A

Project Report

On

"A Distributed Application for Interactive Multiplayer Games"

In partial fulfilment of requirements for the degree of

Bachelor of Year Engineering

In

Savitribai Phule Pune University, Pune

Information Technology

SUBMITTED BY:

B511042	**Mayank Jaiswal**
B511002	**Ashish Kumar**
B511058	**Vishwajit Kharose**
B511010	**Manish Bharambe**

Under the guidance of

Prof. Nilam S. Honmane

ZEAL EDUCATION SOCIETY'S

ZEAL COLLEGE OF ENGINEERING AND RESEARCH

NARHE | PUNE -41 | INDIA

Academic Year 2024-2025

CERTIFICATE

This is to certify that project report entitled

"A Distributed Application for Interactive Multiplayer Games"

SUBMITTED BY:

B511042	**Mayank Jaiswal**
B511002	**Ashish Kumar**
B511058	**Vishwajit Kharose**
B511010	**Manish Bharambe**

is a bonafide work carried out by them under the supervision of Dr. Praveen Blessington and it is approved for the partial fulfilment of the requirement of Savitribai Phule Pune University for the award of degree of Bachelor of Engineering (Information Technology).

Prof. Nilam Honmane	**Prof. Balaji Chaugule**	**Dr. A. M. Kate**
Project Guide	Head of Department Information Technology	Principal ZCOER, NARHE

Place: ZCOER, Pune.

Date: 01/ 04/2025

ACKNOWLEDGEMENT

I would like to express my special thanks of gratitude to our Principal Dr. A.M. Kate Sir and HOD of INFORMATION TECHNOLOGY Department, Prof. Balaji Chaugule Sir of ZEAL COLLEGE OF ENGINEERING AND RESEARCH, NARHE for his kind support in completing my Project.

I would also like to extend my gratitude to the Prof. Nilam Honmane, faculty teacher, for sustained co-operation, interest and encouragement throughout this project work.

B511042 **Mayank Jaiswal**

B511002 **Ashish Kumar**

B511058 **Vishwajit Kharose**

B511010 **Manish Bharambe**

ABSTRACT

Colyseus distributes dynamic game-play state and computation while adhering to tight latency constraints and maintaining scalable communication costs. Two key architectural decisions form the basis of our design: first, a single copy consistency model for game objects allows low-latency read/writes at the cost of weaker consistency, which is tolerated by most multiplayer games. Second, we utilize locality and predictability in the game workload to speculatively and quickly pre-fetch objects needed for performing game logic computation. We have implemented Colyseus and demonstrated its practicality by modifying a popular first person shooter (FPS) game called Quake II to use Colyseus for distributing game state across multiple servers or peers. While current single server implementations can support many tens of players, our playable-prototype shows that Colyseus is easily able to support low-latency game-play for hundreds of participants. In addition, our results show that per-node bandwidth requirements for Colyseus are an order of magnitude lower than traditional client-server or broadcast architectures in two different deployment scenarios.

1 Introduction

Networked games are rapidly evolving from small 4-8 person, one-time play games to large-scale games involving thousands of participants [4] and persistent game worlds. However, like most Internet applications, current networked games are centralized. Players send control messages to a central server and the server sends (relevant) state updates to all other active players. This design suffers from the well known robustness and scalability problems of single server designs. For example, complex game-play and AI computation prevent even well provisioned servers from supporting more than several tens of players for first person shooter (FPS) games. Further, client-server game designs often force players to rely on infrastructure provided by the game manufacturers. These infrastructures are sometimes not well provisioned or long-lived; thus, they either provide poor performance or prevent users from playing their game long after their purchase.

A distributed design can potentially address the above shortcomings. However, architecting distributed applications is difficult. The most fundamental challenge in distributing an application is in partitioning the application's state (e.g., the game world state) and execution (e.g., the logic to simulate player and game AI actions) among the participating nodes. Distributing a networked game is made even more difficult by the performance demands of the real-time game-play. In addition, since the game-play of an individual player translates to updates to the shared state of the game application, there is much more write traffic and write- sharing than most distributed applications.

Fortunately, there are two fundamental properties of such games that we can take advantage of in addressing these challenges. First, games tolerate weak consistency in the application state. For example, even current client-server implementations minimize interactive response time by presenting a weakly consistent view of the game world to players. Second, game-play is usually governed by a strict set of rules that make the reads/writes of the shared state highly predictable. For example, most reads and writes of a player occur upon objects which are physically close to the player.

