

Реализация механизма валидации и синхронизации игрового состояния в клиент-серверной браузерной системе

Саяпин Никита Анатольевич

Приамурский государственный университет им.Шолом-Алейхема

Студент

Научный руководитель:

Лучанинов Дмитрий Васильевич

Приамурский государственный университет имени Шолом-Алейхема

Старший преподаватель кафедры информационных систем и защиты информации

Аннотация

В статье рассматривается механизм обеспечения целостности данных и синхронизации состояния в браузерной клиент-серверной системе на примере многопользовательской игры. Предлагаемый механизм основан на серверной валидации входных данных, централизованном хранении игрового состояния и событийной модели обновлений через постоянное соединение. Подход направлен на предотвращение нарушений правил, поддержание согласованности между участниками и минимизацию рисков манипуляций. Теоретические аспекты механизма иллюстрируются практическим примером реализации в системе «Морской бой», где сервер выступает в роли арбитра игрового процесса.

Ключевые слова: клиент-серверная архитектура, валидация данных, синхронизация состояния, игровая логика, WebSocket, браузерные приложения.

Implementation of the validation and game state synchronization mechanism in a browser-based client-server system

Sayapin Nikita Anatolyevich

Sholom-Aleichem Priamursky State University

Student

Scientific supervisor:

Luchaninov Dmitry Vasilyevich

Sholom-Aleichem Priamursky State University

Senior Lecturer, Department of Information Systems and Information Security

Abstract

The article examines a mechanism for ensuring data integrity and state synchronization in a browser-based client-server system using the example of a

multi-user game. The proposed mechanism is based on server-side validation of input data, centralized storage of the game state, and an event-based update model via a persistent connection. The approach is aimed at preventing rule violations, maintaining consistency between participants, and minimizing the risks of manipulation. The theoretical aspects of the mechanism are illustrated by a practical example of implementation in the Sea Battle system, where the server acts as an arbiter of the game process.

Keywords: client-server architecture, data validation, state synchronization, game logic, WebSocket, browser applications.

1. Введение

1.1 Актуальность

В современных браузерных приложениях, ориентированных на многопользовательское взаимодействие, обеспечение целостности данных и синхронизации состояния является ключевым фактором для поддержания справедливости и стабильности системы. Особую роль это играет в онлайн-играх, где пользователи могут пытаться обходить правила через модификацию клиентского кода или некорректные запросы. Традиционные подходы, основанные исключительно на клиентской логике, уязвимы к таким вмешательствам, что приводит к несогласованности данных между участниками и потере доверия к системе.

Актуальность разработки надежного механизма валидации и синхронизации обусловлена ростом популярности распределенных веб-приложений, где сервер должен выступать в роли единственного источника достоверной информации. В контексте игр реального времени, таких как «Морской бой», это позволяет гарантировать соблюдение правил расстановки объектов, очередности действий и корректности исходов, минимизируя задержки и повышая пользовательский опыт. Без такого механизма система становится подвержена ошибкам, что особенно критично в образовательных или развлекательных проектах, требующих строгой логики.

1.2 Обзор исследований

Ю.А. Ушаков рассматривает реализацию модуля сервера видеоконференцсвязи на базе WebRTC и Janus для интеграции в систему Moodle, демонстрируя, как постоянное соединение обеспечивает синхронизацию данных в реальном времени между клиентом и сервером, с акцентом на обработку событий и управление состоянием в распределённых веб-приложениях [1]. Л.А. Нежелская и А.А. Першина анализируют процедуры оценивания параметров в асинхронных потоках событий, подчеркивая важность точной синхронизации и валидации временных данных в клиент-серверных системах реального времени для предотвращения ошибок в распределённых информационных потоках [2]. К.Г. Дамирбек и Э.А. Жакыпова исследуют обеспечение целостности информации в автоматизированных системах, показывая, что серверная валидация данных и централизованное управление состоянием позволяют гарантировать

