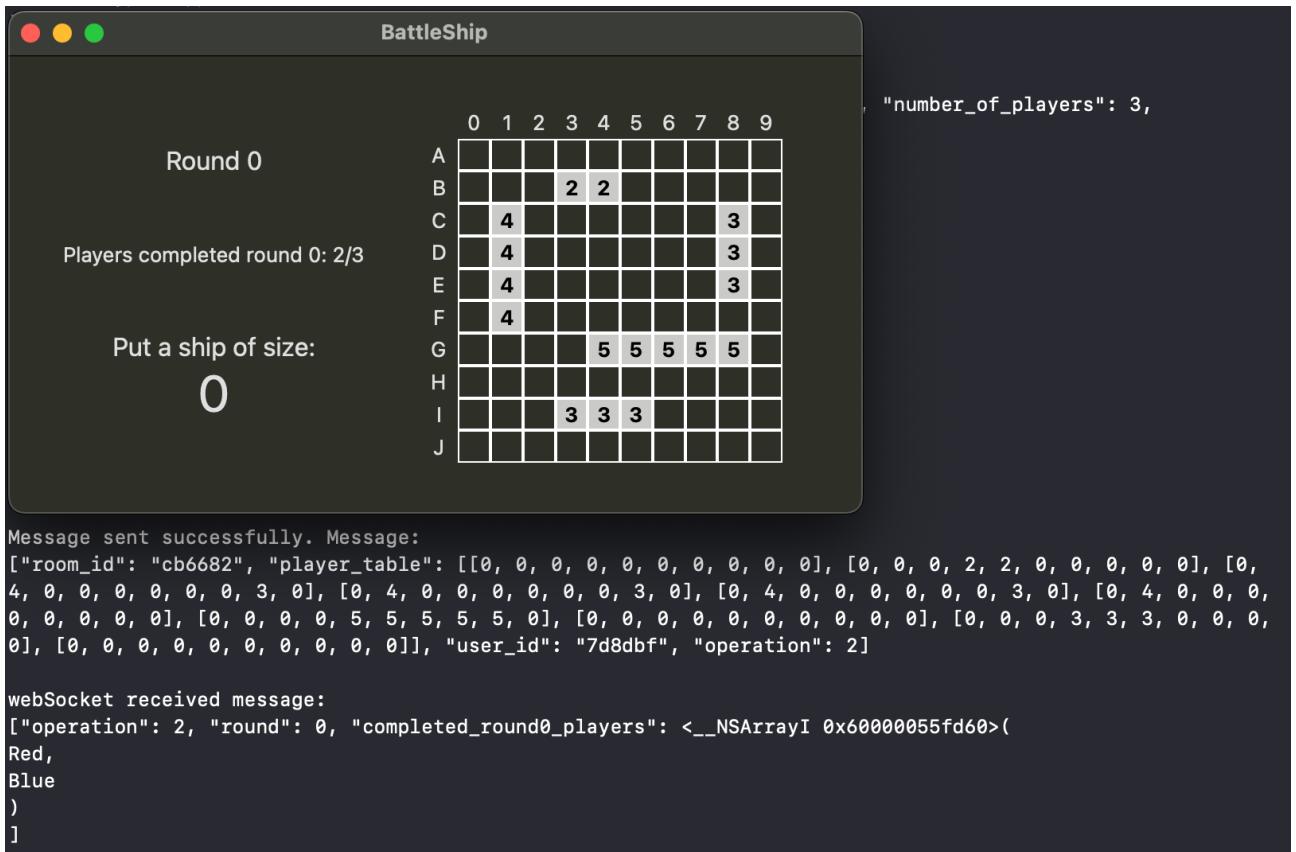


Рис. 16 Играют раунд 0

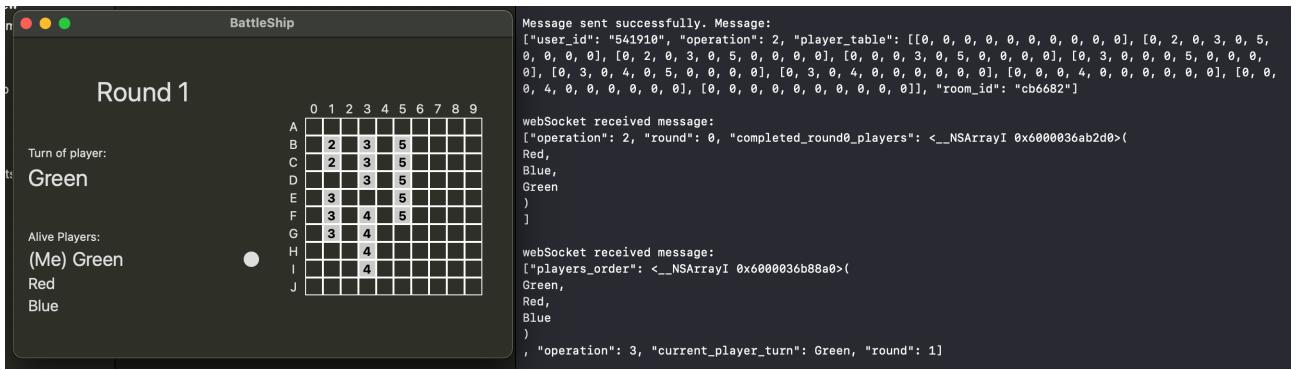


Рис. 17 Играют раунд 0

- Играют раунд 1:

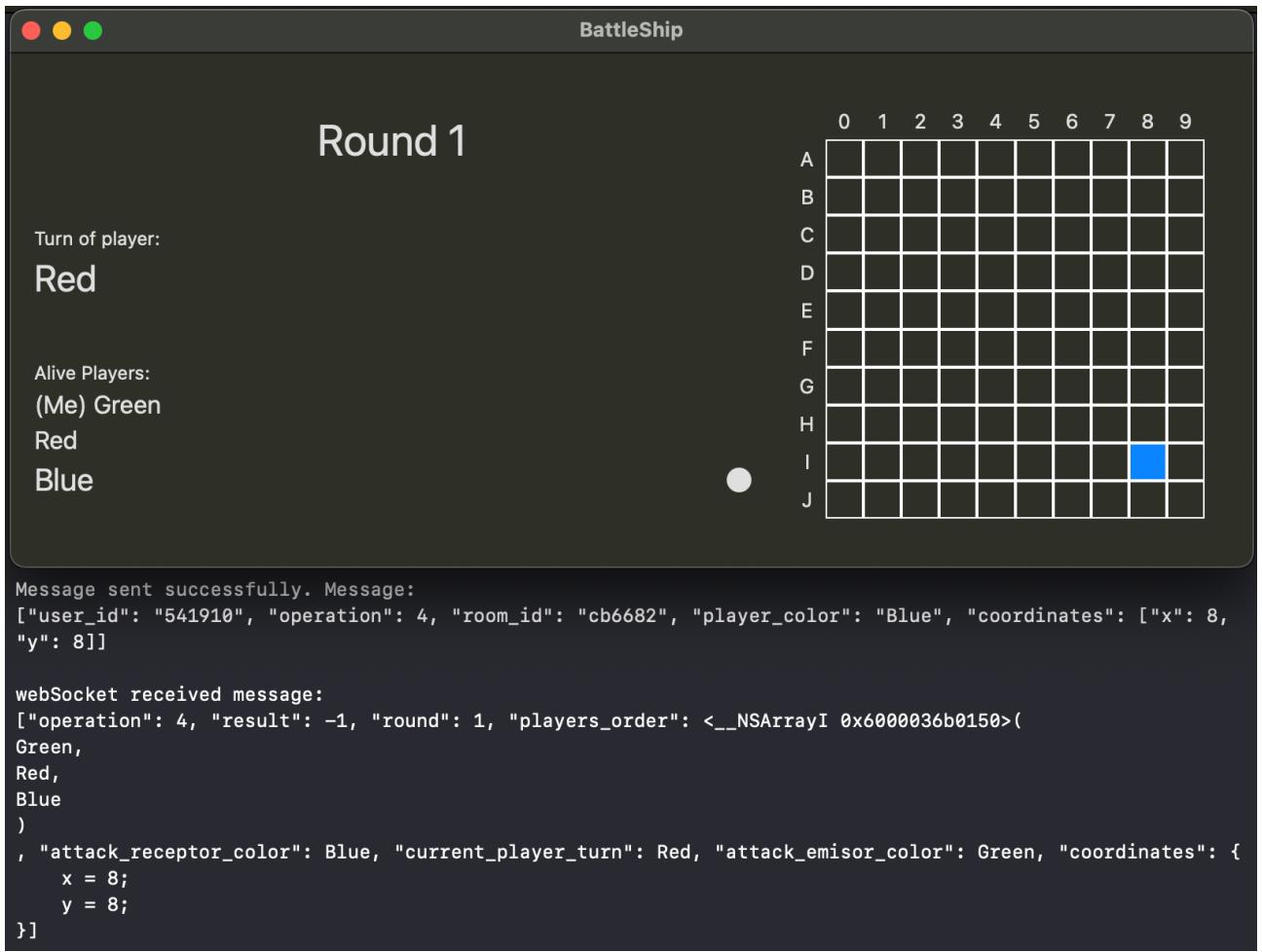


Рис. 18 Игровы играют раунд 1

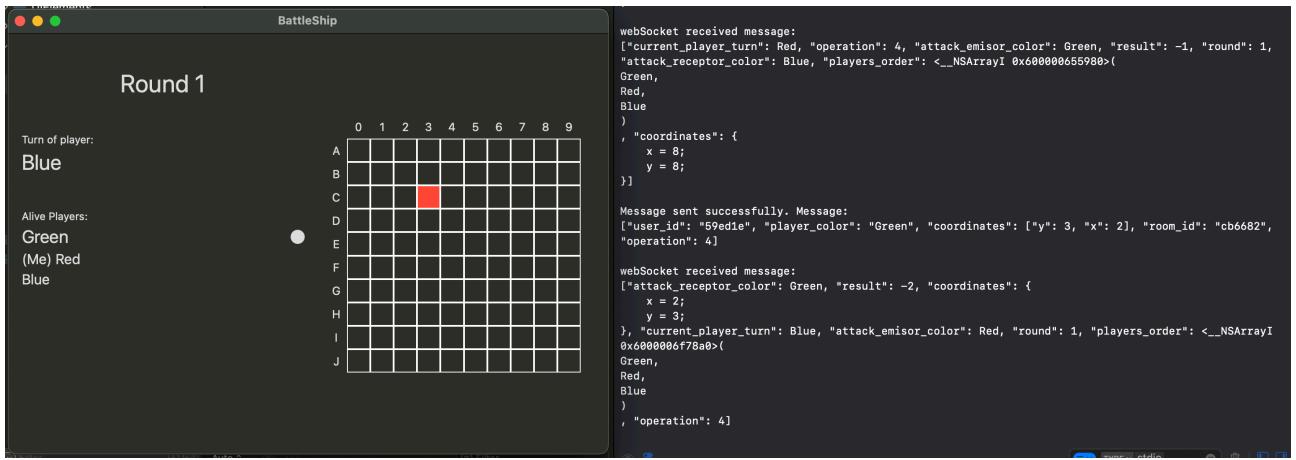


Рис. 19 Игровы играют раунд 1

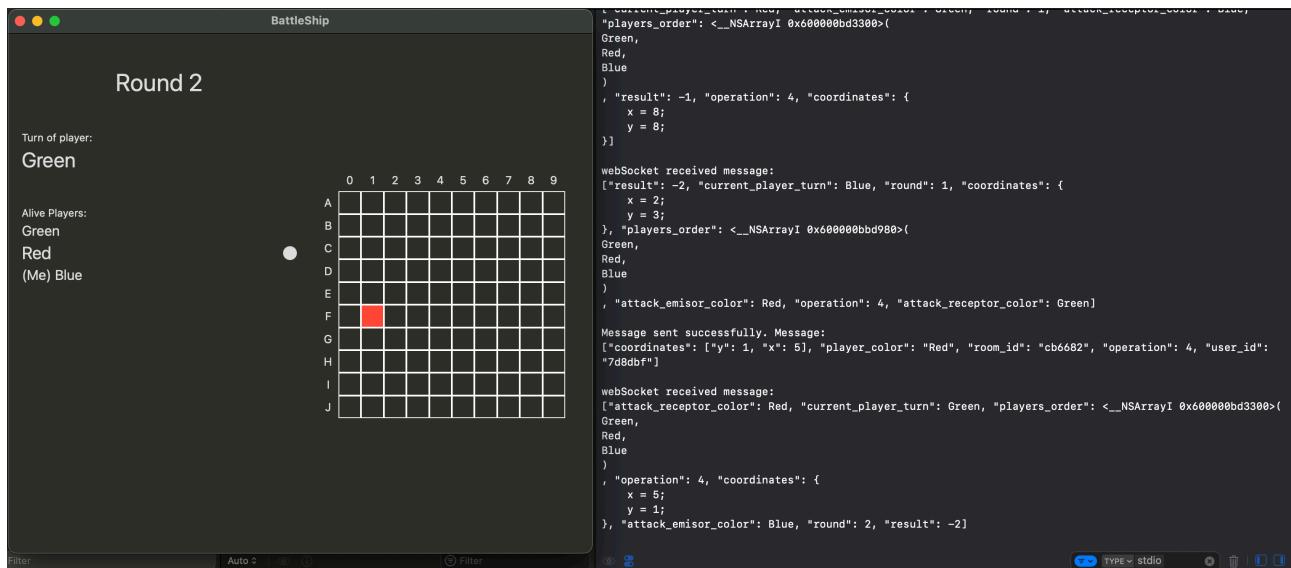


Рис. 20 Играют раунд 1

- Игрок «красный» проиграл:

