Distributed Quiz

Hassan Hayat
Pinkal Patel
Andrés Salgado

Preface

This project comes as a result of wanting to explore the topic of clock synchronization and real-time systems and the application of these topics. Online games are amongst the most pervasive applications of the theory of distributed systems and have very stringent latency requirements. Online games, especially real-time multiplayer games, have extremely strict latency requirements as the slightest perception of latency can ruin the experience for players.

By implementing a distributed real-time quiz, we have chosen a relatively simple game that nevertheless encompasses most of the main challenges faced by real-time online multiplayer games. Furthermore, this project represents an opportunity to explore the multitude of techniques and algorithms that span the field of clock synchronization and distributed computing.

Acknowledgements

We would like to take this opportunity to express our sincere gratitude to everyone who has helped us make this project successful. We also would like to thank the frustrating lag, or bursts of latency, that plague many online multiplayer games for frustrating us to become inspired to make this project.

We are also very thankful to all of the researchers and journals who have laid the theoretical foundations for this project. Without their contributions, we would not have a functioning internet or a functioning cloud computing system that has enable the creation of this project.

Last, but not least, we are deeply grateful to our project instructor and advisor, Prof. Ming-Hwa Wang, for his continued support and encouragement.

Abstract

Clock synchronization is an issue in real-time distributed systems as each independent process tends to keep its own time. This, in turn, implies that most processes eventually go out-of sync. Online multiplayer quizzes are usually played on a turn-based basis. This has the unfortunate consequence of not being very fun. The challenge in quizzes usually comes from strict time requirements. This paper explores the application of clock synchronization to online multiplayer quizzes in order to produce a real-time online multiplayer quiz. The goal is to produce a fun and engaging multiplayer game.

Table of Contents

Introduction

The objective of this project is to explore the topic of Clock Synchronization and demonstrate its utility with a fun and practical application, a quiz game. This project will consider the Berkeley and Cristian algorithms and their usefulness in the context of clock synchronization.

The goal of this game is to reproduce a quiz show in a distributed fashion. The rules of the quiz are as follows: Multiple players connect to a server. The server asks a question to the players and a buzzer appears on each player's display. The first player to press on the buzzer gets to answer the question in a multiple choice fashion. The player earn 10 points per correct response and loses 5 points per incorrect response, in order to discourage players from mindlessly pressing the buzzer.

This project is partly based on the paper "QuizFun: Mobile based quiz game for learning" by Isuru et al. In that paper, the authors implemented a multiplayer quiz in order to "increase students' interactive participation in learning". The goal of this project is to produce a real-time version of their implementation.

Theoretical Basis and Literature Review

The main problem we are trying to solve boils down to the problem of clock synchronization. The quiz is played in real-time and the first player who presses the buzzer gets to answer the question. As such, it is imperative to determine which player pressed the buzzer first. In order for the game to be as fair as possible, this determination has to be made regardless of which client response managed to reach the server. This cannot be relied up as the amount of time it takes for a packet to travel across the internet is not fixed and cannot be predetermined. Therefore, we need to consider the theory behind clock synchronization and the ordering of events across independent processes.

CLOCK SYNCHRONIZATION

Clock synchronization deals with understanding the temporal ordering of events produced by concurrent processes. It is useful for synchronizing senders and receivers of messages, monitoring simultaneous activity, and controlling concurrent access to shared objects. The main goal is to have multiple independent processes somehow be in agreement as to the order of events.

How do we determine if event a happened before event b? Most people would say that event a happened first if it happened at an earlier time than event b. The problem with this definition is that it depends on the existence of a physical clock. Nevertheless, it is very useful to have a time-value that is associated with each event and to be able to compare the values. As such, we consider the concept of “happened-before” without using physical clocks.

HAPPENED BEFORE

We define the relation "→" (happened-before) on the set of events of a system is the smallest relation satisfying the following conditions:

1. If a and b are events in the same process and a comes before b, then a → b.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then a → b.
3. If a → b and b → c, then a → c.

Furthermore, we say that events a and b are concurrent if: a □ b and b □ a

LOGICAL CLOCK

We further define the concept of a logical clock. A logical clock is a way of assigning a number to an event, where the number is thought of as the time at which the event occurred. The concept of a logical clock is very useful as we cannot rely on a single physical clock to synchronize multiple independent processes, each of which may have different clock state at each given time. With logical clocks, we can define the Clock Condition.

CLOCK CONDITION

For any events a, b: if a → b then C(a) < C(b), where C(x) is the clock value associated with event x. Note that we cannot expect the converse to be true or else that would imply that any two concurrent events must occur at the same time, which is not true. The Clock Condition is satisfied if the following two conditions are satisfied:

1. If a and b are events in process P_i, and a comes before b, then $C_i(a) < C_i(b)$
2. If a is the act of sending a message by process P_i and b is the act of receiving that message by process P_j, then $C_i(a) < C_j(b)$

We consider the concept of the Clock Condition because we can place a total ordering on the events in a system of clocks satisfying the Clock Condition. To break ties, we can use any arbitrary ordering of the processes.

ORDERED BEFORE

In order to perform an ordering of events, we define the relation "⇒" (ordered-before) as follows. If a is an event in process P_i and b is an event in process P_j, then a ⇒ b if and only if either.

1. $C_i(a) < C_j(b)$
2. $C_i(a) = C_j(b)$ and $P_i < P_j$

Note that the ordering ⇒ on a system of clocks because the ordering $<$ of processes can be arbitrary and is left to the discretion of the implementation.

of Lamport timestamps. This algorithm forces a resequencing of timestamps to ensure that the relation "→" is properly preserved throughout a system of MN

1. Each process has a clock, which can be a simple counter that is incremented for each event.
2. When a process sends a message, it includes the counter value with the message
3. When a process receives a message, this recipient process updates its counter, if necessary, to the greater of its current counter value and the timestamp in the received message. The counter is then incremented by 1 before the message is considered received.

This algorithm has the main advantage of being very simple to implement and to understand, but it has a flaw. While we know that if a → b then C(a) < C(b), we cannot tell that a → b given C(a) < C(b). Unfortunately, looking at Lamport timestamps, we cannot conclude which pairs of events are causally related and which are not. One proposed solution is the concept of vector clocks.

LAMPORT TIMESTAMPS

A very simple implementation of this concept is with the use of Lamport timestamps. This algorithm forces a resequencing of timestamps to ensure that the relation "→" is properly preserved throughout a system of processes.

1. Each process has a clock, which can be a simple counter that is incremented for each event.
2. When a process sends a message, it includes the counter value with the message
3. When a process receives a message, this recipient process updates its counter, if necessary, to the greater of its current counter value and the timestamp in the received message. The counter is then incremented by 1 before the message is considered received.

This algorithm has the main advantage of being very simple to implement and to understand, but it has a flaw. While we know that if a → b then C(a) < C(b), we cannot tell that a → b given C(a) < C(b). Unfortunately, looking at Lamport timestamps, we cannot conclude which pairs of events are causally related and which are not. One proposed solution is the concept of vector clocks.

VECTOR CLOCKS

A vector clock in a system of N processes is a vector of N integers. Each process maintains its own vector clock (V_i for a process P_i) to timestamp local events. Like Lamport timestamps, vector clocks are sent with each message. The rules for using vector clocks are as follows:

1. The vector is initialized to 0 for all processes: $V_i(j) = 0$ for all i, j
2. Before a process Pi timestamps an event, it increments its element of the vector in its local vector: $V_i(i) = V_i(i) + 1$
3. A message is sent from process P_i with V_i attached to the message
4. When a process P_j receives a vector timestamp t, it compares the two vectors element by element, setting its local vector clock to the higher of the two values: $V_j(i) = \max(V_j(i), t(i))$ for all i

The disadvantage with vector clocks is the greater storage and message payload size, since we have to include an entire vector with each message. An alternative to using vector clocks is to decide on a single source for time and have each process call this time server to obtain the time. Cristian's Algorithm is an algorithm that achieves clock synchronization using a time server.