согласованность в многопользовательских веб-приложениях с интенсивным обменом [3]. А.А. Романов, Д.Г. Волчек и Д.И. Муромцев описывают семантические технологии для экстракции и синхронизации данных в онлайн-курсах на платформе Open edX, иллюстрируя, как клиент-серверная архитектура с постоянными соединениями обеспечивает актуальность состояния и валидацию обогащенных данных в реальном времени [4]. Н.Д. Колесников и С.Ю. Щербина рассматривают методы мошенничества, основанные на перехвате и модификации сетевого трафика. Особое внимание уделяется отличительным особенностям сетевого трафика, характерного для многопользовательских игр. В работе подробно рассмотрены средства криптографической защиты информации, применяемые для сетевого трафика [5].

1.3 Цель исследования

Целью исследования является описание и обоснование механизма валидации данных и синхронизации игрового состояния в браузерной клиент-серверной системе, обеспечивающего предотвращение нарушений правил, согласованность данных между пользователями и устойчивость к внешним вмешательствам.

2 Материалы и методы

2.1 Общая постановка задачи

Рассматривается задача обеспечения целостности и синхронизации данных в распределенной браузерной системе с многопользовательским взаимодействием. Необходимо реализовать механизм, который предотвращает некорректные действия на клиентской стороне, централизованно хранит состояние системы и оперативно обновляет информацию у всех участников.

Специфика таких систем заключается в необходимости обработки динамических событий в реальном времени, где клиенты генерируют входные данные (например, координаты действий), а сервер проверяет их на соответствие правилам. В качестве примера используется браузерная игра «Морской бой», где требуется валидация расстановки объектов, контроль очередности и фиксация результатов.

2.2 Требования к механизму валидации и синхронизации

Механизм валидации и синхронизации должен обеспечивать серверную проверку всех критических данных, поступающих от клиентов, с целью полного исключения возможности манипуляций со стороны пользователя. Централизованное хранение игрового состояния на сервере гарантирует согласованность информации между участниками, предотвращая расхождения, которые могли бы возникнуть при использовании только клиентской логики. Передача обновлений осуществляется в асинхронном режиме, что позволяет минимизировать задержки при обмене событиями в реальном времени. Кроме того, механизм обязан корректно обрабатывать

нестандартные ситуации, в частности внезапные отключения одного из участников, сохраняя при этом работоспособность сессии для оставшегося игрока и освобождая ресурсы. Реализация должна быть полностью совместима со стандартными веб-технологиями и работать в современных браузерах без установки дополнительного программного обеспечения. Масштабируемость механизма достигается за счёт эффективного использования памяти и поддержки одновременного функционирования множества независимых игровых сессий при умеренной сетевой нагрузке (Рис.1).