The challenge, then, is to arrive at a scalable and efficient state and logic partitioning that enables reasonably consistent game-play without incurring much latency. This paper presents the design, implementation and evaluation of Colyseus, a novel distributed architecture for interactive multiplayer games designed to achieve the above goals. Our design is based on two key architectural decisions: First, distributed objects in Colyseus follow a single-copy consistency model – i.e., all writes to an object are serialized through exactly one node in the system. This allows low-latency reads and writes at the cost of weak consistency. This mirrors the consistency model of a client-server architecture, albeit on a per-object basis. Secondly, Colyseus utilizes locality and predictability in the movement patterns of players to speculatively pre-fetch objects needed for driving game logic computation. Efficiently locating objects to pre-fetch is achieved using a scalable distributed range-query lookup service.

Our design enables games to efficiently use widely distributed servers to support a large community of users. We have integrated our implementation of Colyseus with Quake II [8], a popular server-based FPS game. This concrete case study illustrates the simplicity of using our architecture to distribute existing game implementations. In our distributed Quake II

implementation, unmodified Quake II clients can connect to any instance of the Colyseus-based distributed Quake II server. As a result, the system can be run as a peer-to-peer application (with every client running a copy of the distributed server) or as a distributed infrastructure (with clients connecting to the closest 1 distributed server). Our measurement of this prototype on an Emulab testbed indicates that Colyseus scales to hundreds of players by distributing game traffic well across the participating nodes. In addition, we show that Colyseus is able to provide each server/player with low latency and a consistent view of the game world.

2 Background and Motivation

In this section, we survey the requirements of online multiplayer games in detail and demonstrate the fundamental limitations of existing client-server implementations. In addition, we provide evidence that resources exist for distributed deployments of multiplayer games. This motivates our exploration of distributed architectures for such games. 2.1 Contemporary Game Design To determine the requirements of multiplayer games, we studied the source code of a few popular and publicly released online game engines, such as Quake II [8] and the Torque Networking Library [9]. In these games, each player (game participant) controls one or a more avatars (player's representative in the game) in a relatively large game world (a two or three dimensional space where the game is played). This description applies to a large number of popular genres, including first person shooters (FPSs) (such as Quake, Unreal Tournament, and Counter Strike), role playing games (RPGs) (such as Everquest, Final Fantasy Online, and World of Warcraft), and others. In addition, this model is similar to that of military simulators and virtual collaborative environments [22, 23].

Almost all commercial games of this type are based on a client-server architecture, where a single server maintains the state of the game world.1 The game state is typically structured as a collection of objects, each of which represents a part of the game world, such as the game world's terrain, buildings, walls, players' avatars, computer controlled characters, doors, items (e.g., healthpacks) and projectiles. Each object is associated with a piece of code called a think function that determines the actions of the object. Typical think functions examine and update the state of both the associated object and other objects in the game. For example, a monster may determine his move by examining the surrounding terrain and the position of nearby players. The game state and execution is composed from the combination of these objects and associated think functions.

The server implements a discrete event loop in which it invokes the think function for each object in the game and sends out the new view (also called a frame in game parlance) of the game world state to each player. In FPS games, this loop is executed 10 to 20 times a second; this frequency (called the frame-rate) is generally lower in other genres.

Distributed Rock-Paper-Scissors

The aim of this project is to develop a **distributed Rock-Paper-Scissors game** using a **client-server setup** powered by the **Java RMI** protocol. In this model, the **server** acts as the **moderator** overseeing the game, maintaining the state, and ensuring that the game progresses correctly. The **client** functions as the player, sending their moves to the server and receiving responses related to the game state and results. The server also ensures synchronization between two players (clients) and handles game transitions based on the rules of Rock-Paper-Scissors.

Conduct

Server-Client Relationship

- The **server** will remain active at all times, ready to facilitate matches between players.
- Players will interact solely with the **client** interface, which prompts the user to begin a game. Upon choosing to play, the **client** will connect to the server, prompting the server to add the player to the game queue.
- Only **two players** can be connected to the server at any given time. If a third player attempts to join, the server will respond with a **"Wait"** or **"Busy"** message, indicating that the game is currently full.

Game Play

- When the first player connects, the server will wait for the second player to join. Once the second player connects, the server randomly assigns **Player 1** and **Player 2**. Player 1 will play **Rock**, **Paper**, or **Scissors** first, while Player 2 will follow.
- Upon Player 1's move, the **server** will send the current state of the game (i.e., the move made by Player 1) to Player 2, who will then make their move. Both players will be informed when it's their turn.
- The server will evaluate the **round result** based on the rules:
  - **Rock** beats **Scissors**
  - **Scissors** beats **Paper**
  - **Paper** beats **Rock**
- Once a move is made, the **server** updates the game state and notifies the other player of the result. The **scoreboard** (if used) will also update after each round.

