CS262 Final project
Adolfo Roquero, Leticia Schettino

Spring 2023

QuiP2Plash: implementing a fault-tolerant peer-to-peer multiplayer game

Link to Github Repository: https://github.com/AdolfoRoquero/cs262

Quiplash description and rules

Quiplash is a party game developed and published by Jackbox Games. It is a multiplayer game that can be played with up to 8 players. The objective of the game is to provide funny answers to prompts in a head-to-head competition with other players. Each randomly selected prompt for the round will be sent to exactly two players and each player receives two prompts. Players have to come up with a witty response to their prompts within a time limit. Once all players have submitted their responses, players vote on which response is the funniest. The player with the most votes wins that round.

For this final project, we have adapted the rules of Quiplash (slightly). Before we discuss the details of our implementation, focusing on the distributed system part of the project, we will review the basic rules and flow of the game.

How to play the game

- Anyone who wants to play the game will run the program and be prompted to make a decision between starting a new game or joining someone else's game.
- If a player chooses to start a new game, they will be the hosts for the game, which means they start out as the primary server (and remain so until the end of the game unless their system crashes). Once a player starts a game, they can share their game code with friends to have them join the game[1].
- There needs to be at least 2 players in a game.
- The host player will make the decision to officially start playing the game by pressing enter on their screen once all the players they are expecting have connected and joined the waiting room for the game.
- (official game start) All players will be presented with prompts of questions to answer and they have a time limit to do so.
- Once all players have answered all of their assigned questions or if the time to answer both questions run out, players will proceed to voting for the funniest responses, which will be presented to each of them anonymously.

[1] The game code is the IP address and port of the host player, which is used to establish a GRPC connection between the nodes. Note: this implies that a user can start a game in one terminal and join as another player as long as they specify a different port. This version of the game requires players to be connected to the same wifi network.

- Once all players have voted for their favorite response to each question that was assigned to players or if the time to vote for all questions runs out, the final score will be computed.

Project goal

Our goal was to implement a peer-to-peer version of the game, designed to be ***n-1*** fault-tolerant when there are ***n*** initial players in the game. The advantages of having a peer-to-peer design in this case is that any player can host a game at any time and have other players join their game. Such a distributed system would also allow for any player system to crash without disrupting the game experience for other players (given that it is able to replicate the game state across all of its player nodes).

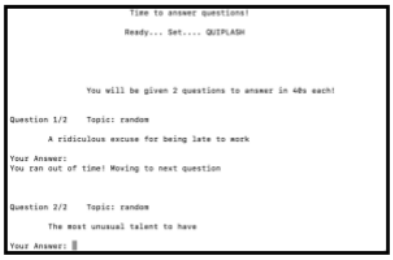
This peer-to-peer design has real applications in distributed game development since it allows for the move away from the centralized-server model where a single server responds to all requests and performs the bulk of the work. The goal of our system is to be able to run the game in each client, thus avoiding server hosting costs for game developers.

Current Implementation and Terminal Game screens

Below is the view for the terminal-based version of the game.

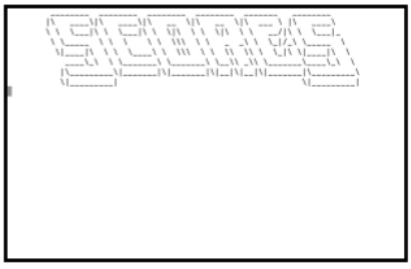
Welcome screen

Waiting screen (view from primary)