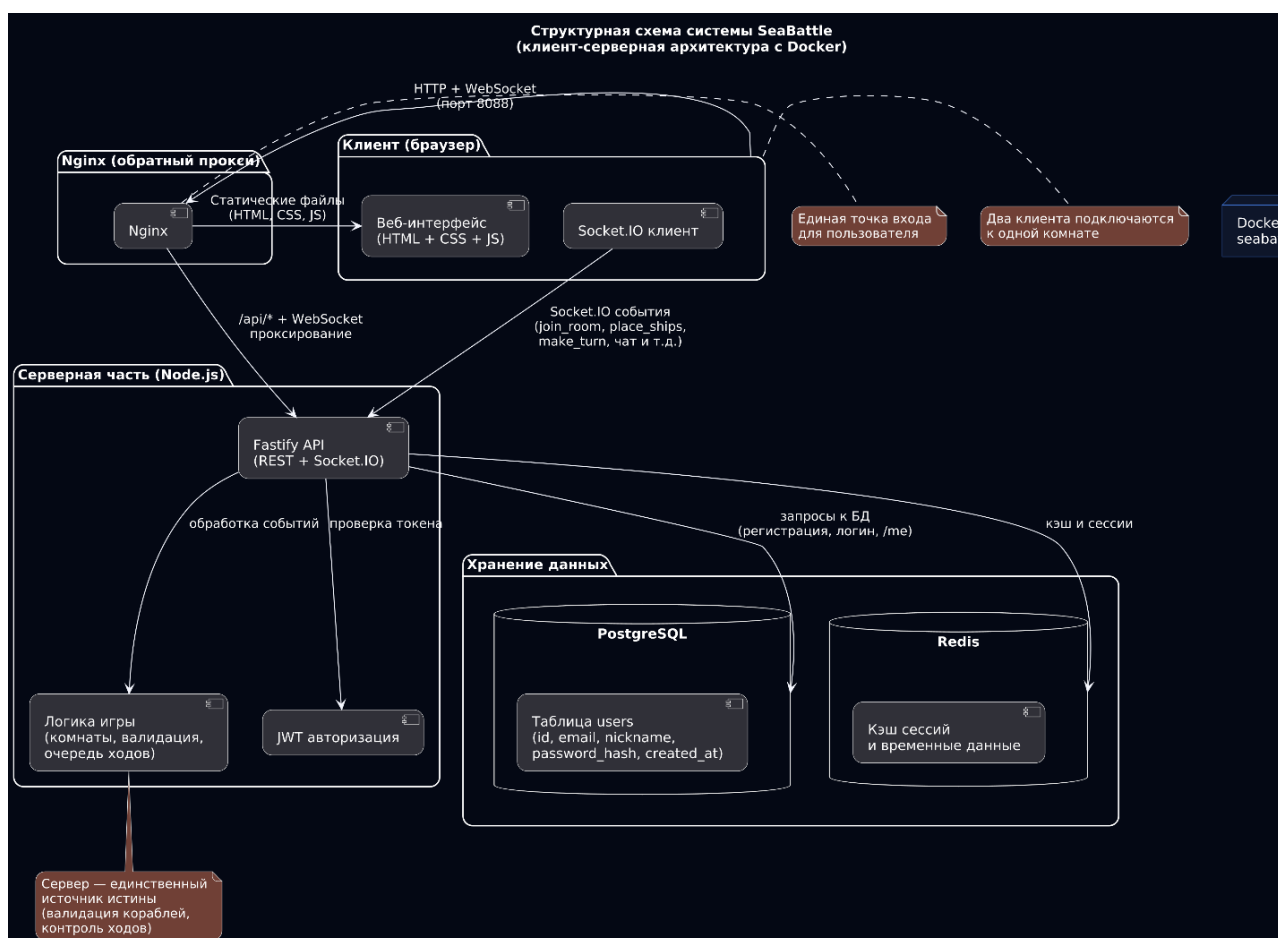


Рисунок 1 – Структурная схема проекта

Предлагаемый механизм функционирует следующим образом: сначала сервер принимает данные от клиента, затем проводит их тщательную валидацию в соответствии с установленными правилами игры. При успешной проверке происходит обновление централизованного состояния игрового процесса, хранящегося исключительно на сервере. После этого сервер формирует и рассылает соответствующие события всем участникам сессии, обеспечивая мгновенную синхронизацию интерфейсов. В случае обнаружения нарушений клиенту отправляется уведомление об отклонении действия с указанием причины, что позволяет пользователю оперативно исправить ошибку. Данный цикл повторяется для каждого значимого события в игре,

гарантируя, что сервер остаётся единственным авторитетным источником достоверной информации (Рис.2).



Рисунок 2 – Диаграмма взаимодействия компонентов

3. Результаты

3.1 Состав программных компонентов

Серверная часть системы реализована на платформе Node.js с использованием веб-фреймворка Fastify, который обеспечивает высокую производительность и удобную работу с HTTP-запросами. Для организации двустороннего взаимодействия в реальном времени применяется библиотека Socket.IO, интегрированная в Fastify через плагин fastify-socket.io. Аутентификация пользователей построена на механизме JWT-токенов с использованием плагина @fastify/jwt, а хэширование паролей выполняется с помощью bcryptjs. Хранение учетных записей осуществляется в реляционной СУБД PostgreSQL, доступ к которой организован через драйвер pg.