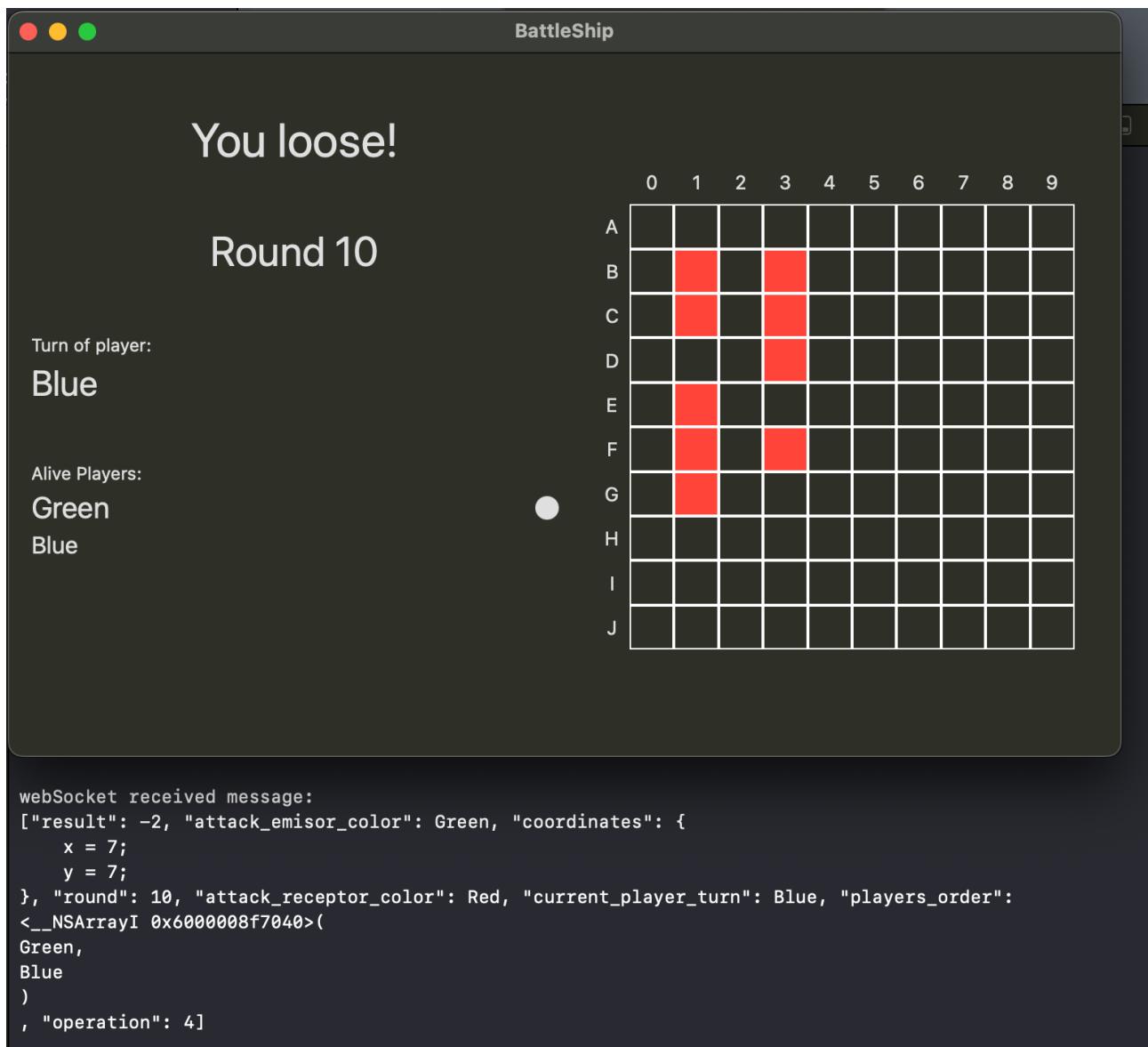


Рис. 21 Игрок «красный» проиграл

- Игроки играют последний раунд:

**BattleShip**

## Round 24

Turn of player:  
**Blue**

Alive Players:  
**Blue**  
(Me) **Green**

```

Message sent successfully. Message:
[{"user_id": "07c4b6", "room_id": "48e281", "coordinates": [{"y": 5, "x": 4}, {"y": 5, "x": 5}], "operation": 4, "player_color": "Blue"}]

webSocket received message:
[{"attack_receptor_color": Blue, "current_player_turn": Blue, "round": 24, "attack_emisor_color": Green, "result": -2, "coordinates": {"x": 4, "y": 5}, "players_order": [Blue, Green]}, {"operation": 4}]

```

Рис. 22 Игроки играют последний раунд

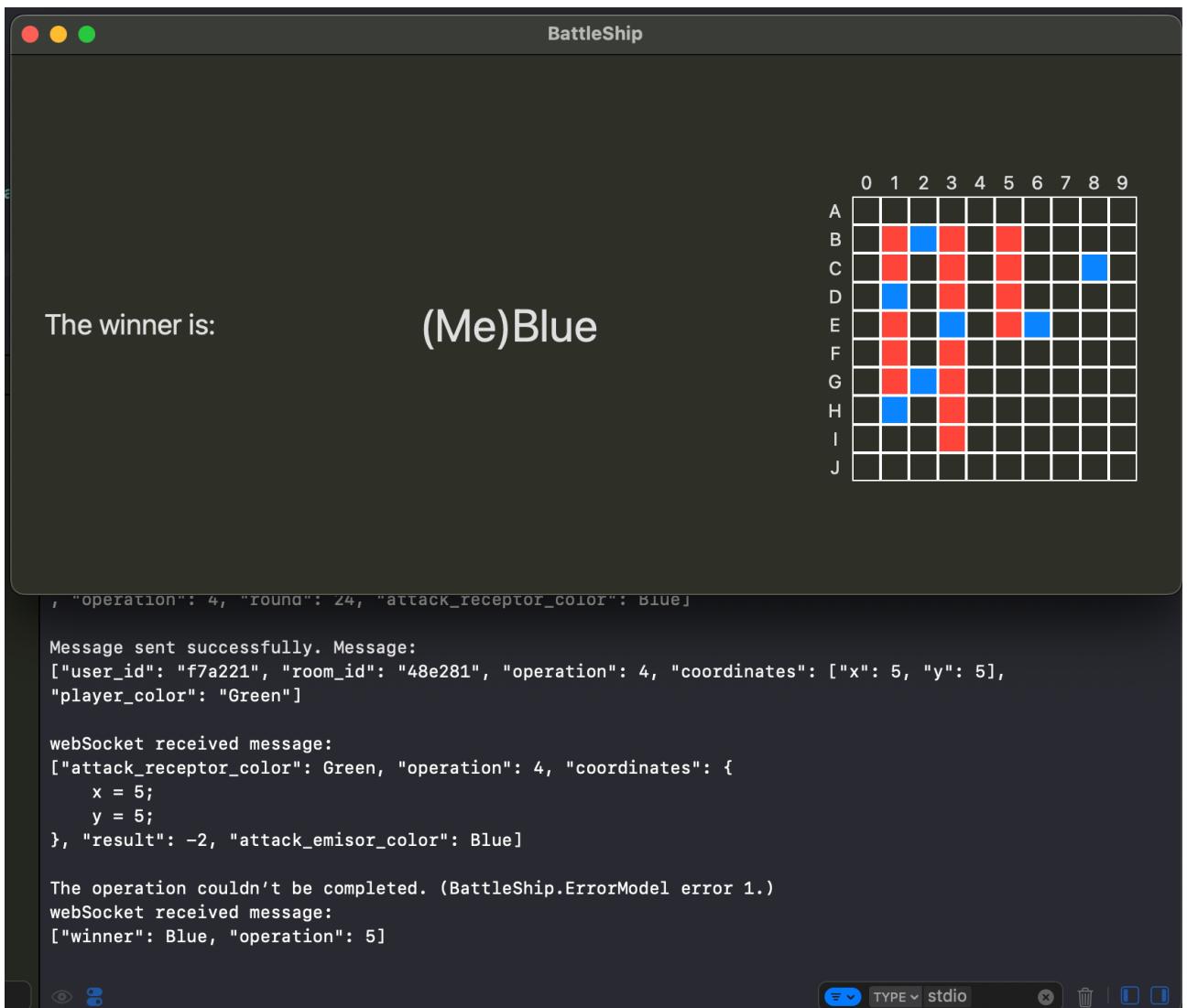


Рис. 23 Играющие игроки играют последний раунд

## Выводы

В этом разделе были описаны языки программирования и модули, используемые для разработки бэкэнда и фронтэнда игры «Морской бой». Также был представлен пример реализованного протокола в работе.