CRISTIAN'S ALGORITHM

The easiest way to set the time would be to simply issue a remote procedure call to a time server and obtain the time. The problem is that the time return does not factor in the network and processing delays. To compensate for this, we simply measure the local system time at which the request is sent, let's call it T_0. We also measure the local system time at which the response in received, let's call it T_1. If we

assume that it takes the same amount of time for a request to go to the server and return from the server, then we can set our new time to be: $T = T_{server} + (T_1 - T_0) / 2$

The main issue with Cristian's algorithm is that the time server may fail and thus clock synchronization may be unavailable. An alternative would be to have the processes agree on a common time without having to resort to a time server. The Berkeley Algorithm is an algorithm that achieves this goal.

BERKELEY ALGORITHM

The Berkeley algorithm makes no assumptions on the accuracy of each clock and does not require a remote time server. The goal of the algorithm is to achieve synchronization by getting all processes to set their clocks to the average of all the clocks in the system. This is done by selecting a master process that coordinates the entire synchronization process while the remaining process are all slave processes. The algorithm works as follows:

1. A master process is chosen via an election process (such as Chang and Roberts algorithm)
2. The master polls the slaves who reply with their time
3. The master observes the Round-Trip Time of the messages and estimates the time of each slave and its own (the estimation can be done via Cristian's algorithm)
4. The master then averages the clock times, ignoring any outlier values
5. The master then sends out the amount (positive or negative) that each slave must adjust its clock.

REVIEW

We review all the above algorithms for use in the research. We dismiss Lamport Timestamps and Vector Clocks because of the property of Ordered Before where, while we know that if $a \rightarrow b$ then $C(a) < C(b)$, we cannot tell that $a \rightarrow b$ given $C(a) < C(b)$. This means that we cannot decide on which client has buzzed first just from timestamps. Timestamps require an imposed ordering, such as the index or id of the client. The problem is we want to rely on actual physical time, which cannot be inferred from the timestamps as described above. Therefore, we are compelled to rely on frequent ad-hoc clock synchronization, as per the Berkeley algorithm.

Hypothesis

In this research, we implement the Berkeley's algorithm, enhanced with Cristian's algorithm, in order to keep multiple quiz clients in sync. As such, we hypothesize that the Berkeley's algorithm will allow to reduce the error to such an extent that it seems as if the order in which the players buzz perceived by the server is the same as the actual physical order of the events.

This proposal aims to provide a general audience with evidence and experimentation sufficient to prove that such a distributed quiz game system may exist under the conditions mentioned above.

Online games have proven time and again that people are engaged when they know that they are competing against other individuals, rather than against artificial intelligence. By developing a game that is played in real time, engagement is only enhanced, therefore increasing the appeal perceived by the potential audience.

Methodology

For this research, we have opted to use the Berkeley algorithm in order to perform clock synchronization. Unfortunately, we cannot use Lamport logic because timestamps do not tell us which events physically happened in what order. Lamport logic imposes a logical ordering and as such describes which events logically happened before another. The problem with this is that logical ordering is imposed via id's or indices and has nothing to do from physical time. Therefore, we are compelled to approximate physical time by constantly synchronizing multiple clients via the Berkeley algorithm.

In this research, the server is considered the master and the clients are all the slaves. The server queries the time from each client, using Cristian's algorithm. Then, the server averages out all the times from the clients and itself and then sends the appropriate corrections to each client and itself. This process is done every 3 seconds or so in order to account for the different speeds at which time flows on each client.

A quiz game requires a design that is highly available, and easy to deploy. We are using Node.js which is a JavaScript runtime, for the logical foundation of the game, values are stored in volatile memory in the client and server side. The front end will tie everything together with HTML, CSS and JavaScript. We have deployed the application on Heroku and is available at: http://distributed-quiz.herokuapp.com/

Implementation

It was important to determine the interaction between the users and the game before the implementation phase took place. We came to the conclusion that it was necessary to provide a platform that would be controlled by a perpetual admin. The code took into consideration this premise, and we implemented a blocking mechanism that allowed the administrator to basically trigger a switch when the game was starting in order to allow for the quiz experience to commence. After this statement it could sound counter intuitive, but all the code used during the development of this game is based on Node.js, which is a non-blocking coding standard.

Another key aspect of the implementation was the algorithmic code that constantly analyzes the time discrepancies between clients and server. This algorithm was coded to correct time based on the values received from the clients. First, the server queries the times from each client using Cristian's algorithm. Then, the server calculates the time discrepancies, averages the values and sends a correction to each client and itself, based on each client's unique reported times.

The UI, or frontend, is implemented in HTML, CSS, and JS and relies on the Polymer library for simple HTML components. The UI is very simple and relies on websockets to establish a bi-directional communication scheme with the server. The server often pushes updates to the client, such as the next question or the result of who buzzed. Furthermore, in the waiting room, as soon as a new client joins the game, all clients are notified and these updates are reflected in real-time.

Conclusions and Recommendations

After researching the different algorithms presented on our preliminary presentation, the group established that the best suited algorithms for our application were Christian's and Berkeley's algorithms. We came to that conclusion based on the evidence that these algorithms allow for calculating remote discrepancies in time, based upon a control value given by a server. Berkeley's algorithm was used to achieve a distributed experience, in which any device can join the game regardless of its location and time zone. Because the game had to be driven via a centralized server, we chose to run the game from a perpetual master process, and skipped the election process mentioned earlier in the theoretical basis and literature review of the proposal.

GENERAL IMPROVEMENTS AND FUTURE WORK

The following are elements that have been considered into future versions of the application:

- Provide a backend mechanism that will synchronize the display of the questions or options regardless of when they arrive to the client. Clients with poor connections could suffer if there wasn't a mechanism to homogenize the a consistent visualization and synchronized timing of when questions are show. We could implement a mechanism that would block the display of the questions until a round of verification that every client confirms the reception of questions is through.
- Implement a mechanism that allows a user to rejoin the game after losing a socket connection.

- Implement a mechanism that automatically pauses the game if the admin is to lose socket connection.
- Provide a front-end debugging interface that allows for troubleshooting on situations where poor connectivity is part of the game experience.
- If a client were to enter an incorrect answer, the question should go to the client who buzzed second and so on, until the question is answered correctly.

Bibliography

Baillieul, John, and Panos J. Antsaklis. “Control and Communication Challenges in Networked Real-Time Systems.” *Proceedings of the IEEE*, vol. 95, no. 1, 2007, pp. 9–28.

Bharambe, Ashwin R et al. “A Distributed Architecture for Interactive Multiplayer Games.” Carnegie Mellon School of Computer Science, Jan. 2005.

Claypool, Mark, and Kajal Claypool. “Latency and Player Actions in Online Games.” *Communications of the ACM*, vol. 49, no. 11, 2006, p. 40.

Cristian, Flaviu. “Probabilistic Clock Synchronization.” *Distributed Computing*, vol. 3, no. 3, 1989, pp. 146–158.

Fidge, C. “Logical Time in Distributed Computing Systems.” *Computer*, vol. 24, no. 8, 1991, pp. 28–33.

Fidge, Colin J. “Timestamps in Message-Passing Systems That Preserve the Partial Ordering.” zoo.cs.yale.edu/classes/cs426/2012/lab/bib/fidge88timestamps.pdf.

Gusella, R., and S. Zatti. “The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD.” *IEEE Transactions on Software Engineering*, vol. 15, no. 7, 1989, pp. 847–853.

Hashimoto, Yousuke, and Yutaka Ishibashi. “Influences of Network Latency on Interactivity in Networked Rock-Paper-Scissors.” *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games - NetGames '06*, 2006.

Knutsson, Björn et al. “Peer-to-Peer Support for Massively Multiplayer Games.” *Ieee Infocom 2004*.

Lamport, Leslie. “Time, Clocks, and the Ordering of Events in a Distributed System.” *Communications of the ACM*, vol. 21, no. 7, 1978, pp. 558–565.

Lee, Kyungmin et al. “Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming.” *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '15*, 2015.