Game Outcome

- When the game reaches an outcome:
  - The player who wins will be informed with a **"Win"** message.
  - The player who loses will receive a **"Lose"** message.
  - If a **draw** occurs (i.e., both players select the same move), both players will get a **"Draw"** message.
- After the game concludes, both players will be prompted with a **"Play another game?"** message. If both players agree, a new game starts with fresh settings, otherwise, both clients disconnect from the server and the session ends.

Game Parallelization

- The **server** must be capable of running **multiple games simultaneously**.
- When players connect, the server should initiate a **game pair** between the first two players, followed by starting a new game with each subsequent pair of players. This allows the server to manage multiple games at once without interference between the matches.
- For example, if **X players** connect, where **X is even**, **X/2 games** should be running in parallel, ensuring each game proceeds independently. For instance, if **Player 1 (P1)** and **Player 2 (P2)** start their game, and **Player 3 (P3)** and **Player 4 (P4)** connect afterward, the server will initiate a fresh game between **P3** and **P4** while **P1** and **P2** continue theirs.

Open Terminal-1 Run :

```
javac *.java

rmiregistry 5000
```

Open Terminal-2 Run :

```
java MyServer
```

Open Terminals-3 Run :

```
java MyClient
```

Open Terminals-3 Run :

```
java MyClient
```

MyClient.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.util.List;

public class MyClient extends JFrame {
    private RPSContract rps;
    private String playerName;
    private JButton rockButton, paperButton, scissorsButton;
    private JLabel statusLabel, player1ScoreLabel, player2ScoreLabel,
timerLabel;
    private JTextArea historyArea;
    private Timer gameTimer;
    private int remainingTime = 15;

    public MyClient(String playerName) throws Exception {
        this.playerName = playerName;
        rps = (RPSContract) Naming.lookup("rpsGame");

        setTitle("Rock-Paper-Scissors - " + playerName);
        setLayout(new BorderLayout());

        // Buttons
        rockButton = new JButton("🪨 Rock (R)");
        paperButton = new JButton("📄 Paper (P)");
        scissorsButton = new JButton("✂️ Scissors (S)");

        JPanel buttonPanel = new JPanel();
        buttonPanel.add(rockButton);
        buttonPanel.add(paperButton);
        buttonPanel.add(scissorsButton);

        rockButton.addActionListener(e -> makeMove("R"));
        paperButton.addActionListener(e -> makeMove("P"));
        scissorsButton.addActionListener(e -> makeMove("S"));

        // Status + Scores + Timer
        statusLabel = new JLabel("Waiting for another player...");
        player1ScoreLabel = new JLabel("Player1: 0");
        player2ScoreLabel = new JLabel("Player2: 0");
        timerLabel = new JLabel("⏳ Time: 15s");

        JPanel topPanel = new JPanel(new GridLayout(2, 2));
        topPanel.add(player1ScoreLabel);
```

```
        topPanel.add(player2ScoreLabel);
        topPanel.add(timerLabel);
        topPanel.add(statusLabel);

        // Game History
        historyArea = new JTextArea(10, 30);
        historyArea.setEditable(false);
        JScrollPane historyScroll = new JScrollPane(historyArea);
        historyScroll.setBorder(BorderFactory.createTitledBorder("📜 Game
History"));

        add(topPanel, BorderLayout.NORTH);
        add(buttonPanel, BorderLayout.SOUTH);
        add(historyScroll, BorderLayout.CENTER);

        setSize(500, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        // Register player
        rps.registerPlayer(playerName);

        // Start update loop
        startUpdater();
        startTimer();
    }

    private void makeMove(String move) {
        try {
            String result = rps.makeMove(playerName, move);
            statusLabel.setText("📢 " + result);
            remainingTime = 15; // Reset timer
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    private void startUpdater() {
        new Thread(() -> {
            while (true) {
                try {
                    Thread.sleep(1000);
                    updateGameState();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
```

```
    private void updateGameState() throws RemoteException {
        int player1Score = rps.getPlayer1Score();
        int player2Score = rps.getPlayer2Score();
        List<String> history = rps.getGameHistory();

        player1ScoreLabel.setText("Player1: " + player1Score);
        player2ScoreLabel.setText("Player2: " + player2Score);