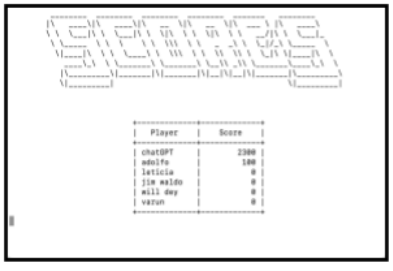


Question screen

Voting screen



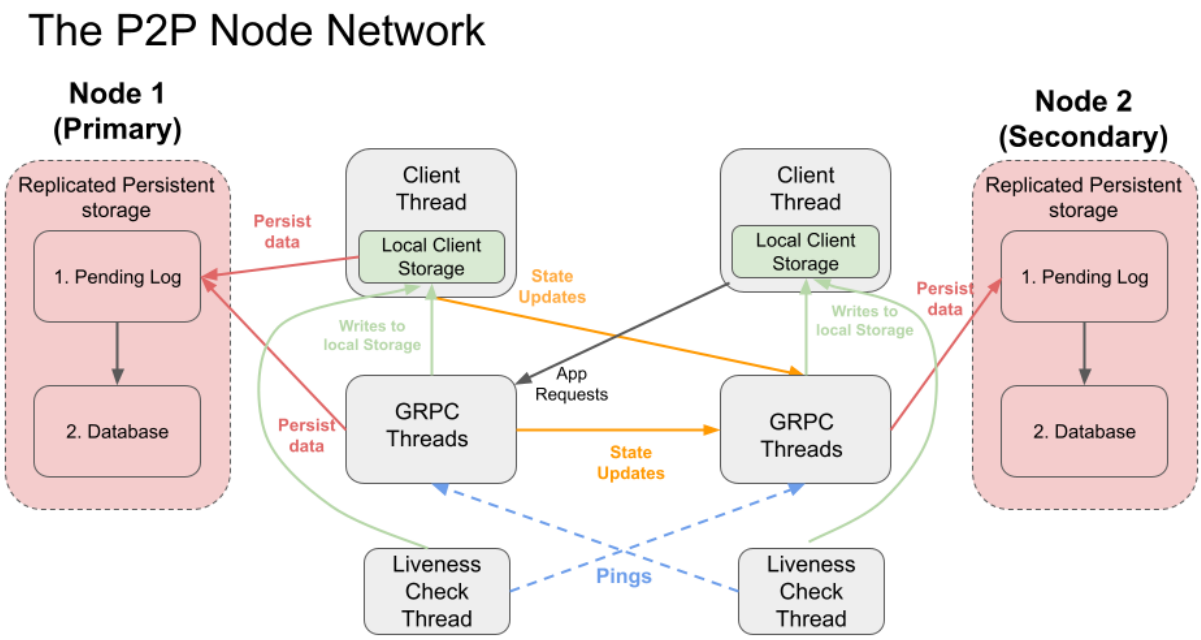
Scores waiting screen



Game results screen

System design and replication

The key element of how we designed this application is that every node should be able to behave as both a client (UI facing towards the player) as well as a replicated server (inside the peer-to-peer network). Thus, we call each of these units **NODES** where every node is running several threads that are tied by the same failure conditions (if one thread fails, all others must all fail and the node is “down”).
The idea of the peer-to-peer network is to let any node host (“Create”) a game, but also take over the hosting of the game in case the primary host fails. In order to implement such features, we developed a peer-to-peer network (from now on the **P2PNet**) as described in the graph below.



Each node has 3 different types of threads:

- **GRPC Threads** that handle the communication between nodes.
- **Client Thread** that handles the player UI as well as makes some App requests
- **Liveness Check Thread** that periodically pings nodes to check which in the **P2PNet** are running.

Each node also has 2 types of storage:

- **Local Client Storage (LCS)** which takes the form class attributes (are not written to disk) nor shared between nodes.

- **Replicated Persistent Storage (RPS)** which is a set of files (written to disk) that keep track of the game state globally and which are replicated by the **P2PNet** in order to achieve fault tolerance.

Nodes in our **P2PNet** run in two possible **modes**:

- **Primary:** At any given time, the **P2PNet** only has **one** primary node who is in charge of *administering* the game (timing, change of stage, …) and *coordinating* data replication.
- **Secondary**: There are many secondary nodes (one for each player) which primarily handle the UI for their player (based on synchronization notifications from **Primary**) and also, replicate the global game state base on the state updates (received from **Primary**)

Communication

We use GRPC for communication between the nodes in a similar way as we did in design exercise 3. For every action a player could take that would modify the game state (i.e. join the game, send an answer to a prompt or vote for a response), the client thread sends a GRPC request to the primary node that gets handled by the GRPC thread.