## **ЗАКЛЮЧЕНИЕ**

В заключение следует отметить, что в данной работе был разработан протокол для игры в морской бой для 2 и более игроков. Также была разработана игра в морской бой для 2 и более игроков с использованием указанного выше протокола.

## **СПИСОК ЛИТЕРАТУРЫ**

1. Topic: Online gaming [Электронный ресурс]. URL: <https://www.statista.com/topics/1551/online-gaming> (дата обращения: 21.01.2025).
2. TCP - MDN Web Docs Glossary: Definitions of Web-related terms | MDN [Электронный ресурс]. URL: <https://developer.mozilla.org/en-US/docs/Glossary/TCP> (дата обращения: 02.02.2025).
3. UDP (User Datagram Protocol) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN [Электронный ресурс]. URL: <https://developer.mozilla.org/en-US/docs/Glossary/UDP> (дата обращения: 02.02.2025).
4. The WebSocket API (WebSockets) - Web APIs | MDN [Электронный ресурс]. URL: [https://developer.mozilla.org/en-US/docs/Web/API/WebSockets\\_API](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API) (дата обращения: 21.01.2025).
5. MozDevNet. Обзор протокола HTTP - http: MDN [Электронный ресурс]. URL: <https://developer.mozilla.org/ru/docs/Web/HTTP/Overview> (дата обращения: 02.02.2025).

## **Приложение А – Обработка сообщений WebSocket на сервере**

```
wss.on("connection", (ws, req) => {
    const user_id = req.url.split("=")[1];

    playerConnections.set(user_id, ws);

    ws.on("message", (message) => {
        const data = JSON.parse(message);

        if (data.operation === 1) {
            handleStartGame(ws, data);
        }
        else if (data.operation === 2) {
            handleRound0(ws, data);
        }
        else if (data.operation === 4) {
            handleRound1(ws, data);
        }
    });
});

ws.on("close", () => {
    handlePlayerDisconnection(user_id)
    playerConnections.delete(user_id);
});
});
```

## **Приложение Б - Обработка запросов на присоединение к игре на сервере**

```
app.post('/join-room', (req, res) => {
    const { room_id } = req.body;

    if (!room_id) {
        return res.status(400).json({
            message: 'Missing parameter room_id',
            error_code: 2002,
        });
    }

    if (!rooms.get(room_id)) {
        return res.status(400).json({
            message: 'Invalid room ID',
            error_code: 2001,
        });
    }

    const room = rooms.get(room_id);

    if (room.players.length >= room.maxPlayers) {
```

```

        return res.status(400).json({
            message: 'The room is already full',
            error_code: 2003,
        });
    }

    if (room.round >= 0) {
        return res.status(400).json({
            message: 'The game in the room are already started',
            error_code: 2004,
        });
    }

    const user_id = uuidv4().replace(/-/g, "").substring(0, 6);

    room.players.push(user_id);
    users.set(user_id, room_id)

    room.players.forEach((playerId) => {
        const ws = playerConnections.get(playerId);
        if (ws) {
            ws.send(
                JSON.stringify({
                    operation: 0,
                    number_of_current_players: room.players.length,
                })
            );
        }
    });

    res.status(200).json({
        user_id: user_id,
        room_id: room_id,
        websocket_url: websocketUrl + "?userId=" + user_id,
        number_of_players: room.maxPlayers,
        number_of_current_players: room.players.length,
    });
}
);

```

## **Приложение В - Обработка запроса на создание игры на сервере**

```

app.post('/create-room', (req, res) => {
    console.log("create room")
    const { number_of_players } = req.body;

    if (!Number.isInteger(number_of_players)) {
        return res.status(400).json({
            error_code: 1002,
            message: 'Missing parameter number_of_players',
        });
    }
}
);

```

```

if (number_of_players <= 1) {
    return res.status(400).json({
        error_code: 1001,
        message: 'The number of players need to be 2 or more',
    });
}

const userId = uuidv4().replace(/-/g, "").substring(0, 6);
const roomId = uuidv4().replace(/-/g, "").substring(0, 6);

rooms.set(roomId, {
    owner: userId,
    players: [userId],
    maxPlayers: number_of_players,
    round: -1
});

users.set(userId, roomId)

return res.status(200).json({
    user_id: userId,
    room_id: roomId,
    websocket_url: websocketUrl + "?userId=" + userId,
});
}
);

```

## Приложение Г - Обработка отключений игроков (сервер)

```

function handlePlayerDisconnection(userId) {
    const roomId = users.get(userId)
    const room = rooms.get(roomId)

    if (room.round < 0) {
        room.players = removeElement(room.players, userId)
        room.players.forEach((playerId) => {
            const ws = playerConnections.get(playerId);
            if (ws) {
                ws.send(
                    JSON.stringify({
                        operation: 0,
                        number_of_current_players: room.players.length,
                    })
                );
            }
        });
        return
    }
    else if (room.round == 0) {
        room["players_tables"][user_id] = null
        room["players_ready_user_id"] = removeElement(room["players_ready_user_id"], userId)
    }
}