        StringBuilder historyText = new StringBuilder();
        for (String h : history) {
            historyText.append(h).append("\n");
        }
        historyArea.setText(historyText.toString());

        if (rps.isGameOver()) {
            String winner = (player1Score > player2Score) ? "Player1" :
"Player2";
            statusLabel.setText("🎉 Game Over! Winner: " + winner);
            rockButton.setEnabled(false);
            paperButton.setEnabled(false);
            scissorsButton.setEnabled(false);
            gameTimer.stop();
        }
    }

    private void startTimer() {
        gameTimer = new Timer(1000, e -> {
            remainingTime--;
            timerLabel.setText("⏳ Time: " + remainingTime + "s");
            if (remainingTime <= 0) {
                statusLabel.setText("⏳ Time's up! Waiting for
opponent...");
                remainingTime = 15;
            }
        });
        gameTimer.start();
    }

    public static void main(String[] args) {
        try {
            String name = JOptionPane.showInputDialog("Enter your player
name:");
            new MyClient(name);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

MyServer.java

```java
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.net.MalformedURLException;

public class MyServer {
    public static void main(String[] args) {
        try {
            // Start RMI registry
            Registry registry = LocateRegistry.createRegistry(1099);

            // Create and bind the remote object
            RPSImpl rps = new RPSImpl();
            Naming.rebind("rpsGame", rps); // This can throw
MalformedURLException
            System.out.println("RPS Server is ready.");
        } catch (RemoteException | MalformedURLException e) {
            e.printStackTrace();
        }
    }
}
```

RPSImpl.java

```java
// RPSImpl.java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.List;

public class RPSImpl extends UnicastRemoteObject implements RPSContract {
    private String player1 = null;
    private String player2 = null;
    private String player1Move = "";
    private String player2Move = "";
    private int player1Score = 0;
    private int player2Score = 0;
    private final List<String> gameHistory = new ArrayList<>();
    private boolean gameOver = false;

    public RPSImpl() throws RemoteException {}

    @Override
```

```
    public synchronized boolean registerPlayer(String name) throws
RemoteException {
        if (player1 == null) {
            player1 = name;
            return true;
        } else if (player2 == null) {
            player2 = name;
            return true;
        }
        return false;
    }

    @Override
    public synchronized String makeMove(String playerName, String move)
throws RemoteException {
        if (gameOver) return "Game Over.";

        if (playerName.equals(player1)) {
            player1Move = move;
        } else if (playerName.equals(player2)) {
            player2Move = move;
        }

        if (!player1Move.isEmpty() && !player2Move.isEmpty()) {
            String roundResult = determineRoundWinner();
            gameHistory.add(player1 + " played " +
moveFullName(player1Move) +
                            " | " + player2 + " played " +
moveFullName(player2Move) +
                            " => " + roundResult);

            player1Move = "";
            player2Move = "";

            if (player1Score == 5 || player2Score == 5) {
                gameOver = true;
            }

            return roundResult;
        }

        return "Move registered. Waiting for other player...";
    }

    private String determineRoundWinner() {
        if (player1Move.equals(player2Move)) {
            return "It's a draw!";
        }

        boolean p1Wins = (player1Move.equals("R") &&
player2Move.equals("S")) ||
```

```
                        (player1Move.equals("P") &&
player2Move.equals("R")) ||
                        (player1Move.equals("S") &&
player2Move.equals("P"));

        if (p1Wins) {
            player1Score++;
            return player1 + " wins the round!";
        } else {
            player2Score++;
            return player2 + " wins the round!";
        }
    }

    private String moveFullName(String move) {
        switch (move) {
            case "R": return "Rock";
            case "P": return "Paper";
            case "S": return "Scissors";
            default: return "Unknown";
        }
    }

    @Override
    public int getPlayer1Score() throws RemoteException {
        return player1Score;
    }

    @Override
    public int getPlayer2Score() throws RemoteException {
        return player2Score;
    }

    @Override
    public List<String> getGameHistory() throws RemoteException {
        return gameHistory;
    }

    @Override
    public boolean isGameOver() throws RemoteException {
        return gameOver;
    }

    @Override
    public synchronized void resetGame() throws RemoteException {
        player1 = null;
        player2 = null;
        player1Move = "";
        player2Move = "";
        player1Score = 0;
        player2Score = 0;
```

```
        gameHistory.clear();
        gameOver = false;
    }

    @Override
    public String getCurrentTurn() throws RemoteException {
        return (player1Move.isEmpty() && player2Move.isEmpty()) ? player1
: player2;
    }
}
```