If the player making the state-modifying action is the secondary node, the Primary receives the attempted state change, validates it and propagates it to other secondary nodes (via State Updates).

If the player making the action is the primary node (hosting the game), it does not make sense to send requests within the same node and the changes are dealt with directly and propagated to the other nodes through what we call ‘state update’ GRPC methods. In particular, the **RPS** is updated only after game state changes have propagated whereas the **LCS** can be done directly.

This means that for every state-modifying action that can be taken, we have 2 related GRPC methods, for example, a GRPC method ‘SendAnswer’ which is sent from secondary client thread and executed by the primary node and ‘SendAnswer_StateUpdate’ which is a request sent from the GRPC thread of the **Primary** node and executed by the **Secondary** node.

Fault tolerance

Our system allows for both primary servers as well as secondary servers to crash without the game being disrupted.

Secondary nodes can fail at **any** point during the game without causing system failures. All their previous answers and votes are still visible to other players and other players can even still vote for a failed-player answer.

In the case of the primary node, our system only covers the cases where the primary server crashes after the game has started. Unlike in design exercise 3 where we assumed that the servers had the IP and port address of all other servers, in our application we do not make this

assumption. This means that we need to wait for all users to join before and require a queue (press <ENTER>) from the initial primary server to begin the game.
Once the game has started, which we define as the moment when the primary host presses enter on his screen, then fault tolerance holds and the primary nodes can crash with other nodes taking over as primary (and keeping track of timing …).

Leader election in the case of primary node crash

We employ a very simple protocol for leader election where we simply pick one of the remaining living nodes in alphabetical order based on their IP:Port, which is how we uniquely identify nodes.

Replication in the system

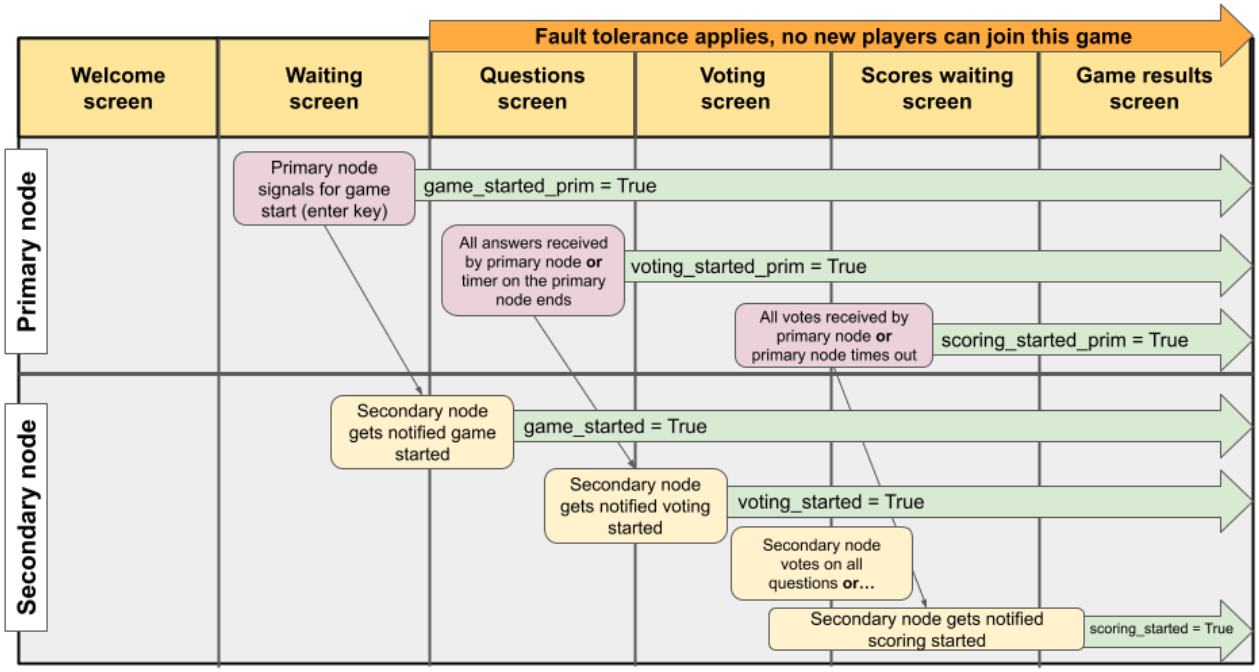
We employed a similar approach as in design exercise 3, where to ensure eventual consistency in the game state across the nodes, changes to the database (which holds the game state) are always mediated through a log file where pending changes are added. Thus, the database is only modified by a method called **execute_log(),** which reads from the local storage log file for the node.