Li, Qun, and Daniela Rus. “Global Clock Synchronization in Sensor Networks.” IEEE INFOCOM, 2004, ieeexplore.ieee.org/document/1566581/.

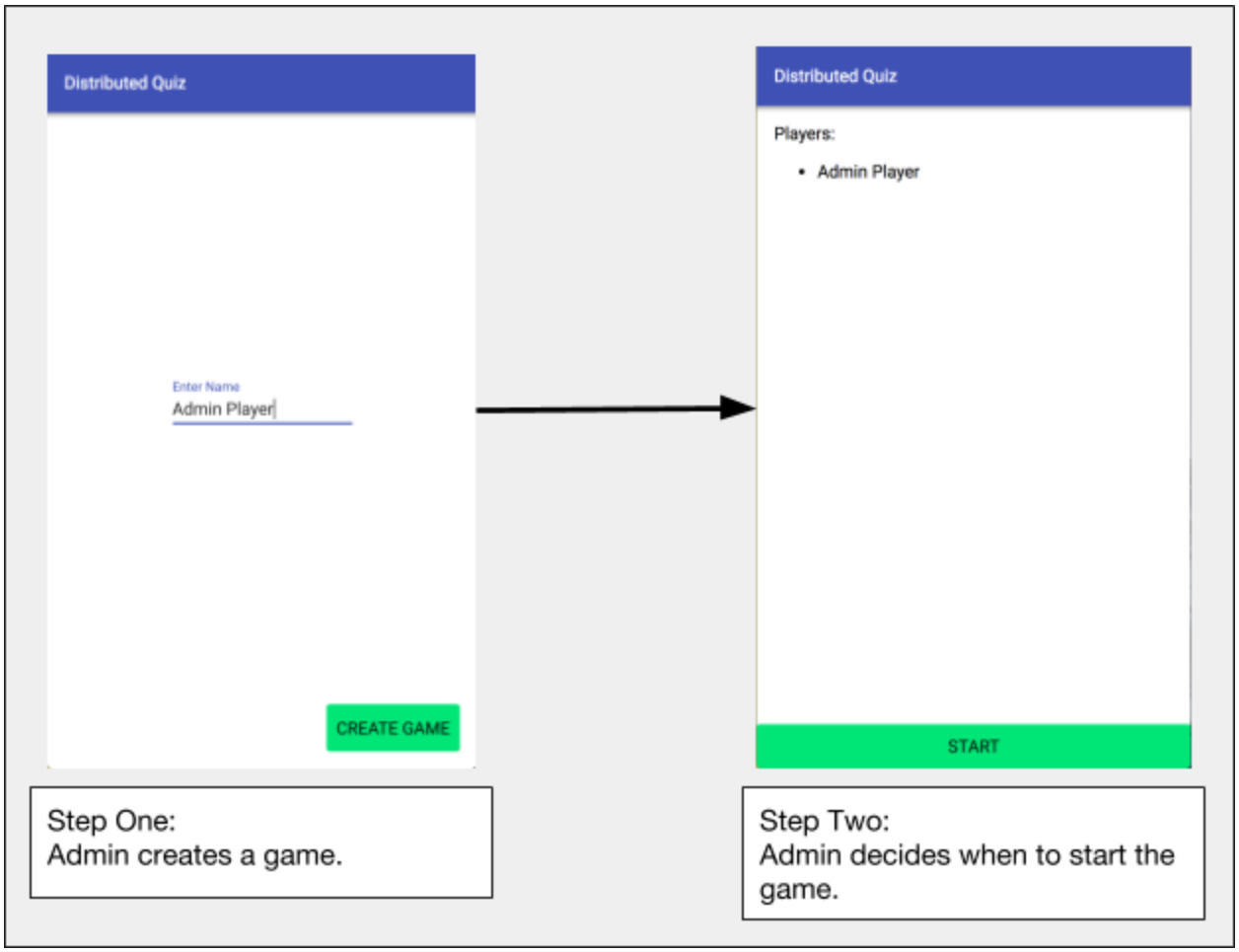
Lokuge, Kulari et al. “QuizFun: Mobile Based Quiz Game for Learning.” *2009 International Workshop on Technology for Education*, 2009.

Saga, Masaki et al. “Development of a Multiple User Quiz System on a Shared Display.” *2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing*, 2009.

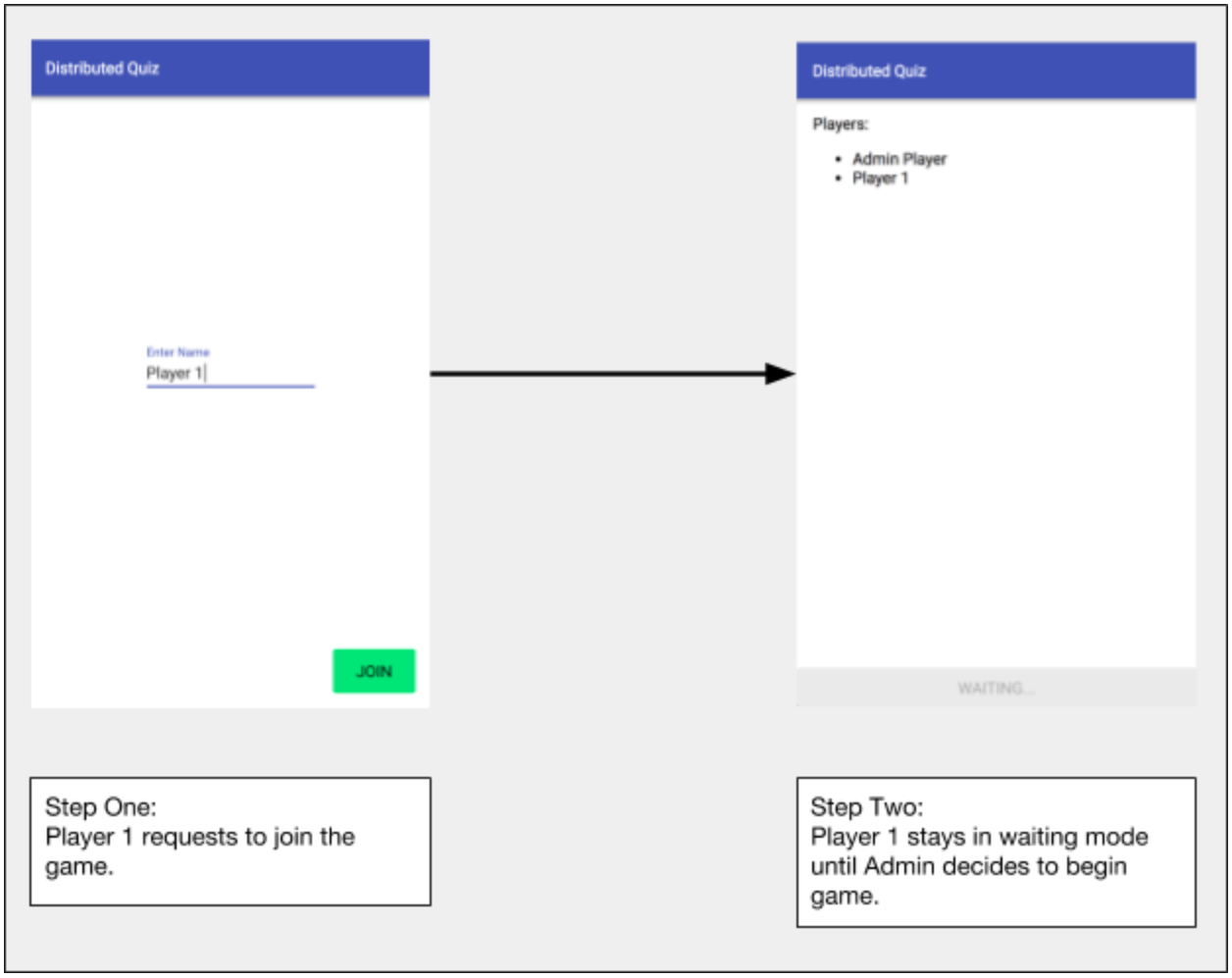
Appendix A : Game Logic Workflow and Specifics