```

```

        room["players_ready_color"] = removeElement(room["players_ready_color"],
room["players_colors"][user_id])
        room.players.forEach((playerId) => {
            const ws = playerConnections.get(playerId);
            if (ws) {
                ws.send(
                    JSON.stringify({
                        operation: 2,
                        round: room["round"],
                        completed_round0_players: room["players_ready_color"]
                    })
                );
            }
        });
        return
    }

const current_turn = room["players_order"][room["current_order"]]

if (room.players_order.indexOf(current_turn) >
room.players_order.indexOf(room.players_colors[userId])) {
    room.current_order = room.current_order -2
    nextTurn(room)
}
room.players_order = removeElement(room.players_order, current_turn)

if (room.players_order.length == 1) {
    room.players.forEach((playerId) => {
        const ws = playerConnections.get(playerId);
        if (ws) {
            ws.send(
                JSON.stringify({
                    operation: 5,
                    winner: room.players_order[0],
                })
            );
        }
    });
}
else {
    room.players.forEach((playerId) => {
        const ws = playerConnections.get(playerId);
        if (ws) {
            ws.send(
                JSON.stringify({
                    operation: 6,
                    round: room["round"],
                    current_player_turn: room["players_order"][room["current_order"]],
                    players_order: room["players_order"]
                })
            );
        }
    });
}

```

```
    });
}
}
```

## Приложение Д - Обработка сообщения о начале игры (сервер)

```
function handleStartGame(ws, data) {
    const { operation, user_id, room_id } = data;
    if (!rooms.has(room_id)) {
        return ws.send(
            JSON.stringify({
                message: "Invalid room_id",
                error_code: 3002,
            })
        );
    }

    const room = rooms.get(room_id);
    if (room.owner != user_id) {
        return ws.send(
            JSON.stringify({
                message: "The game can only be started by the creator of the room",
                error_code: 3003,
            })
        );
    }

    if (room.round != -1) {
        return ws.send(
            JSON.stringify({
                message: "The game in the room are alredy started",
                error_code: 3004,
            })
        );
    }

    if (room.players.length < 2) {
        return ws.send(
            JSON.stringify({
                message: "The number of players in the room need to be at least 2",
                error_code: 3001,
            })
        );
    }

    const playerColors = {};
    const intern_colors_by_user_id = {};
    room.players.forEach((playerId, index) => {
        playerColors[playerId] = availableColors[index % availableColors.length];
        intern_colors_by_user_id[playerColors[playerId]] = playerId
    })
}
```

```

    });

room["round"] = 0
room["players_ready_user_id"] = []
room["players_ready_color"] = []
room["players_colors"] = playerColors
room["intern_colors_by_user_id"] = intern_colors_by_user_id
room["players_tables"] = new Map()

room.players.forEach((playerId) => {
  const ws = playerConnections.get(playerId);
  if (ws) {
    ws.send(
      JSON.stringify({
        operation: 1,
        round: room["round"],
        user_color: room["players_colors"][playerId],
      })
    );
  }
});
}

```

## Приложение Е - Обработка сообщения раунда 0 (сервер)

```

function handleRound0(ws, data) {
  const { operation, user_id, room_id, player_table } = data;
  if (!user_id || !room_id) {
    return ws.send(
      JSON.stringify({
        message: "Missing parameters",
        error_code: 4003,
      })
    );
  }

  if (!rooms.has(room_id)) {
    return ws.send(
      JSON.stringify({
        message: "Invalid room_id",
        error_code: 4002,
      })
    );
  }

  const room = rooms.get(room_id);
  if (room.round != 0) {
    return ws.send(
      JSON.stringify({
        message: "The round 0 has already ended or not started yet",
      })
    );
  }
}

```

```

        error_code: 4004,
    })
);
}

if (!room.players.includes(user_id)) {
    return ws.send(
        JSON.stringify({
            message: "Invalid user_id",
            error_code: 4005,
        })
    );
}

if (room.players_ready_user_id.includes(user_id)) {
    return ws.send(
        JSON.stringify({
            message: "The user has already played his round 0",
            error_code: 4006,
        })
    );
}

if (!checkTableCorrectness(player_table)) {
    return ws.send(
        JSON.stringify({
            Message: "Invalid table format",
            Error_code: 4001,
        })
    );
}

room["players_tables"][user_id] = player_table
room["players_ready_user_id"].push(user_id)
room["players_ready_color"].push(room["players_colors"][user_id])

room.players.forEach((playerId) => {
    const ws = playerConnections.get(playerId);
    if (ws) {
        ws.send(
            JSON.stringify({
                operation: 2,
                round: room["round"],
                completed_round0_players: room["players_ready_color"]
            })
        );
    }
});

if (room.players_ready_user_id.length == room.players.length) {
    room["players_order"] = Object.values(room.players_colors)
    room["current_order"] = 0
}

```

```

room["round"] = 1
startTimer(room["intern_colors_by_user_id"][room["players_order"]][room["current_order"]])
room.players.forEach((playerId) =>
  const ws = playerConnections.get(playerId);
  if (ws) {
    ws.send(
      JSON.stringify({
        operation: 3,
        round: room["round"],
        players_order: room["players_order"],
        current_player_turn: room["players_order"][room["current_order"]],
      })
    );
  }
)
}

