PaperProto
Final Project Engineering Notebook

Kamryn Ohly + Grace Li

👾 Try our Project: https://www.paper-proto.com/

Links

- **Paper-Proto:** https://www.paper-proto.com/
- **Github:** https://github.com/kamrynohly/PaperProto
- **Video Demo:** https://drive.google.com/file/d/1QYby1SO8hx94wd5iCNFqiCeh3jX2pSrv/view?usp=sharing
- **SEAS Design Fair Poster:** https://drive.google.com/file/d/1nO0DCGHarU7Hm7bsD-tWlRl9FhPnq-gv/view?usp=sharing

Disclaimer on Project Change

We want to start with a brief note that our final project for CS2620 has changed since we originally submitted our proposal. We originally planned to create a project highlighting a curiosity we had about the airline industry, except after much research and discussion with Professor Waldo, we came to the conclusion that we would not be able to meaningfully implement our original idea. Thus, we excitedly pivoted (thank you to Professor Waldo for allowing us to pivot!) to implementing a multiplayer gaming functionality on top of a project that we started recently called PaperProto. In our introduction, we will outline the new project goals that we set, followed by our implementation and learning experiences.

Introduction

Our project, PaperProto, transforms simple sketches and descriptions into fully functional games through the power of AI. PaperProto serves as a bridge between imagination and implementation, allowing anyone to rapidly prototype game ideas without coding knowledge.

While PaperProto was created to make single-player games, we saw an opportunity to explore an exciting, complex distributed systems challenge by extending its capabilities to multiplayer game generation. Adding multiplayer generation required solving fundamental distributed systems projects such as handling state synchronization, real-time communication of game updates, and handling many "unknown unknowns".

For our final project, we implemented a distributed architecture that enables PaperProto to generate and host live multiplayer games. At the core of this system is a server component that handles communication between multiple clients, maintains authoritative game state, and handles the complexities of concurrent player interactions.

The technical challenges were considerable: our server had to dynamically adapt to completely different game types generated on-the-fly, maintain consistent state across variable network conditions, and handle the unpredictable behaviors of both human players and AI-generated game elements. We implemented efficient serialization mechanisms, conflict resolution strategies, and optimized network protocols to ensure responsive gameplay despite the inherent latency challenges in distributed applications.

By addressing these distributed systems challenges, we've expanded PaperProto beyond its original vision as a prototyping tool into a platform that demonstrates how creative AI can power social gaming experiences through sophisticated distributed architecture.

The Start of PaperProto

Origins and Inspiration

PaperProto was born at the Anthropic x Harvard hackathon. While we struggled to think of an idea to create, we couldn't help but notice something: our friends all around the room were sketching their ideas on whiteboards as a form of rapid prototyping. This sparked our curiosity about how we could elevate the traditional paper prototyping process into something more interactive and immediately functional.

We chose to focus on game development specifically because of our passion for creative expression in gaming and our recognition that traditional game development has a notably high technical barrier to entry. Our goal was to lower the technical barrier to game creation by allowing anyone to transform a simple sketch or description into a playable game - no coding required.

Technical Implementation

For our tech stack, we selected Next.js with JavaScript using the app-router architecture, providing us with a modern, efficient framework for building our web application. Our development process began with creating the core UI components for our project:

1. A login/authentication system to keep track of user data
2. A profile page where users could view and manage their created games
3. A creation page featuring Claude integration for generating new games
4. A community page showcasing all published games that could be played directly in the browser

We built these components with Claude, as part of the hackathon, to design the UI aesthetics, settling on a "retro gamer chic" style that reminded us of when we played online video games during middle-school.

Data Infrastructure

For our database needs, we originally implemented Firebase to handle both user authentication and game data storage. The Firebase authentication system supports both email/password login and Google account integration, creating user profiles that track essential information like the following:

- Email address
- Username
- Game count
- Like count
- Project IDs (linking to created games)

Our game data structure also stores comprehensive information for each game such as:

- Title and description
- Creator ID and username
- Game UUID
- Play count
- Generated game code
- Like count
- User ratings
- Profile image URL

We also created utility functionalities to easily communicate with and fetch data from our Firebase Firestore database.

Game Generation with Claude

The heart of PaperProto lies in our integration with Claude's API for code generation. We invested significant time in prompt engineering to ensure Claude could consistently produce functional, playable games from user inputs. This required careful consideration of constraints, game mechanics possibilities, and output formatting.

We also wanted users to be able to immediately iterate upon their generated games and test them, so we created a way to play the games within the browser. Our "GameDisplay" and "GameDisplayMulti" components act as containers for the singleplayer and multiplayer HTML/CSS/JS games generated by Claude, respectively. These game codes are rendered within an iframe that allows users to play directly on our platform. This required solving several technical challenges around sandboxing, resource loading, and responsive design.

Community Features

The community tab serves as a showcase for all published games, featuring filtering options to help users discover content. With the implementation of reviews & favorites, anyone can also show support and share their favorite games with others!

By the end of the hackathon, we were thrilled with the state of PaperProto, yet we knew that we wanted to continue working and iterating. In particular, we tried to implement a multiplayer functionality into the game purely with AI, yet it proved to be far too complicated without deeper investigation and planning. For these reasons, we decided to build a multiplayer architecture for PaperProto for our final project!

Approaching a Multiplayer Architecture with Generated Code

Approaching this project, we knew that we had many interesting challenges to navigate. In particular, while we could visualize how to approach building a singular multiplayer experience, the uncertainty of having an unknown number of LLM generated, containerized games was difficult to approach. We decided that we needed to create an architecture that was as deterministic as possible in its server to client communication to offset the uncertainty of client to game communication.