If the player on the **primary** node performs an action that will change the game state, the primary node will not execute that change directly in the database. Instead, it will first add it to the pending log, send a **state update** GRPC request to all of the secondary nodes and await a response that the request was successfully received by all of the secondary nodes. Only once the responses have been received, the node can proceed with the call to **execute_log().**
If one of the state update requests fails (no response received on timeout), the node assumes that the corresponding secondary node has permanently crashed and will no longer expect requests from it **.**

Similarly, a **state update** request received by the secondary node first adds the proposed change to the game state to its local storage pending log, then makes a call to the **execute_log()** method.

An important consideration is that we do not handle persistence as we did in design exercise 3 (i.e. nodes crashing and being rebooted, without losing changes that were made in the meantime) since it does not make sense with the game design. Since there is only one round for the game and the stages are controlled by timeouts in the primary node, a noded that crashes cannot be recovered as the game has kept running without it.

Control and synchronization scheme



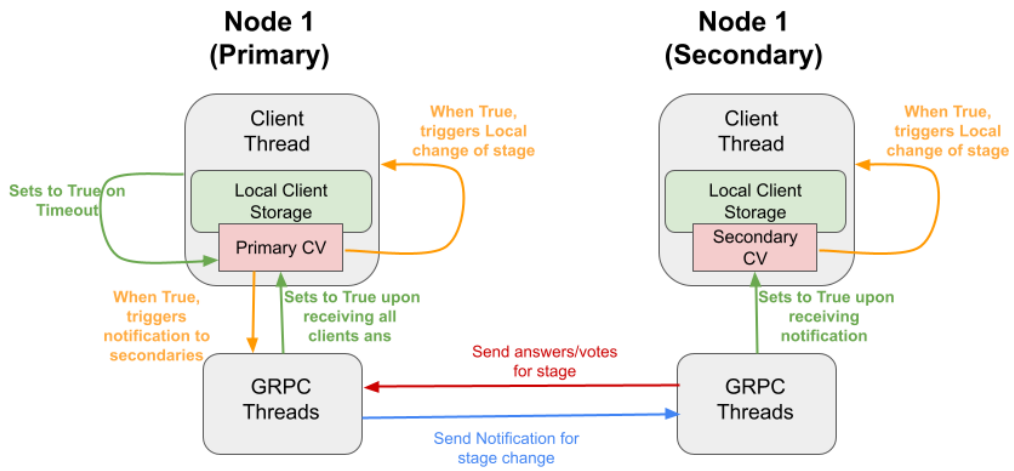
The chart above shows how we synchronize and control the stages of the game between the primary and secondary nodes. There are 4 stages of the game: setup, question stage, voting stage and scoring stage, which map nicely to the different ‘screens’ we have in our application. The setup stage takes place before the host player starts the game, which is the time period where more players can still join the game. Once the host player presses <ENTER> to start the game, the game moves to the next stage, which is the question stage where players receive question prompts that they must answer.