The game logic is described in the following workflow charts:

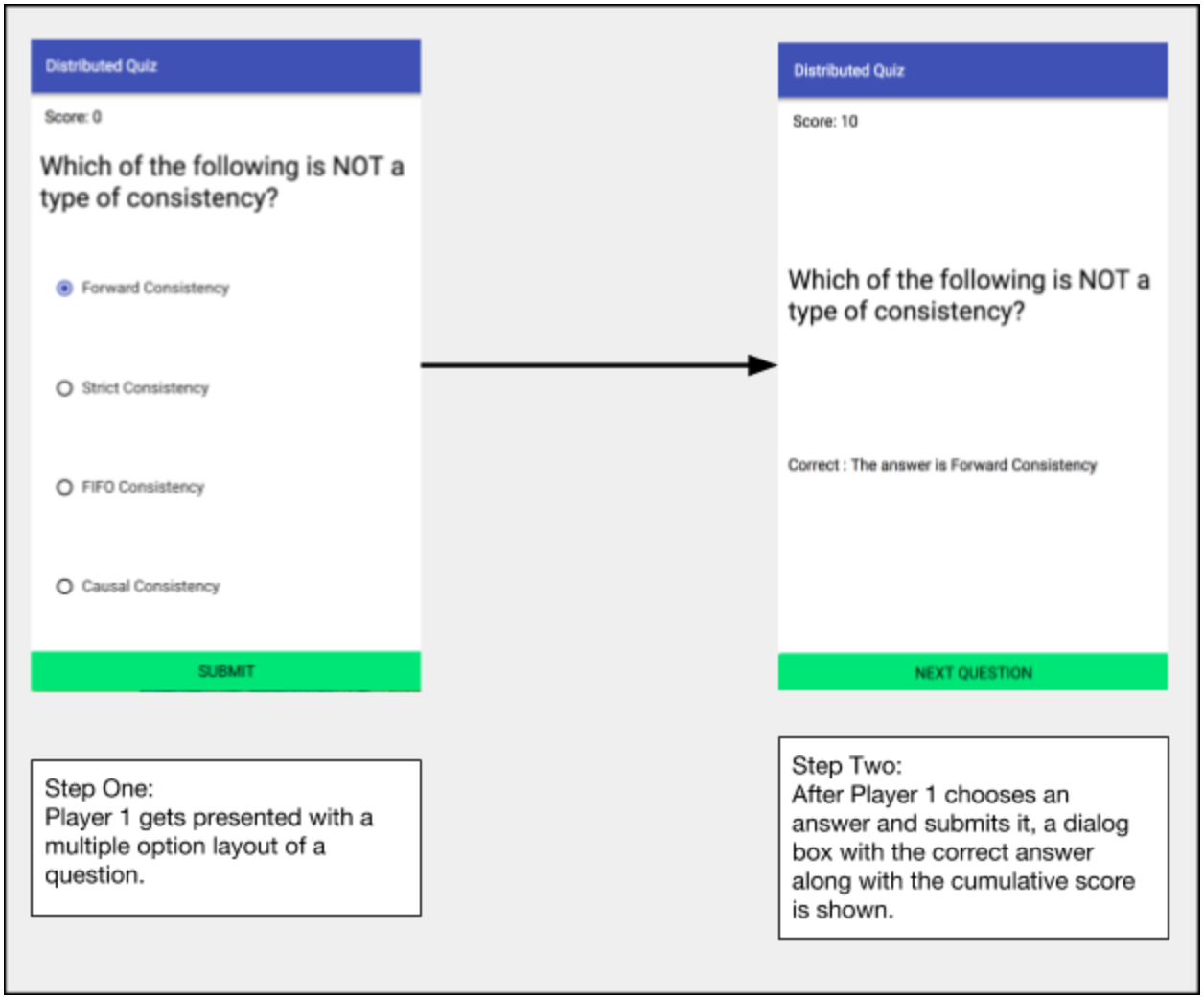
Admin Phase A1



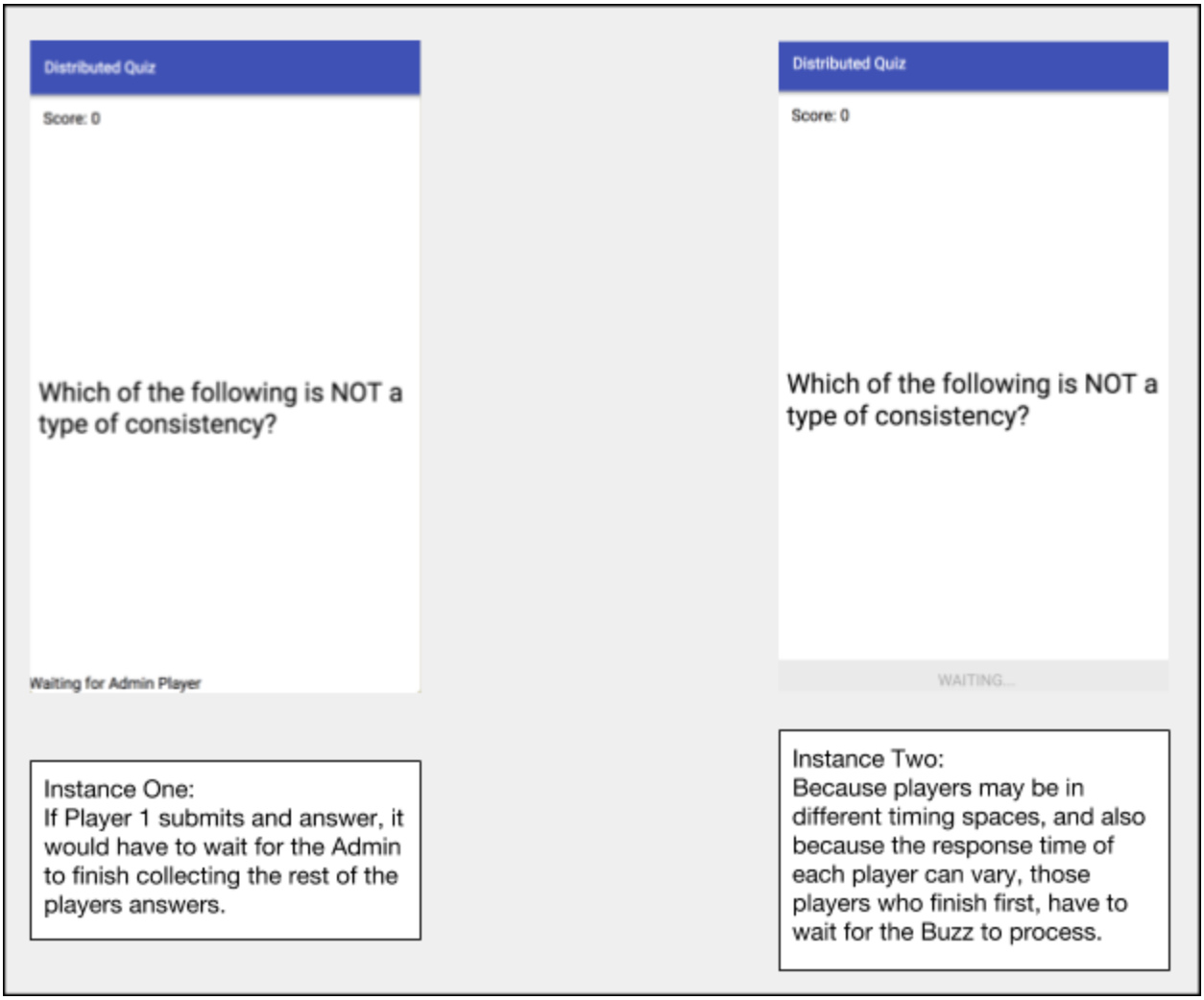
Player 1 Phase A1



Player 1 Phase B1



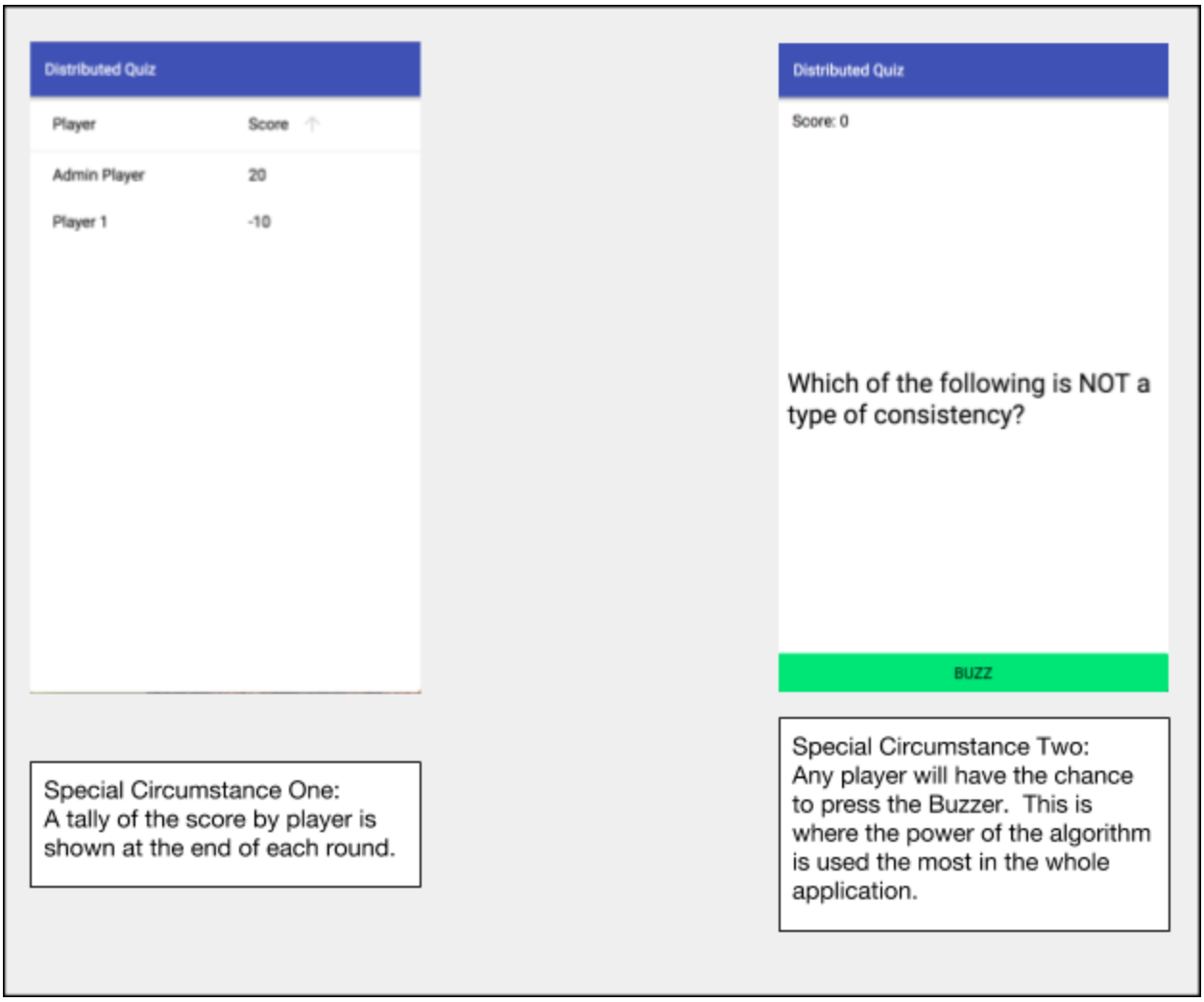
Special Circumstances A1



Instance One:
If Player 1 submits and answer, it would have to wait for the Admin to finish collecting the rest of the players answers.

Instance Two:
Because players may be in different timing spaces, and also because the response time of each player can vary, those players who finish first, have to wait for the Buzz to process.

Special Circumstances A2



Appendix B : Source Code

SERVER: app.js

```
'use strict';

const express = require('express');
const fs = require('fs');
const path = require('path');
const http = require('http');
const socketio = require('socket.io');
const request = require('request');

const Game = require('./lib/game');
const Player = require('./lib/player');

// Special reset option available to admin
// This is to be used if anything goes wrong during the demo
const ADMIN_RESET = 'Admin Reset';


// The Application
class App {
  constructor() {
     // Initialize the express app, the http server, and the socket io port
     this.app = express();
     this.httpserver = http.Server(this.app);
     this.io = socketio.listen(this.httpserver);

     // Initialize the game
     this.game = null;

     this.host = (process.env.ENVIRONMENT === 'production') ?
'https://distributed-quiz.herokuapp.com' : 'localhost';
     this.port = process.env.PORT || 8080;

     // Setup Http routes
     this._setupRoutes();

     // Setup Socket Connection
     this._setupSocketConnection();

     // Run the App
     this.run();
  }
  _setupRoutes() {
     // Lets us import files from the server on the client side
     this.app.use('/', express.static(path.join(__dirname, 'client')));

     // Sets up the home directory
     this.app.get('/', (req, res) => {
        res.sendFile(path.join(__dirname, 'client', 'index.html'));

     });
  }
```

```
_setupSocketConnection () {
    // Handle response on new socket connection
    this.io.on('connection', (socket) => {
        // We have two types of players, a normal player and an admin player
        // An admin player creates the game and moves the game forward
        // A normal player only answers questions and waits for the next question to be served