RPSContract.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

public interface RPSContract extends Remote {
    boolean registerPlayer(String playerName) throws RemoteException;
    String getCurrentTurn() throws RemoteException;
    String makeMove(String playerName, String move) throws
RemoteException;
    int getPlayer1Score() throws RemoteException;
    int getPlayer2Score() throws RemoteException;
    List<String> getGameHistory() throws RemoteException;
    boolean isGameOver() throws RemoteException;
    void resetGame() throws RemoteException;
}
```

Outputs-

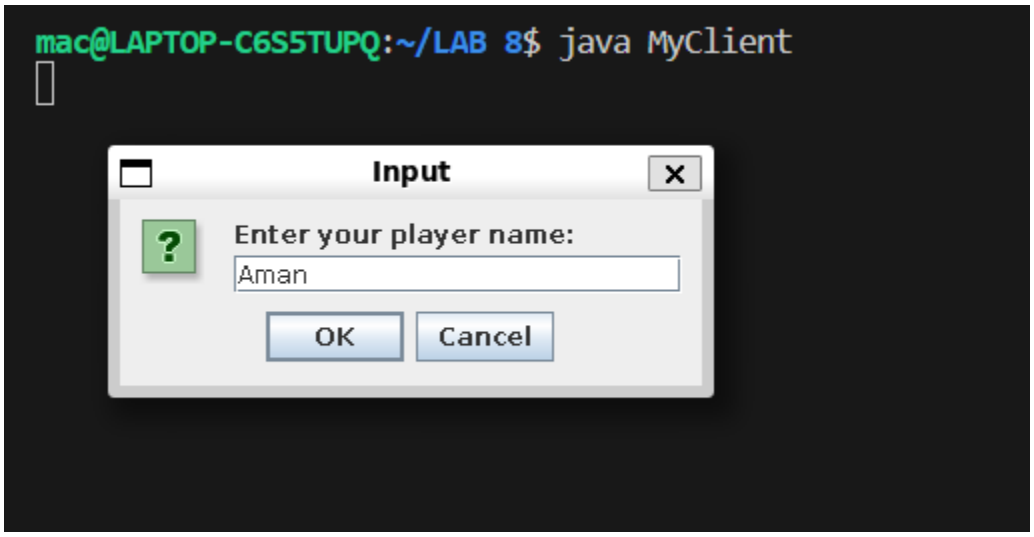
Terminal-1

```
mac@LAPTOP-C6S5TUPQ:~/LAB 8$ javac *.java
mac@LAPTOP-C6S5TUPQ:~/LAB 8$ rmiregistry 5000
WARNING: A terminally deprecated method in java.lang.System has been called
WARNING: System::setSecurityManager has been called by sun.rmi.registry.RegistryImpl
WARNING: Please consider reporting this to the maintainers of sun.rmi.registry.RegistryImpl
WARNING: System::setSecurityManager will be removed in a future release
```

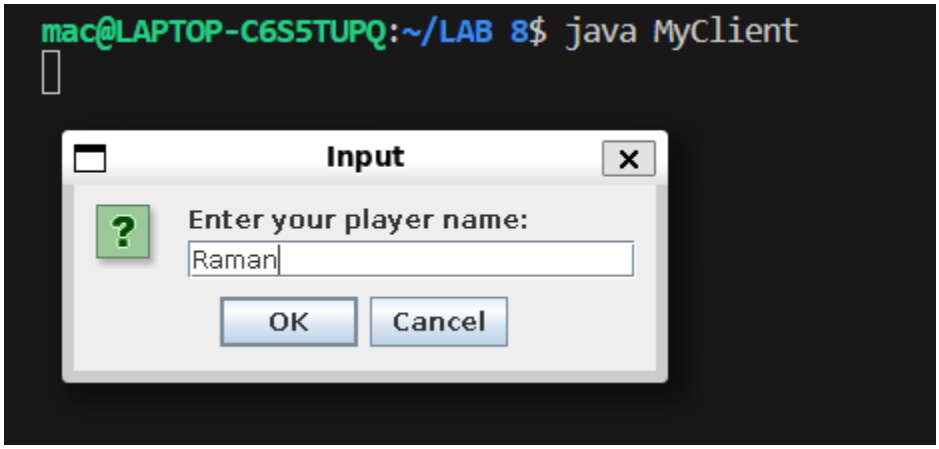
Terminal-2