Центральным элементом серверной логики является объект `rooms` (Рис.3), представляющий собой ассоциативный массив, где ключом выступает название комнаты, а значением — объект, содержащий информацию о текущей игровой сессии. В структуре каждой комнаты хранятся идентификаторы участников (массив `players`), идентификатор игрока, чей сейчас ход (`turn`), а также объект `boards`, в котором для каждого пользователя сохраняются данные о расстановке кораблей (`ships`), множестве полученных попаданий (`hits`) и произведенных выстрелов (`shots`).

```
if (!rooms[room]) rooms[room] = { players: [], turn: null, boards: {} };
const game = rooms[room];

if (!game.players.includes(userId)) game.players.push(userId);
if (!game.boards[userId]) {
  game.boards[userId] = { ships: null, hits: new Set(), shots: new Set() };
}
```

Рисунок 3 – Структура объекта хранения состояния комнаты на сервере

Клиентская часть представляет собой одностраничное веб-приложение, написанное на чистом JavaScript без использования дополнительных фреймворков. Интерфейс реализован с помощью HTML и CSS, обеспечивая адаптивность и современный внешний вид. Взаимодействие с сервером осуществляется через библиотеку Socket.IO-client: сначала выполняются HTTP-запросы для регистрации и аутентификации, а после получения JWT-токена устанавливается постоянное WebSocket-соединение с передачей токена в параметрах auth.

3.2 Реализация серверной валидации расстановки кораблей

Ключевым элементом механизма валидации является функция `validateLayout`, которая выполняет полный анализ предложенной игроком расстановки флота. Функция последовательно проверяет количество и длины кораблей на соответствие стандартному набору (один четырёхпалубный, два трёхпалубных, три двухпалубных и четыре однопалубных), линейность расположения каждого корабля (горизонтально или вертикально без разрывов), нахождение всех клеток в пределах игрового поля 10×10, отсутствие пересечений между кораблями и недопустимость касания кораблей даже по диагонали (правило «без касания»)(Рис.4).

Для эффективной проверки пересечений и касаний используется множество `taken`, в которое последовательно добавляются координаты занятых клеток с учётом восьми соседних позиций для каждого размещаемого корабля. При обнаружении любого нарушения функция возвращает объект с признаком `ok: false` и кодом ошибки, который затем передаётся клиенту для отображения причины отклонения.

```

17 // validateLayout
31 const SIZE = 10;
32 const REQUIRED_SHIPS = [4, 3, 3, 2, 2, 2, 1, 1, 1, 1].sort((a, b) => b - a);
33 const k = (x, y) => `${x},${y}`;
34 const cellsFromShip = (ship) => ship.cells.map(c => k(c.x, c.y));
35
36 function validateLayout(layout) {
37   if (!Array.isArray(layout) || layout.length !== REQUIRED_SHIPS.length)
38     return { ok: false, code: 'INVALID_SHIP_COUNT' };
39
40   const lengths = layout.map(s => s.len).sort((a, b) => b - a);
41   if (lengths.join(',') !== REQUIRED_SHIPS.join(','))
42     return { ok: false, code: 'WRONG_LENGTH_SET' };
43
44   const inBounds = (x, y) => x >= 0 && x < SIZE && y >= 0 && y < SIZE;
45   const neigh8 = (x, y) => [
46     [x - 1, y - 1], [x, y - 1], [x + 1, y - 1],
47     [x - 1, y], [x, y], [x + 1, y],
48     [x - 1, y + 1], [x, y + 1], [x + 1, y + 1]
49   ];
50   const taken = new Set();
51
52   for (const s of layout) {
53     if (!Array.isArray(s.cells) || s.cells.length !== s.len)
54       return { ok: false, code: 'LENGTH_MISMATCH' };
55
56     const xs = [...new Set(s.cells.map(c => c.x))];
57     const ys = [...new Set(s.cells.map(c => c.y))];
58     const horiz = ys.length === 1 && xs.length === s.len;
59     const vert = xs.length === 1 && ys.length === s.len;
60     if (!horiz && !vert) return { ok: false, code: 'NOT_LINEAR' };
61
62     if (horiz) {
63       const y = ys[0];
64       const X = s.cells.map(c => c.x).sort((a, b) => a - b);
65       for (let i = 1; i < X.length; i++)
66         if (X[i] !== X[i - 1] + 1) return { ok: false, code: 'GAPS' };
67       for (const x of X) {
68         if (!inBounds(x, y)) return { ok: false, code: 'OUT_OF_BOUNDS' };
69         const kk = k(x, y);
70         if (taken.has(kk)) return { ok: false, code: 'OVERLAP' };
71         for (const [nx, ny] of neigh8(x, y))
72           if (taken.has(k(nx, ny))) return { ok: false, code: 'TOUCH' };
73       }
74     }
75   }
76   return { ok: true, code: '' };
77 }