        if (this.game && this.game.admin) {
            App.notifyIsNotAdmin(socket);
        } else {
            App.notifyIsAdmin(socket);
        }

        // Setup the main socket routes
        this._setupSocketRoutes (socket);
    });
}

_setupSocketRoutes (socket) {
    // Handle creation of new game
    socket.on('create new game', (data) => {
        this.onCreateNewGame (socket, data);
    });

    // Handle joining of game
    socket.on('join game', (data) => {
        this.onJoinGame (socket, data);
    });
}

onCreateNewGame (socket, data) {
    // Create a new game
    this.game = new Game();

    // Set the new player as an admin player and add the admin player to the game
    let adminPlayer = new Player(socket, data.name);
    this.game.addPlayer(adminPlayer);

    // Notify to admin player that the game has just been created
    App.notifyGameCreated(socket, adminPlayer);
}

onJoinGame (socket, data) {
    // If game exists
    if (this.game && this.game.admin) {
        // Reset the game in the case of an admin reset
        if (data.name === ADMIN_RESET) {
            this.game.resetGame();
        } else {
            // Create the new player
            let player = new Player(socket, data.name);

            // Notify all other players that the player has been added
            App.notifyGameJoined(socket, player, this.game);

            // Add the player to the game
            this.game.addPlayer(player);
```

```
        }
      }
    }

    static notifyIsAdmin(socket) {
      socket.emit('admin', {});
    }

    static notifyIsNotAdmin(socket) {
      socket.emit('not admin', {});
    }

    static notifyGameCreated(socket, player) {
      socket.emit('game created', {
        name: player.name,
        id: player.id
      });
    }

    static notifyGameJoined(socket, player, game) {
      socket.emit('game joined', {
        name: player.name,
        id: player.id,
        players: game.players.map((player) => ({
          name: player.name,
          id: player.id
        }))
      })
    }

    // Run the server by listening to the main port or port 8080
    run() {
      if (module === require.main) {
        this.httpserver.listen(this.port);
        console.log(`Server Listening on port ${this.port}`);
      }

      module.exports = this.app;
    }
}

// Create the app and run it
const app = new App();
```