```
mac@LAPTOP-C6S5TUPQ:~/LAB 8$ java MyServer
RPS Server is ready.
```

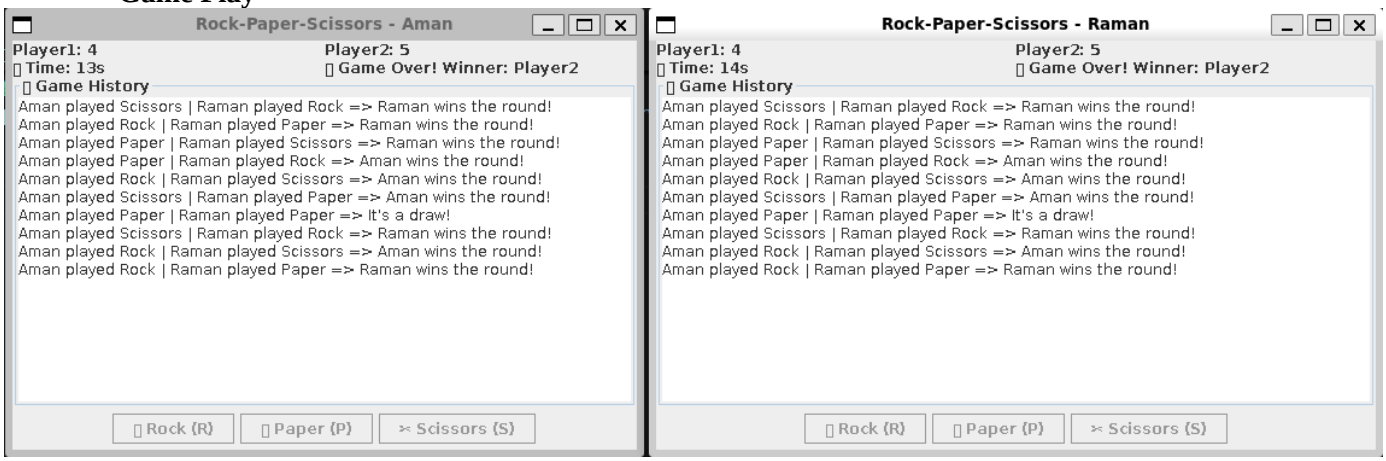
Terminal-3



Terminal-4



Game Play-



2.2 Client-Server Scaling Properties

The most significant drawback of this client-server game architecture is its reliance on a single server to maintain the game. The server can become a computation and communication bottleneck. To quantify these bottlenecks, we describe the general scaling properties of games and present the scaling measurements from a typical client-server FPS game, Quake II.

Scalability Analysis:

The three game parameters that most impact network performance are: (1) the number of objects in play (NumObjs), (2) the average size of those objects (ObjSize), and (3) the game's frame-rate (UpdateFreq). In Quake II, if we only consider objects representing players (which tend to dominate the game update traffic since they are the most dynamic), NumObjs ranges from 8 to 64, ObjSize is ∼200 bytes, and UpdateFreq is 10 updates per second. A na¨ıve server implementation which simply broadcasts the updates of all objects to all game clients (NumClients) would incur an outbound bandwidth cost of NumClients × NumObjs × ObjSize × UpdateFreq, or 1-66Mbps in the case of Quake II games between 8 and 64 players.

Two common optimizations used by games are area-of-interest filtering and delta-encoding. 2 Individual players typically only interact with or see a small portion of the game world at any one time. Servers need only update clients about objects in this area-of-interest, thus, reducing the number of objects transferred from the total set of objects (NumObjs) to the number of objects in this area (NumAoiObjs). If we only consider players, NumAoiObjs is typically about 4 for most Quake II games. Additionally, the set of objects and their state change little from one update to the next. Therefore, most servers simply encode the difference (delta) between updates, thus, reducing the number of bytes for each object transferred from the object's size (ObjSize) to the average delta size (AvgUpdateSize), which is about 24 bytes in Quake II. Thus, the optimized outbound server bandwidth cost would be NumClients × NumAoiObjs × AvgUpdateSize × UpdateFreq, or about 62-492kbps in the case of Quake II for 8-64 players.

Empirical Scaling Behavior:

Figure 1 shows the performance of a Quake II server running on a Pentium-III 1GHz machine with 512 RAM. We vary the number of clients on this server from 5 to 600. Each client is simulated using a server-side AI bot. Quake II implements area-of-interest filtering, delta-encoding and does not rate-limit clients. We run the game for a 10 minute duration at 10 frames per second. As the computational load on a server increases, the server may require more than 1 frametime of computation to service all clients. Hence, it may not be able to sustain the target frame rate. Figure 1(a) shows the mean number of frames per second actually computed by the server, while Figure 1(b) shows the bandwidth consumed at the server for sending updates to clients. We note several points: first, as the number of players increases, area-of-interest filtering computation becomes a bottleneck 3 and the frame rate drops. Second, Figure 1(b) shows that, as the number of players increases, the bandwidth-demand at the server increases more than linearly, since as the number of players increases, players interaction increases (more missiles are shot, for example). Thus, NumAoiObjs increases along with NumClients resulting in almost quadratic increase in bandwidth. Finally, we note that for large number of players computational load becomes the bottleneck. The huge reduction in frame-rate offsets any

increase in bandwidth that might be expected due to an increase in the number of clients. Therefore, we actually see the bandwidth requirements dropping. Although the absolute limits shown can be raised by a factor of 2 or 3 by employing more powerful servers, clearly a centralized server quickly becomes a bottleneck.

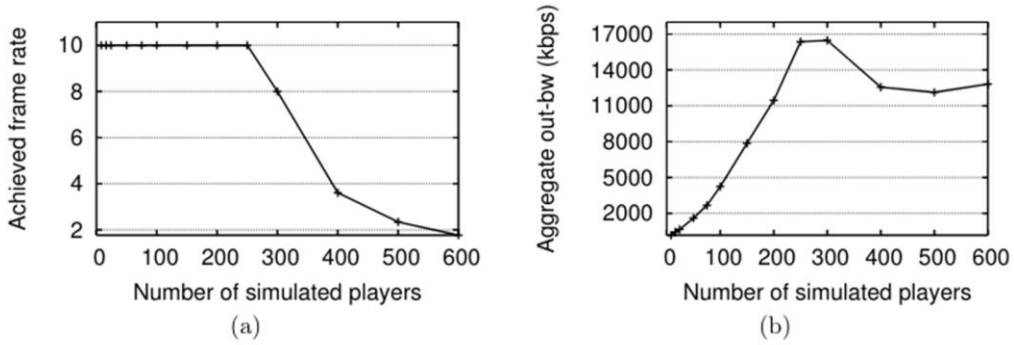


Figure 1: Computational and network load scaling behavior at the server end of client-server system.

Related Work