```

Рисунок 4 – Основной алгоритм серверной валидации расстановки кораблей

3.3 Централизованная обработка выстрелов и синхронизация результатов

Обработка каждого выстрела полностью сосредоточена на сервере в обработчике события `make_turn`. Сервер последовательно проверяет очередность хода, корректность координат и отсутствие повторного выстрела в уже обстрелянную клетку (с помощью множества `shots` текущего игрока). Далее определяется результат: промах, попадание или потопление корабля соперника путём поиска пересечения координаты выстрела с клетками его кораблей.

При попадании сервер обновляет множество hits атакующего и проверяет, уничтожен ли корабль полностью. В случае потопления длина корабля передаётся клиентам для последующей визуализации зоны вокруг потопленного корабля. Обновлённый результат (miss, hit или sink) рассылается обоим игрокам: атакующему — через событие shot_result, а защищающемуся — через turn_made. При промахе ход передаётся сопернику, о чём уведомляются оба участника событием turn_info.

Победа фиксируется сразу после полного уничтожения флота одного из игроков: сервер рассылает событие game_over с идентификатором победителя и удаляет комнату из памяти (Рис.5).

```
369 const me = userId;
370 const opp = opponentOf(game, me);
371 const shotK = k(target.x, target.y);
372 const myShots = game.boards[me].shots;
373 if (myShots.has(shotK)) {
374   socket.emit('turn_rejected', { reason: 'CELL_ALREADY_TARGETED' });
375   return;
376 }
377 myShots.add(shotK);
378
379 const oppShips = game.boards[opp].ships;
380 let result = 'miss';
381 let sunkLen = null;
382
383 const hitShip = oppShips?.find((ship) =>
384   ship.cells.some((c) => c.x === target.x && c.y === target.y)
385 );
386 if (hitShip) {
387   result = 'hit';
388   const hits = game.boards[me].hits;
389   hits.add(shotK);
390   if (hitShip.cells.every((c) => hits.has(k(c.x, c.y)))) {
391     result = 'sink';
392     sunkLen = hitShip.len;
393   }
394 }
395
396 socket.emit('shot_result', { target, result, shipLen: sunkLen });
397 socket.to(room).emit('turn_made', {
398   from: me,
399   target,
400   result,
401   shipLen: sunkLen,
402 });
403
404 const hits = game.boards[me].hits;
```

Рисунок 5 – Алгоритм обработки выстрела на сервере

3.4 Обработка отключений и поддержание согласованности

Механизм устойчивости к разрывам соединения реализован в обработчике события disconnect. При отключении сокета сервер определяет идентификатор пользователя и удаляет его из всех комнат, в которых он участвовал. Оставшемуся игроку отправляется системное сообщение об уходе соперника. Если комната становится пустой, она полностью удаляется из объекта rooms, освобождая ресурсы (Рис.6).