SERVER: lib/game.js

```
const async = require('./async');
const random = require('./random');
const QUESTIONS  = require('./question-repository');

class Game {
  constructor() {
     this.admin = null;
     this.players = [];
     this.isPlaying = false;
     this.correction = 0;
     this.buzzBuffer = [];
     this.isWaitingForBuzzes = false;
     this.currentQuestionIndex = 0;

     setInterval(() => {
        this.syncClocks();
     }, 3000);
  }

  resetGame() {
     this.admin = null;
     this.players = [];
     this.isPlaying = false;
     this.correction = 0;
     this.buzzBuffer = [];
     this.isWaitingForBuzzes = false;
     this.currentQuestionIndex = 0;
  }

  getTime() {
     return Date.now() + Math.round(this.correction);
  }

  correct(correction=0) {
     this.correction = this.correction + correction;
  }

  syncClocks() {
     if (this.players.length > 1) {
        console.log("Sync Clocks");

        const timeRequests  = this.players.map((player, index) => {
           return player.getTime()
              .then((time) => ({
                     index: index,
                     time: time
                 })
              );
        });

        return async.optional(timeRequests)
           .then((timeResponses) => {
              const localtime  = this.getTime();
```

```
            const numberOfClocks = timeResponses.length + 1;

            let averageTime = localtime;

            timeResponses.forEach((timeResponse) => {
                averageTime = averageTime + timeResponse.time;
                console.log(`Time Difference ${timeResponse.index}: ${localtime -
timeResponse.time} `);
            });

            averageTime = averageTime / numberOfClocks;

            const localcorrection = averageTime - localtime;
            this.correct(localcorrection);

            timeResponses.forEach((timeResponse) => {
                const correction = averageTime - timeResponse.t ime;
                this.players[timeResponse.index].correct(correction);
            });
        });
    }
  }

  static getQuestion(index) {
    const question = QUESTIONS[index];

    if (question) {
      let choices = question.choices.slice();
      choices.splice(random.randInt(choices.length), 0, question.answer);

      return {
        statement: question.statement,
        choices: choices
      };

    } else {
      return null;
    }
  }

  evaluateBuzz() {
    if (this.buzzBuffer.length === 0) {
      setTimeout(() => {
        this.evaluateBuzz();
      }, 3000);
    } else {
      this.buzzBuffer.sort((i1, i2) => i1.timestamp - i2.timestamp);
      const currentPlayer = this.buzzBuffer[0].player;

      this.players.forEach((player) => {
        if (player == currentPlayer) {
          player.answer();
        } else {
          player.waitForAnswer(currentPlayer);
        }
      });
```

```
        this.isWaitingForBuzzes = false;
        this.buzzBuffer = [];
    }
}

sendNextQuestion () {
    if (this.currentQuestionIndex < QUESTIONS.length) {
        let question = Game.getQuestion(this.currentQuestionIndex);
        this.currentQuestionIndex = this.currentQuestionIndex + 1;
        this.players.forEach((player) => {
            player.sendNextQuestion (question);
        });
    } else {
        this.sendGameOver ();
    }
}

getScoreTable () {
    return this.players.map((player) => ({
        name: player.name,
        id: player.id,
        score: player.score
    }));
}

sendGameOver () {
    const scoreTable = this.getScoreTable ();
    this.players.forEach((player) => {
        player.sendGameOver (scoreTable);
    });
    this.resetGame ();
}

getCurrentAnswer () {
    if (this.currentQuestionIndex < QUESTIONS.length + 1) {
        return QUESTIONS[this.currentQuestionIndex - 1].answer;
    } else {
        return null;
    }
}

processAnswer (answer, player) {
    console.log("Provided Answer: ");
    console.log(answer);
    console.log("Correct Answer: ");
    console.log(this.getCurrentAnswer ());
    const correctAnswer = this.getCurrentAnswer ();
    if (answer === this.getCurrentAnswer ()) {
        player.score = player.score + 10;
        this.players.forEach((player) => {
            player.sendAnswerIsCorrect (correctAnswer);
        });
    } else {
        player.score = player.score - 5;
        this.players.forEach((player) => {
            player.sendAnswerIsIncorrect (correctAnswer);
        });
```

```
        }
    }

    handleBuzz(player, socket, data) {
        console.log(`Buzzed at ${data.timestamp}`);
        if (!this.isWaitingForBuzzes) {
            this.isWaitingForBuzzes = true;
            this.buzzBuffer = [];

            setTimeout(() => {
                this.evaluateBuzz();
            }, 3000);
        }
        const bufferItem = {
            player: player,
            timestamp: data.timestamp
        };
        this.buzzBuffer.push(bufferItem);
    }

    addPlayer(player) {
        if (!this.isPlaying) {

            player.onBuzz = (socket, data) => {
                this.handleBuzz(player, socket, data)
            };

            player.onAnswer = (socket, data) => {
                this.processAnswer(data.answer, player);
            };

            if (!this.admin) {
                player.onStartGame = (socket, data) => {
                    this.sendNextQuestion();
                };

                player.onNextQuestion = (socket, data) => {
                    this.sendNextQuestion();
                };

                this.admin = player;
            }

            this.players.push(player);

            this.players.forEach((p) => {
                p.sendNewPlayerJoined(player);
            });
        }
    }
}

module.exports = Game;
```

SERVER: lib/player.js

```
const Promise = require('bluebird');
const uuid = require('uuid/v4');

class Player {

  constructor(socket, name) {
     this.score = 0;
     this.socket = socket;
     this.name = name;
     this.id = uuid();

     // Event Handlers
     this.onRespondTime = null;
     this.onStartGame = null;
     this.onNextQuestion = null;
     this.onBuzz = null;
     this.onAnswer = null;

     // Setup Socket Handlers
     this._setupSocketHandlers();
  }

  _setupSocketHandlers() {
     this.socket.on('respond time', (data) => {
        if (this.onRespondTime) {
           this.onRespondTime(this.socket, data);
        }
     });

     this.socket.on('start game', (data) => {
        if (this.onStartGame) {
           this.onStartGame(this.socket, data);
        }
     });

     this.socket.on('next question', (data) => {
        if (this.onNextQuestion) {
           this.onNextQuestion(this.socket, data);
        }
     });

     this.socket.on('buzz', (data) => {
        if (this.onBuzz) {
           this.onBuzz(this.socket, data);
        }
     });

     this.socket.on('select answer', (data) => {
        if (this.onAnswer) {
           this.onAnswer(this.socket, data);
        }
     });
  }
```

```
  getTime() {
     return new Promise((resolve) => {
        const sendTime = Date.now();
        this.socket.emit('request time', {});
        this.onRespondTime = (socket, data) => {
           console.log(data);
           const receiveTime = Date.now();
           const roundTripTime = receiveTime - sendTime;
           const remoteTime = data.timestamp;
           const time = remoteTime + Math.round(roundTripTime / 2);
           resolve( time);
        };
     });
  }
  correct(correction) {
     this.socket.emit('correct time', {
        correction: correction
     });
  }
  answer() {
     this.socket.emit('answer', {});
  }
  waitForAnswer(player) {
     this.socket.emit('wait for answer', {
        name: player.name,
        id: player.id
     });
  }
  sendNextQuestion(question) {
     this.socket.emit('next question', question);
  }
  sendGameOver(scoreTable) {
     this.socket.emit('game over', {
        scoreTable: scoreTable
     });
  }
  sendNewPlayerJoined(player) {
     this.socket.emit('new player joined', {
        id: player.id,
        name: player.name
     });
  }
  sendAnswerIsCorrect(answer) {
     this.socket.emit('correct', {
        answer: answer,
        score: this.score
     });
  }
  sendAnswerIsIncorrect(answer) {
     this.socket.emit('incorrect', {
        answer: answer,
        score: this.score
     });
  }
}

module.exports = Player;
```

SERVER: lib/async.js

```
const Promise = require('bluebird');

// Function that evaluates a list of promises, only keeping the promises that are successful
function optional(promises) {
  return Promise.all(
     promises.map((promise) => promise.reflect()))
           .filter((inspection) => inspection.isFulfilled())
           .map((inspection) => inspection.value());
}

module.exports = {
  optional: optional
};
```

SERVER: lib/random.js

```
// Function to generate a random uniformly distributed integer from 0 to n - 1
function randInt(n) {
  return Math.floor(Math.random() * n);
}

module.exports = {
  randInt: randInt
};
```