In this section, we briefly comment on the designs adopted by current games, as well as previous research architectures for distributed games. Some games including MiMaze [18], Halo [5], and most Real Time Strategy (RTS) games [1], have adopted a parallel simulation architecture, where each player in the game simulates the entire game world. Thus, all objects within this world are replicated everywhere and kept consistent using lock-step synchronization. The obvious disadvantages of this architecture are its requirement of broadcasting updates to every player, resulting in quadratic bandwidth scaling behavior, and its need for synchronization, limiting response time to the speed of the slowest client and the latency between the players. These deficiencies are tolerated in RTS games because individual games rarely involve more than 8 players. Second-Life [11] and Butterfly.net [3] perform interest filtering by partitioning the game world into disjoint regions or cells. The basic idea is to describe an object's area-of-interest by the cell in which it currently resides and/or a subset of cells nearby.

SimMUD [13] is similar to these systems but implements area-of-interest filtering by assigning regions to keys in a Distributed Hash Table allowing it to operate in a distributed environment. Zou, et al. [29] theoretically compared cell-centric approaches with entity-centric approaches, like Colyseus. The cell-based approach requires that the region-granularity must be specified in advance; in a distributed environment this would either ensure frequent migration of objects between cells if they are small, or force particular nodes to take on a large number of objects if they host large, popular regions. Colyseus differs in that it allows areas-of-interest to be specified in terms of arbitrary bounding boxes. In addition, it does not restrict the placement of

objects in the system. These factors allow for efficient communication as well as provide more flexibility for balancing load. Furthermore, while the above approaches share some commonalities with our design, we believe we are the first to demonstrate the feasibility of implementing a real-world game on a distributed architecture that is not designed for a centralized cluster (like Second-Life and Butterfly.net).

Colyseus is able to support FPS games which have much tighter latency constraints as compared 20 to RPGs (which were targeted by SimMUD, for example.) Several architectures proposed for Distributed Virtual Reality (VR) environments and distributed simulation (notably, DIVE[17], MASSIVE [19], and High Level Architecture (HLA) [22]) have similar goals as Colyseus but focus on different design aspects; DIVE and MASSIVE focus on sharing audio and video streams between participants while HLA is designed for military simulations. None address the specific needs of modern multiplayer games and, to our knowledge, none of them have been demonstrated to scale to large numbers of participants in practice; DIVE and HLA, for example, originally assumed wide-scale deployment of IP-Multicast.

Summary and Future Work

In this paper, we have described the design, implementation and evaluation of Colyseus, a distributed architecture for online multiplayer games. Colyseus takes advantage of a game's ability to tolerate inconsistent state in its partitioning of state across system nodes. It also takes advantage of game software's predictable read/write workloads to aggressively pre-fetch objects to a system node. Our adaptation of a commercial game (Quake II) to use Colyseus showed that it provides an effective interface for distributing existing game designs. Our evaluation of the Quake II implementation showed that Colyseus is able to: 1) effectively use over a hundred server nodes, 2) support hundreds of game participants and 3) support real-time game-play. Nonetheless, we should note that the current Colyseus design does not address a few important problems in a distributed setting. Our current implementation does not tolerate node departures, but we believe that the presence of a large number of object replicas in our current design can be leveraged to provide continuity after node failures. Furthermore, note that the Mercury location component can withstand moderate node churn rates. In some deployment scenarios, a distributed architecture also raises issues about cheating. To address these problems, we believe we can leverage Colyseus' flexibility in object placement. For example, by carefully selecting the owners of primary objects, we may be able to limit the damage malicious players or nodes can inflict on others. We plan to address these challenges in future work.