```

## Приложение Ё - Обработка сообщения раунда 1 (сервер)

```

function handleRound1(ws, data) {
  const { operation, user_id, room_id, player_color, coordinates } = data;

  if (coordinates.x < 0 || coordinates.y < 0 || coordinates.x > 9 || coordinates.y > 9) {
    return ws.send(
      JSON.stringify({
        message: "Invalid coordinates",
        error_code: 5001,
      })
    );
  }

  if (!user_id || !room_id) {
    return ws.send(
      JSON.stringify({
        message: "Missing parameters",
        error_code: 5003,
      })
    );
  }

  if (!rooms.has(room_id)) {
    return ws.send(
      JSON.stringify({
        message: "Invalid room_id",
        error_code: 5002,
      })
    );
  }
}

```

```

const room = rooms.get(room_id);

if (room.round < 1) {
    return ws.send(
        JSON.stringify({
            Message: 'The game current round is 0 or below',
            Error_code: 5004,
        })
    );
}

if (!room.players.includes(user_id)) {
    return ws.send(
        JSON.stringify({
            Message: "Invalid user_id ",
            Error_code: 5008,
        })
    );
}

const current_turn = room["players_order"][room["current_order"]]

if (current_turn != room.players_colors[user_id]) {
    return ws.send(
        JSON.stringify({
            Message: "Is the turn of another player",
            Error_code: 5007,
        })
    );
}

const attacked_user_id = room.intern_colors_by_user_id[player_color]

if (!attacked_user_id || !room.players_order.includes(player_color) || attacked_user_id == user_id) {
    return ws.send(
        JSON.stringify({
            Message: "The attacked player is not valid",
            Error_code: 5005,
        })
    );
}

const cell_value = room.players_tables[attacked_user_id][coordinates.x][coordinates.y]

if (cell_value < 0) {
    return ws.send(
        JSON.stringify({
            Message: "Some player already attacked that cell of the player",
            Error_code: 5006,
        })
    );
}

```

```

    );
}

else if (cell_value == 0) {
    room.players_tables[attacked_user_id][coordinates.x][coordinates.y] = -1
    nextTurn(room)
    startTimer(room["intern_colors_by_user_id"][room["players_order"]][room["current_order"]])
    room.players.forEach((playerId) => {
        const ws = playerConnections.get(playerId);
        if (ws) {
            ws.send(
                JSON.stringify({
                    operation: 4,
                    attack_emisor_color: current_turn,
                    attack_receptor_color: player_color,
                    coordinates: coordinates,
                    result: -1,
                    round: room["round"],
                    current_player_turn: room["players_order"][room["current_order"]],
                    players_order: room["players_order"],
                })
            );
        }
    });
}
else {
    if (room.players_tables[attacked_user_id]) {
        room.players_tables[attacked_user_id][coordinates.x][coordinates.y] = -cell_value
        nextTurn(room)

        startTimer(room["intern_colors_by_user_id"][room["players_order"]][room["current_order"]])
        if (checkIfPlayerDie(room.players_tables[attacked_user_id])) {
            room.players_order = removeElement(room.players_order, player_color)
        }

        if (room.players_order.length == 1) {
            room.players.forEach((playerId) => {
                const ws = playerConnections.get(playerId);
                if (ws) {
                    ws.send(
                        JSON.stringify({
                            operation: 4,
                            attack_emisor_color: current_turn,
                            attack_receptor_color: player_color,
                            coordinates: coordinates,
                            result: -2,
                        })
                );
            });

            ws.send(
                JSON.stringify({
                    operation: 5,
                    winner: room.players_order[0],
                })
            );
        }
    }
}

```

```

        })
    );
}
else {
    room.players.forEach((playerId) => {
        const ws = playerConnections.get(playerId);
        if (ws) {
            ws.send(
                JSON.stringify({
                    operation: 4,
                    attack_emisor_color: current_turn,
                    attack_receptor_color: player_color,
                    coordinates: coordinates,
                    result: -2,
                    round: room["round"],
                    current_player_turn: room["players_order"][room["current_order"]],
                    players_order: room["players_order"]
                })
            );
        }
    });
}
}

```

## Приложение Ж - Обработка HTTP-запросов (клиент)

```

class NetworkService {
    static let shared = NetworkService()

    private init() {}

    func httpRequest<T: Decodable>(
        endpoint: String,
        method: httpMethod,
        body: Codable? = nil,
        responseType: T.Type
    ) async throws -> T {
        guard let url = URL(string: endpoint) else {
            throw URLError(.badURL)
        }

        var request = URLRequest(url: url)
        request.httpMethod = method.rawValue
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
    }
}

```

```

if let body = body {
    request.httpBody = try JSONEncoder().encode(body)
}
print("HTTP Request:")
print(endpoint)
print(method.rawValue)
print("Content-Type: application/json")
print(body ?? "")
print("")

let (data, response) = try await URLSession.shared.data(for: request)

guard let httpResponse = response as? HTTPURLResponse,
(200...299).contains(httpResponse.statusCode) else {
    throw URLError(.badServerResponse)
}
print("HTTP Response from server:")
print(try JSONSerialization.jsonObject(with: data, options: []) as? [String: Any] ?? "")
print("")

return try JSONDecoder().decode(T.self, from: data)
}

enum httpMethod: String {
    case get = "GET"
    case post = "POST"
    case put = "PUT"
    case delete = "DELETE"
}

extension NetworkService {
    public func createRoom(numberOfPlayers: Int) async throws -> RoomInfoModel {
        return try await self.httpRequest(
            endpoint: "https://127.0.0.1:3000/create-room",
            method: .post,
            body: ["number_of_players": numberOfPlayers],
            responseType: createRoomResponseMapper.self
        )
        .execute()
    }

    public func JoinRoom(roomId: String) async throws -> RoomInfoModel {
        return try await self.httpRequest(
            endpoint: "https://127.0.0.1:3000/join-room",
            method: .post,
            body: ["room_id": roomId],
            responseType: createRoomResponseMapper.self
        )
        .execute()
    }
}