Благодаря хранению состояния исключительно на сервере и использованию идентификаторов пользователей (а не socket.id)

обеспечивается возможность корректного переподключения: при повторном соединении с валидным JWT-токеном игрок может заново присоединиться к той же комнате, а сервер восстановит актуальное состояние.

```
socket.on('disconnect', () => {
  const u = socket.data.user;
  app.log.info(`✗ disconnected socket=${socket.id} user=${u?.id}/${u?.nickname}`);

  if (!u) return;
  const uid = u.id;

  for (const [room, game] of Object.entries(rooms)) {
    if (!game.players) continue;
    const before = game.players.length;
    game.players = game.players.filter((id) => id !== uid);
    delete game.boards[uid];

    if (before !== game.players.length) {
      app.io.to(room).emit('system', {
        room,
        text: `Игрок ${u.nickname} отключился`,
      });
    }

    if (game.players.length === 0) {
      delete rooms[room];
    }
  }
});
});
});
```

Рисунок 6 – Обработка отключения клиента

4 Результаты и обсуждение

4.1 Проверка работоспособности механизма валидации и синхронизации

Разработанный механизм был протестирован в условиях реальной эксплуатации браузерной системы «Морской бой». Тестирование включало последовательную проверку всех этапов игрового процесса: регистрацию и аутентификацию пользователей, присоединение к комнате, расстановку кораблей, выполнение ходов и обработку завершения партии (Рис.7).

Серверная валидация расстановки кораблей продемонстрировала высокую эффективность: любые попытки передать некорректные данные (пересечения, касания по диагонали, выход за границы поля или неправильный набор длин) немедленно отклонялись с передачей кода ошибки клиенту. Даже при искусственном формировании запросов через инструменты разработчика браузера сервер сохранял целостность правил, что подтверждает надёжность защиты от манипуляций.

Синхронизация состояния оказалась полностью согласованной:

результаты выстрелов, смена очередности хода и уведомления о потоплении кораблей отображались одновременно у обоих участников с задержкой менее 100 мс в локальной сети. Зоны вокруг потопленных кораблей корректно отмечались на полях игроков благодаря передаче длины потопленного корабля и последующей клиентской визуализации.