References

[1] Age of Empires. http://www.microsoft.com/games/empires/.

[2] Big World. http://www.microforte.com/.

[3] Butterfly.net. http://www.butterfly.net/.

[4] Everquest Online. http://www.everquest.com.

[5] Halo. http://www.xbox.com/en-US/halo/.

[6] King peer-to-peer measurements data set. http://www.pdos.lcs.mit.edu/p2psim/kingdata.

[7] PlanetSide. http://planetside.station.sony.com.

[8] QuakeII Game Engine v3.20. ftp://ftp.idsoftware.com/idstuff/quake2.

[9] Torque Networking Library. http://www.garagegames.com.

[10] Zona. http://www.zona.net/.

[11] Enabling Player-Created Online Worlds with Grid Computing and Streaming. http://www.gamasutra.com/resource_guide/20030916/rosedale_01.shtml, 2003.

[12] Bharambe, A., Agrawal, M., and Seshan, S. Mercury: Supporting scalable multi-attribute range queries. In ACM SIGCOMM 2004.

[13] Bjorn Knutsson and Honghui Lu and Wei Xu and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In Proceedings of the IEEE INFOCOM 2004.

[14] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative

storage with CFS. In Proceedings of the 18th Symposium on Operating System Principles (Oct. 2001).

[15] Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, M. F., and Morris, R. Designing a

DHT for low latency and high throughput. In Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04) (March 2004).

[16] Druschel, P., and Rowstron, A. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In Proceedings of the 18th Symposium on Operating System Principles (Oct. 2001).

[17] Frecon, ´ E., and Stenius, M. DIVE: A scaleable network architecture for distributed virtual environments. Distributed Systems Engineering Journal 5, 3 (1998), 91–100.

[18] Gautier, L., and Diot, C. MiMaze, A Multiuser Game on the Internet. Tech. Rep. RR- 3248, INRIA, France, Sept. 1997.

[19] Greenhalgh, C., and Benford, S. Massive: a distributed virtual reality system incorporating spatial trading. In ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95) (1995), IEEE Computer Society, p. 27.

[20] Gummadi, K. P., et. al. The Impact of DHT Routing Geometry on Resilience and Proximity. In Proceedings of the ACM SIGCOMM '03 (Aug. 2003).

[21] Guttman, A. R-trees: a dynamic index structure for spatial searching. In SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data (1984), ACM Press, pp. 47–57.

[22] Institute of Electrical and Electronics Engineers. High Level Architecture. http://www. dmso.mil/public/transition/hla/index.html.

[23] Institute of Electrical and Electronics Engineers. Standard for Information Technology, Protocols for Distributed Interactive Simulation. Tech. rep., Mar. 1993.

[24] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the emerald system. ACM Trans. Comput. Syst. 6, 1 (1988), 109–133.

[25] Pantel, L., and Wolf, L. C. On the suitability of dead reckoning schemes for games. In NETGAMES '02: Proceedings of the 1st workshop on Network and system support for games (2002), ACM Press, pp. 79–84.

[26] Shaikh, A., Sahu, S., Rosu, M., Shea, M., and Saha, D. Implementation of a Service Platform for Online Games. In Proc. of NetGames 2004 Workshop (Aug. 2004).

[27] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. Chord: A scalable

peer-to-peer lookup service for internet applications. In Proceedings of the SIGCOMM '01 Symposium on Communications Architectures and Protocols (2001).

[28] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M.,Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. In Proc. of the Fifth Symposium on Operating Systems Design and Implementation (Boston, MA, Dec. 2002), USENIX Association, pp. 255–270. 22

[29]Zou, L., Ammar, M. H., and Diot, C. An evaluation of grouping techniques for state dissemination in networked multi-user games. In MASCOTS '01: Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01) (2001), IEEE Computer Society, p. 33.

Conclusion:

The distributed architecture for the **Rock-Paper-Scissors** game can benefit from concepts like centralized state management and parallel simulation to optimize performance and scalability. Adapting principles from systems like **Colyseus** can ensure efficient communication, load balancing, and smoother gameplay as the number of concurrent games increases.