```

## Приложение 3 - Обработка сообщений WebSocket (клиент)

```
class WebSocketService: ObservableObject {
    static let shared = WebSocketService()

    private var webSocketTask: URLSessionWebSocketTask?
    private var session: URLSession?

    @Published var receivedMessage: [String: Any] = [:]
    @Published var isConnected: Bool = false

    @Published var messagesReceived: Int = 0

    @Published var errorMessage: (String, Int) = ("", 0)

    private init() {
        self.session = URLSession(configuration: .default)
    }

    func connect(to url: URL) {
        webSocketTask = session?.websocketTask(with: url)
        webSocketTask?.resume()
        isConnected = true
    }

    receiveMessages()

    print("WebSocket connected to \(url)")
}

func sendMessage(_ message: [String: Any]) {
    do {
        let data = try JSONSerialization.data(withJSONObject: message, options: [])
        let jsonString = String(data: data, encoding: .utf8) ?? ""
        let webSocketMessage = URLSessionWebSocketTask.Message.string(jsonString)

        webSocketTask?.send(webSocketMessage) { [weak self] error in
            if let error = error {
                self?.errorMessage = ("Error sending message: \(error)", 1)
                print("Error sending message: \(error)")
            } else {
                print("Message sent successfully. Message:")
                print(message)
                print("")
            }
        }
    } catch {
        self.errorMessage = ("Error serializing message: \(error)", 2)
        print("Error serializing message: \(error)")
    }
}
```

```

private func receiveMessages() {
    webSocketTask?.receive { [weak self] result in
        switch result {
        case .failure(let error):
            Task { @MainActor in
                self?.errorMessage = ("Error receiving message: \(error)", 3)
            }
            print("Error receiving message: \(error)")
        case .success(let message):
            switch message {
            case .string(let text):
                if let data = text.data(using: .utf8) {
                    do {
                        if let json = try JSONSerialization.jsonObject(with: data, options: []) as? [String: Any] {
                            DispatchQueue.main.async {
                                Task { @MainActor in
                                    self?.receivedMessage = json
                                    self?.messagesReceived += 1
                                }
                                print("webSocket received message:")
                                print(json)
                                print("")
                            }
                        }
                    } catch {
                        Task { @MainActor in
                            self?.errorMessage = ("Error deserializing message: \(error)", 4)
                        }
                        print("Error deserializing message: \(error)")
                    }
                }
            }
        case .data(let data):
            print("Received binary data: \(data)")
        @unknown default:
            break
        }
    }
}

self?.receiveMessages()
}

func disconnect() {
    webSocketTask?.cancel(with: .normalClosure, reason: nil)
    isConnected = false
}
}

struct BattleShipProtocolMessagesHandler {

public static let shared = BattleShipProtocolMessagesHandler()

```

```

private init() {}

private func handle<M: MapperProtocol, O>(message: [String: Any], operation: Int, mapper: M.Type, output: O.Type) -> Result<O, ErrorModel> {
    guard let data = try? JSONSerialization.data(withJSONObject: message, options: []) else
    {return .failure(.init(code: 0, message: "Error on converting message to data"))}
    if let op = message["operation"] as? Int,
       op == operation, let mapper = try? JSONDecoder().decode(M.self, from: data),
       let response = mapper.execute() as? O
    {
        return .success(response)
    }
    else if let error = try? JSONDecoder().decode(ErrorMapper.self, from: data) {
        return .failure(error.execute())
    }
    else {
        return .failure(.init(code: 1, message: "Error on parsing message"))
    }
}

public func newPlayerInRoom(message: [String: Any]) -> Result<Int, ErrorModel> {
    handle(message: message, operation: 0, mapper: NewPlayerInRoomMapper.self, output: Int.self)
}

public func gameStarted(message: [String: Any]) -> Result<GameStatusModel, ErrorModel> {
    handle(message: message, operation: 1, mapper: GameStartedMapper.self, output: GameStatusModel.self)
}

public func round0Status(message: [String: Any]) -> Result<Round0StatusModel, ErrorModel> {
    handle(message: message, operation: 2, mapper: Round0StatusMapper.self, output: Round0StatusModel.self)
}

public func round1StartStatus(message: [String: Any]) -> Result<Round1BasicStatusModel, ErrorModel> {
    handle(message: message, operation: 3, mapper: Round1BasicStatusMapper.self, output: Round1BasicStatusModel.self)
}

public func round1Status(message: [String: Any]) -> Result<Round1StatusModel, ErrorModel> {
    handle(message: message, operation: 4, mapper: Round1StatusMapper.self, output: Round1StatusModel.self)
}

public func round1BaseStatus(message: [String: Any]) -> Result<Round1BasicStatusModel, ErrorModel> {

```

```
    handle(message: message, operation: 6, mapper: Round1BasicStatusMapper.self, output:  
Round1BasicStatusModel.self)  
}  
  
public func endGame(message: [String: Any]) -> Result<String, ErrorModel> {  
    handle(message: message, operation: 5, mapper: EndGameMapper.self, output: String.self)  
}  
}
```