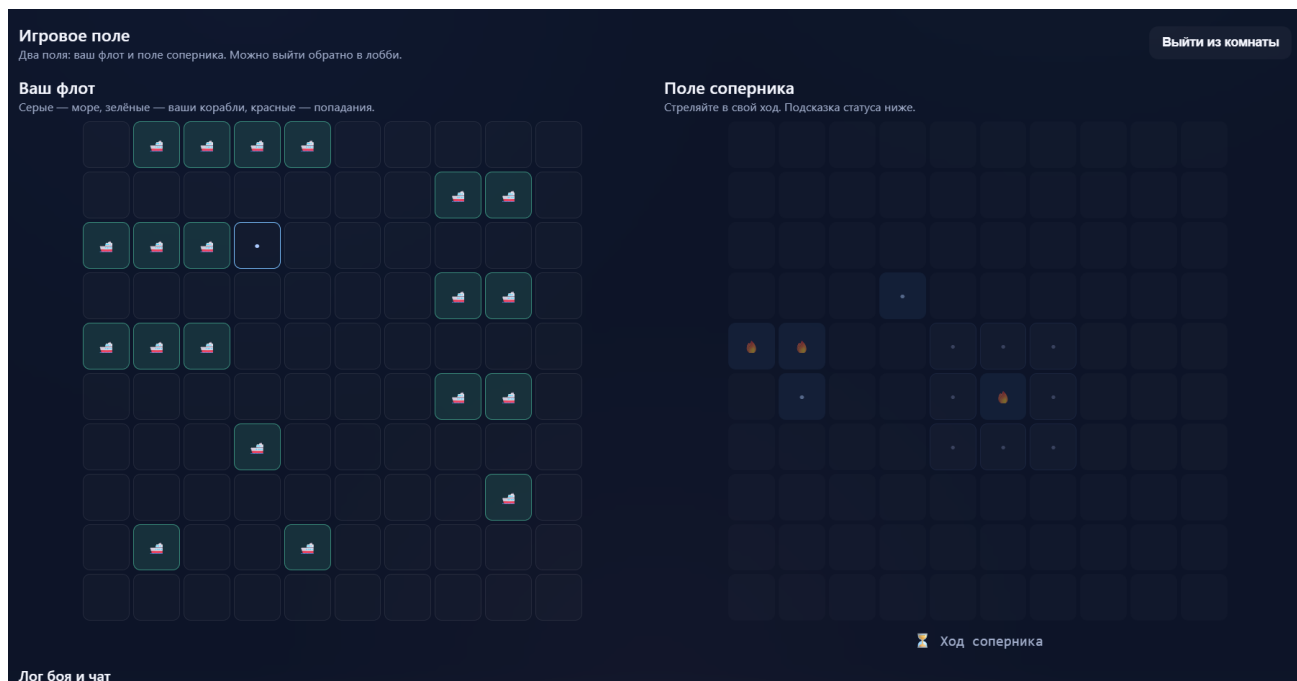


Рисунок 7 – Интерфейс игры во время боя: синхронизированные поля

Особое внимание уделялось расстановке флота. Клиентская часть мгновенно подсвечивала ошибки – касания кораблей, пересечения или выход за границы поля, что делало процесс интуитивным. Серверная валидация дополнительно гарантировала честность: даже если каким-то образом отправить некорректную расстановку, сервер её отклонял. После успешной подготовки обоих игроков игра запускалась автоматически, определялся первый ходящий (Рис. 8).

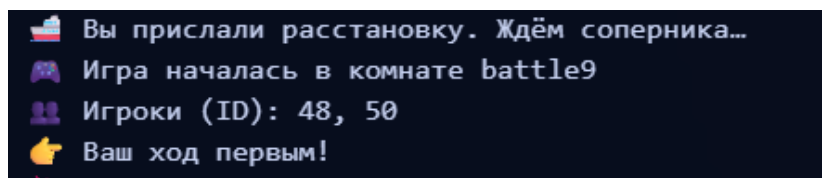


Рисунок 8 – Определение хода для первого игрока

4.2 Обсуждение применимости и ограничений механизма

Предложенный механизм валидации и синхронизации доказал свою эффективность в рамках клиент-серверной браузерной игры и может быть легко адаптирован для других многопользовательских приложений реального времени: чатов, совместных редакторов, простых стратегий или обучающих симуляторов. Централизованный подход с сервером как единственным

арбитром обеспечивает высокую степень справедливости и защиты от читерства, что особенно важно в конкурентных системах.

Основным ограничением является зависимость от постоянного соединения и наличие центрального сервера, что делает систему уязвимой к его отказу. В текущей реализации не рассматривается горизонтальное масштабирование (распределение комнат между несколькими серверами) и поддержка более двух игроков в одной сессии. Эти аспекты могут стать направлениями дальнейшего развития механизма.

Библиографический список

1. Ушаков Ю. А. Модуль сервера видеоконференцсвязи WebRTC на базе видеосервера Janus для системы Moodle : свидетельство о государственной регистрации программы для ЭВМ № 2019666942 Российская Федерация : № 2019663362: заявл. 30.10.2019 : опубл. 17.12.2019. Заявитель Федеральное государственное бюджетное образовательное учреждение высшего образования «Оренбургский государственный университет».
2. Нежелская Л. А., Першина А. А. Процедура оценивания параметра равномерного распределения длительности непродлевающегося случайного мертвого времени в рекуррентном обобщенном асинхронном потоке событий в особом случае // Вестник Томского государственного университета. Управление, вычислительная техника и информатика. 2021. № 54. С. 65–73.
3. Дамирбек К. Г., Жакыпова Э. А. Обеспечение целостности информации в автоматизированных информационных системах // Современные проблемы механики. 2018. № 34(4). С. 66–71.
4. Романов А. А., Волчек Д.Г., Муромцев Д.И. Семантические технологии в массовых открытых онлайн-курсах: экстракция, обогащение и связывание данных электронных учебных курсов для образовательной платформы Open edX // Информационные технологии. 2018. Т. 24. № 8. С. 553–560.
5. Колесников Н. Д., Щербина С.Ю. Особенности применения средств криптографической защиты информации для сетевого трафика многопользовательских игр // Инновации. Наука. Образование. 2021. № 35. С. 520-525.