The move from question stage to voting stage is significantly more intricate than from the setup to question stage, since there are two different events that should trigger this switch: 1) if the primary node has received answers from all the running secondary nodes and has themselves responded to all prompts or 2) if the primary nodes timeout while waiting to receive answers. If one of the players running on a secondary node has submitted all of the responses to their questions but neither of the two conditions described above have been met, the player will continue seeing the question screen until the change to the scoring stage is triggered on the primary node. Once this occurs, the player will see the voting screen and will be presented with one set of question prompts and answer at a time to vote on. There will always be 3 options for the player to vote for, two of them are the answers submitted by the players who were assigned that prompts and the third one is chatGPT’s response to this prompt.

The change from the voting stage to scoring stage is very similar to the change from question stage to the voting stage. The same two conditions apply for the change to be triggered: 1) if the primary node has received votes for all questions from all the running secondary nodes and has

themselves voted for all prompts or 2) if the primary nodes timeout while waiting to receive votes. The only difference between this transition between stages and the previous one is that when a node (primary or secondary) has submitted all of their votes, they will see the scores waiting screen while the two conditions listed above are not met. Once the either one of the conditions are met, the primary node switches to the scoring stage and notifies all of the running secondary nodes. The tallying of the votes then takes place at the primary and secondary nodes independently and the final scores for the game is displayed.

We rely on **conditional variables (CV)** to communicate between the different threads in our node that certain conditions were met.

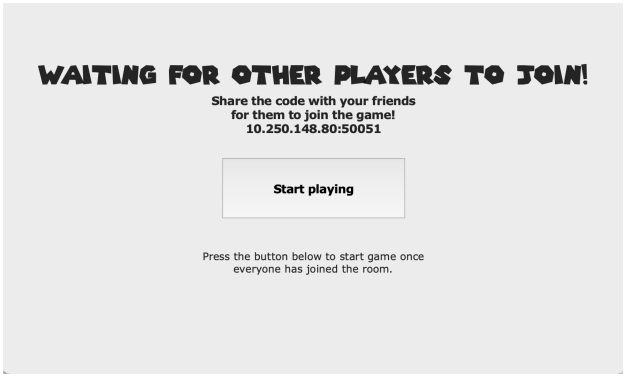


In particular, for each change of stage, we use a pair of conditional variables that signals the change of stage.
At the **primary** node, the conditional variable is set to True whenever all the expected answers/votes have been received from all clients OR whenever the timer (for waiting for answers in the Client thread) times out. This triggers the GRPC thread in the **primary** to send a notification to all **secondary** nodes to move to the next stage.
When the secondary nodes receive the notification, the GRPC thread of the **secondary** node sets the CV to True, which indicates to the Client Thread that it can move to the next stage.

Challenges encountered

Integrating the application with Tkinter UI

One of the greatest challenges we faced during this project was integrating a Tkinter UI we designed for the game with the code that runs the logic game. For the Tkinter version of the code, we relied on the **asynchronous** GRPC API (since Tkinter does not tend to work very well with threads), imported the servicer into the Tkinter code and initialized the GRPC server from within the Tkinter application, as a task that we add to the **mainloop()** function of Tkinter, which runs the event loop. The challenges we encountered were in great part due to the fact that our P2P Net implementation does not have explicit client or server classes, which forced us to add a lot of the game logic into the UI. Our current version of the application with the Tkinter UI is not functional due to an unresolved issue with the GRPC NotifyPlayer requests that go from the primary node and should be processed by the client side of the secondary node. We attempted to fix this issue by making using **await** statements within **async** methods for all the GRPC requests; however, we could not resolve this bug in the implementation and due to time constraints, we moved forward with the terminal-based version of the game. Nevertheless, we learned a lot from the attempt to integrate with Tkinter and from using the grpc.aio as oppose to only relying on the threading library and having the GRPC server running on a separate thread.[2]



[2] The buggy code for the Tkinter integration is included in a separate folder in the repo. Over the next couple of days, we will try to resolve this issue (for our own satisfaction).