CLIENT: client/index.html

```
<!DOCTYPE html>
<html lang="en">
   <head>
       <meta charset="UTF-8">
       <meta name="viewport" content="width=device-width, initial-scale=1.0">
       <title>Distributed Quiz</title>
       <script src="bower_components/webcomponentsjs/webcomponents-lite.min.js"></script>
        <link rel="import" href="components/d-app.html">
       <style>
           html, body {
               height: 100%;
               width: 100%;
               margin: 0;
               font-family: 'Roboto', 'Noto', 'Helvetica Neue', 'Arial', sans-serif;
           }
       </style>
   </head>
   <body class="fullbleed layout vertical">
       <d-app></d-app>
   </body>
</html>
```

CLIENT: client/components/d-app.html

```
<!-- Import Polymer, the framework used to make the client -->
<link rel="import" href="../bower_components/polymer/polymer.html">

<!-- Import the components used -->
<link rel="import" href="../bower_components/paper-button/paper-button.html">
<link rel="import" href="../bower_components/paper-header-panel/paper-header-panel.html">
<link rel="import" href="../bower_components/paper-toolbar/paper-toolbar.html">
<link rel="import" href="../bower_components/paper-input/paper-input.html">
<link rel="import" href="../bower_components/paper-radio-group/paper-radio-group.html">
<link rel="import" href="../bower_components/paper-radio-button/paper-radio-button.html">
<link rel="import" href="../bower_components/iron-data-table/iron-data-table.html">

<!-- Import the socket io library -->
<script src="../bower_components/socket.io-client/dist/socket.io.js"></script>

<!-- The Application component definition -->
<dom-module id="d-app">
    <template>
        <style>
            :host {
                display: block;
                width: 100%;
                height: 100%;
            }

            .new-game-container {
                flex: 1;
                display: flex;
                align-items: center;
                justify-content: center;
            }

            .container {
                height: 100%;
                width: 100%;
                display: flex;
                flex-direction: column;
                position: absolute;
                justify-content: space-between;
            }

            .button-container {
                height: 76px;
                width: 100%;
                display: flex;
                flex-direction: row;
                align-items: center;
                justify-content: flex-end;
            }

            .create-button {
                height: 44px;
                background-color: #00e676;
                margin-right: 16px;
```

```
    }

    .player-list-container {
        flex: 1;
        margin: 16px;
        overflow: scroll;
    }

    .action-button {
        width: 100%;
        margin: 0;
        background-color: #00e676;
    }

    .disabled-button {
        width: 100%;
        margin: 0;
    }
    .statement {
        margin: 10px;
        display: flex;
        align-items: center;
        justify-content: center;
        font-size: 20pt;
    }
    .choices {
        flex: 2;
        display: flex;
        flex-direction: column;
        justify-content: space-around;
        margin: 15px;
    }
    .response {
        height: 80px;
        margin: 10px;
    }

    .score {
        margin: 15px;
    }
</style>

<paper-header-panel>
    <paper-toolbar class="toolbar">
        <div>Distributed Quiz</div>

    </paper-toolbar>

    <!-- The Enter Name Screen -->
    <template is="dom-if" if="[[_shouldRenderEnterNameScreen]]">
        <div class="content fit container">
            <div class="new-game-container">
                <paper-input label="Enter Name" value="{{name}}"></paper-input>
            </div>
            <div class="button-container">

                <template is="dom-if" if="[[isAdmin]]">
```

```
                    <paper-button class="create-button" on-tap="createNewGame">
                        CREATE GAME
                    </paper-button>
                </template>
                <template is="dom-if" if="[[!isAdmin]]">
                    <paper-button class="create-button" on-tap="joinGame">
                        JOIN
                    </paper-button>
                </template>
            </div>
        </div>
    </template>

    <!-- The Join Game Screen -->
    <template is="dom-if" if="[[_shouldRenderJoinGameScreen]]">
        <div class="content fit container">
            <div class="player-list-container">
                <span>Players:</span>
                <ul>
                    <template is="dom-repeat" items="{{players}}">
                        <li>{{item.name}}</li>
                    </template>
                </ul>
            </div>
            <template is="dom-if" if="[[_shouldRenderStartButton]]">
                <paper-button class="action-button" on-tap="startGame">
                    START
                </paper-button>
            </template>
            <template is="dom-if" if="[[_shouldRenderWaitingDisabledButton]]">
                <paper-button disabled class="disabled-button">
                    WAITING...
                </paper-button>
            </template>
        </div>
    </template>

    <!-- The Question Screen -->
    <template is="dom-if" if="[[_shouldRenderQuestion]]">
        <div class="container">
            <div class="score">
                Score: {{score}}
            </div>
            <span class="statement">
                [[statement]]
            </span>

            <template is="dom-if" if="[[_shouldRenderWaitingPage]]">
                <div class="response">
                    Waiting for {{playerCurrentlyAnswering}} to answer
                </div>
            </template>

            <template is="dom-if" if="[[_shouldRenderChoices]]">
                <paper-radio-group selected="{{selectedAnswer}}" class="choices">
                    <template is="dom-repeat" items="{{choices}}">
```

```
                                <paper-radio-button
name="{{index}}">{{item}}</paper-radio-button>
                                </template>
                            </paper-radio-group>

                            <paper-button class="action-button" on-tap="selectAnswer">
                                Submit
                            </paper-button>
                        </template>

                        <template is="dom-if" if="[[_shouldRenderBuzzButton]]">
                            <paper-button class="action-button" on-tap="buzz">
                                Buzz
                            </paper-button>
                        </template>

                        <template is="dom-if"
if="[[_shouldRenderWaitingBuzzResponseDisabledButton]]">
                            <paper-button disabled class="disabled-button">
                                WAITING...
                            </paper-button>
                        </template>

                        <template is="dom-if" if="[[_shouldRenderCorrectPage]]">
                            <div class="response">
                                Correct : The answer is {{correctAnswer}}
                            </div>
                        </template>

                        <template is="dom-if" if="[[_shouldRenderIncorrectPage]]">
                            <div class="response">
                                Incorrect : The answer is {{correctAnswer}}
                            </div>

                        </template>

                        <template is="dom-if" if="[[_shouldRenderNextQuestionButton]]">
                            <paper-button class="action-button" on-tap="nextQuestion">
                                NEXT QUESTION
                            </paper-button>
                        </template>
                    </div>
                </template>

                <!-- The Score Table Screen -->
                <template is="dom-if" if="[[_shouldRenderScoreTable]]">
                    <iron-data-table items="[[scoreTable]]">
                        <data-table-column name="Player">
                            <template>[[item.name]]</template>
                        </data-table-column>
                        <data-table-column name="Score" sort-by="score">
                            <template>[[item.score]]</template>
                        </data-table-column>
                    </iron-data-table>
                </template>
            </paper-header-panel>
        </template>
```

```
<script>
    // Application player state constants
    const EMPTY_STATE = 'empty state';
    const CREATING_NEW_GAME = 'creating new game';
    const JOINING_GAME = 'joining game';
    const PLAYING = 'playing';
    const DONE_PLAYING = 'done playing';

    // Question state constants
    const NOT_BUZZED = 'not buzzed';
    const BUZZED = 'buzzed';
    const ANSWERING = 'answering';
    const WAITING = 'waiting';
    const CORRECT = 'correct';
    const INCORRECT = 'incorrect';

    // Application component
    // The entire component is defined here
    Polymer({
        // The name of the component
        is: "d-app",
         // The list of properties
        properties: {
            // The score table
            scoreTable: {
                type: Array,
                 value: []
            },

             // The web socket
            socket: Object,

             // The player name
            name: {
                 type: String,
                 value: ""
            },

             // The player state of the application
            state: {
                 type: String,
                 value: EMPTY_STATE
            },

             // The question state
            questionState: {
                 type: String,
                 value: NOT_BUZZED
            },

             // Flag if player is an admin player
            isAdmin: {
                 type: Boolean,
                 value: true
            },

             // The id of the player
```

```
id: Number,

// The current question statement
statement: String,

// The question choices
choices: Array,

// The amount by which the physical clock is to be corrected
timeCorrection: {
    type: Number,
    value: 0
},

// The list of players in the game
players: {
    type: Array,
    value: []
},

// The current player score
score: {
    type: Number,
    value: 0
},

// The current selected answer index
selectedAnswer:{
    type: Number,
    value: 0
},

// The correct answer
correctAnswer: String,

// The name of the player currently answering
playerCurrentlyAnswering: String,

// Flag indicating if we should be rendering the enter name screen
_shouldRenderEnterNameScreen: {
    computed: '_computeShouldRenderEnterNameScreen(state)'
},

// Flag indicating if we should be rendering the join game screen
_shouldRenderJoinGameScreen: {
    computed: '_computeShouldRenderJoinGameScreen(state)'
},

// Flag indicating if we should be rendering the start game button
_shouldRenderStartButton: {
    computed: '_computeShouldRenderStartButton(state, isAdmin)'
},

// Flag indicating if we should be rendering the waiting button
_shouldRenderWaitingDisabledButton: {
    computed: '_computedShouldRenderWaitingDisabledButton(state, isAdmin)'
},
```

```
                // Flag indicating if we should be rendering the question
                _shouldRenderQuestion: {
                    computed: '_computeShouldRenderQuestion(state)'
                },

                // Flag indicating if we should be rendering the choices
                _shouldRenderChoices: {
                    computed: '_computeShouldRenderChoices(state, questionState)'
                },

                // Flag indicating if we should be rendering the waiting page
                _shouldRenderWaitingPage: {
                    computed: '_computeShouldRenderWaitingPage(state, questionState)'
                },

                // Flag indicating if we should be rendering the buzz button
                _shouldRenderBuzzButton: {
                    computed: '_computeShouldRenderBuzzButton(state, questionState)'
                },

                // Flag indicating if we should be rendering the correct page
                _shouldRenderCorrectPage: {
                    computed: '_computeShouldRenderCorrectPage(state, questionState)'
                },

                // Flag indicating if we should be rendering the incorrect page
                _shouldRenderIncorrectPage: {
                    computed: '_computeShouldRenderIncorrectPage(state, questionState)'
                },

                // Flag indicating if we should be rendering the next question button
                _shouldRenderNextQuestionButton: {
                    computed: '_computeShouldRenderNextQuestionButton(state, questionState,
isAdmin)'
                },

                // Flag indicating if we should be rendering the score table
                _shouldRenderScoreTable: {
                    computed: '_computeShouldRenderScoreTable(state)'
                },

                // Flag indicating if we should be rendering the waiting for buzz response
disabled button
                _shouldRenderWaitingBuzzResponseDisabledButton: {
                    computed: '_computeShouldRenderWaitingBuzzResponseDisabledButton(state,
questionState)'
                }
            },
            // Method that runs as soon as the component is ready
        ready() {
                // Setup the socket connection
            this._setupSocketConnection();

            // Setup the socket routes
            this._setupSocketRoutes();
        },
```

```
_setupSocketConnection () {
this.socket = io();
},

_setupSocketRoutes () {
this.socket.on('request time', () => {
   this.onRequestTime ();
});

this.socket.on('correct time', (data) => {
   this.onCorrectTime (data);
});

this.socket.on('game created', (data) => {
   this.onGameCreate (data);
});

this.socket.on('game joined', (data) => {
   this.onGameJoined (data);
});

this.socket.on('next question', (data) => {
   this.onNextQuestion (data);
});

this.socket.on('admin', (data) => {
   this.onAdmin ();
});

this.socket.on('not admin', (data) => {
   this.onNotAdmin ();
});

this.socket.on('new player joined', (data) => {
   this.onNewPlayerJoined (data);
});

this.socket.on('answer', () => {
   this.onAnswer ();
});

this.socket.on('wait for answer', (data) => {
   this.onWaitForAnswer (data);
});

this.socket.on('correct', (data) => {
   this.onCorrect (data);
});

this.socket.on('incorrect', (data) => {
   this.onIncorrect (data);
});

this.socket.on('game over', (data) => {
   this.onGameOver (data);
});
},
```

```
    // Socket event handlers
onRequestTime () {
    let  timestamp  = Date.now() + Math.round(this.timeCorrection);
    this.socket.emit('respond time', {
        timestamp: timestamp
    });
},
onCorrectTime (data) {
    this.timeCorrection = this.timeCorrection + data.correction;
},
onGameCreate (data) {
    this.id = data.id;
},
onGameJoined (data) {
    this.id = data.id;
    data.players.forEach((player) => {
        this.push('players', player);
    });
},
onNextQuestion (data) {
    this.statement = data.statement;
    this.choices = data.choices;
    this.questionState = NOT_BUZZED;
    this.state = PLAYING;
},
onAdmin () {
    this.isAdmin = true;
},
onNotAdmin () {
    this.isAdmin = false;
},
onNewPlayerJoined (data) {
    this.push('players', {
        id: data.id,
        name: data.name
    });
},
onAnswer () {
    this.questionState = ANSWERING;
},
onWaitForAnswer (data) {
    this.questionState = WAITING;
    this.playerCurrentlyAnswering = data.name;
},
onCorrect (data) {
    this.correctAnswer = data.answer;
    this.questionState = CORRECT;
    this.score = data.score;
},
onIncorrect (data) {
    this.correctAnswer = data.answer;
    this.questionState = INCORRECT;
    this.score = data.score;
},
onGameOver (data) {
    this.scoreTable = data.scoreTable;
```

```
    this.state = DONE_PLAYING;
},

    // Methods that run when client events are handled
createNewGame () {
    this.state = JOINING_GAME;
    this.isAdmin = true;
    this.socket.emit("create new game",{name:this.name})
},
joinGame (){
    this.state = JOINING_GAME;
    this.socket.emit("join game",{name:this.name})
},
startGame () {
    this.socket.emit("start game",{});
},
buzz () {
    this.socket.emit('buzz', {
       timestamp: Date.now() + Math.round(this.timeCorrection)
    });
    this.questionState = BUZZED;
},
selectAnswer () {
    this.socket.emit('select answer', {
       answer: this.choices[this.selectedAnswer]
    });
},
nextQuestion () {
    this.socket.emit('next question', {});
},

    // Methods to compute the computed properties
    _computeShould RenderEnterNameScreen (state) {
        return state === EMPTY_STATE;
    },
    _computeShouldRenderCreateNewGameScreen (state) {
        return state === CREATING_NEW_GAME;
    },
    _computeShouldRenderJoinGameScreen (state) {
        return state === JOINING_GAME;
    },
    _computeShouldRenderStartButton (state, isAdmin) {
        return (state === JOINING_GAME) && isAdmin;
    },
    _computedShouldRenderWaitingDisabledButton (state, isAdmin) {
        return (state === JOINING_GAME) && (!isAdmin);
    },
    _computeShouldRenderQuestion (state){
        return state === PLAYING;
    },
    _computeShouldRenderChoices (state, questionState) {
        return (state === PLAYING) && (questionState ===  ANSWERING);
    },
    _computeShouldRenderWaitingPage (state, questionState) {
        return (state === PLAYING) && (questionState ===  WAITING);
    },
    _computeShouldRenderBuzzButton (state, questionState) {
```

```
            return (state === PLAYING) && (questionState ===  NOT_BUZZED);
        },
        _computeShouldRenderCorrectPage (state, questionState) {
            return (state === PLAYING) && (questionState ===  CORRECT);
        },
        _computeShouldRenderIncorrectPage (state, questionState) {
            return (state === PLAYING) && (questionState ===  INCORRECT);
        },
        _computeShouldRenderNextQuestionButton (state, questionState, isAdmin){
            return (state === PLAYING) &&
                   ((questionState ===  INCORRECT) || (questionState ===  CORRECT)) &&
                   (isAdmin);
        },
      _computeShouldRenderScoreTable (state) {
           return state === DONE_PLAYING;
        },
      _computeShouldRenderWaitingBuzzResponseDisabledButton  (state, questionState){
           return (state === PLAYING) && (questionState ===  BUZZED);
        }
    });
  </script>
</dom-module>
```