We began by constructing our flow of information. We decided that the server would hold onto all information about the current state of games, the active games, and which players were participating in each game. Since we already store user information and game information in Firebase Firestore, we decided to make the server exclusively responsible for controlling multiplayer interactions, although we may eventually switch all of our data to be stored on the server. The server would be able to stream game state updates to subscribed clients for a specific game session.

The client (the website) would receive updates from the server as well as forward messages to the server. We decided early on during our design of PaperProto that the client would be the only point of communication with the server. We refuse to allow the subclient (the game) to communicate directly with the server. Instead, any updates in the game would be passed to the client through an event listener held by the client, ensuring that the game remained containerized. **By refusing to allow the subclient (the game) to communicate directly with the server, we could better protect the integrity of both the client and server from bugs or malice caused by the generated code**.

Lastly, the client-subclient communication is vital. We decided that when the client passes game state updates into the game container, then the game would then be responsible for parsing the data, as well as updating its own state. This parsing and implementation would need to be generated on the creation of the game, which we knew would be a technical challenge. However, this is a necessary evil to permit customizable, varying games to be created.

In order to allow for this rigidity, yet open-endedness, we focused heavily on constructing the right types of messages to be sent from the client to the server. We could have taken two different approaches:

#1. A gRPC-Inspired Approach
When working with gRPC, a developer can define a proto file to create messaging protocols that are then implemented on both client and server. This is powerful, yet has many potential pitfalls. In particular, though a developer could specify functionalities, it would rely entirely on others to create implementations. If these implementations were not completed, then the system would experience integrity issues. On the other hand, if the proto file did not specify enough functionalities, then the system may not be able to communicate enough information between the client and server as it needs to be successful.

This double-edged sword is one that we considered while brainstorming designs for game updates. We thought that we could specify a long list of potential game functionalities (pointGained, pointLost, playerMoved, etc.). If we created these protos, then we could instruct the LLM to complete the appropriate implementation for the game specified by the user. However, this presents two major problems. In a typical utilization of this gRPC architecture, it would be a developer's fault if they forgot to finish one of the implementations. **Generating these implementations adds another layer of complexity.** If these implementations are LLM-generated, there is a high chance that it fails to implement them exactly, thoroughly, and accurately. Likewise, we still encounter the other issue, where the protos may not be comprehensive enough, limiting the games that our platform could build. Overall, we felt as though this approach would need to be more generalized in order to support our goals.

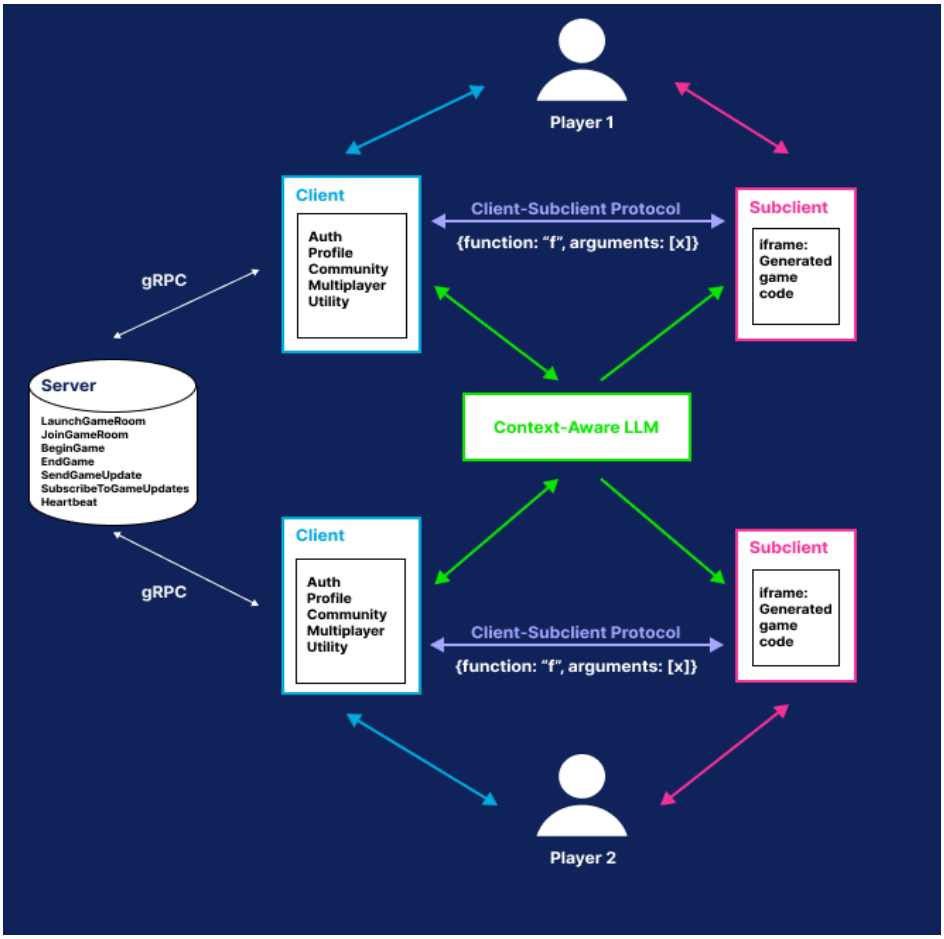
#2: A Logging/Replication-Inspired Approach
We also considered a different approach. We thought about how we could get replicas of servers up-to-date by replaying logs made by the other servers. A replica would need to know how to parse and replay these logs, yet the logs could convey enough information to ensure that the replica would be accurate and could communicate a variety of changes. A downside of this approach would be that parsing can be quite intensive, and it is critical to ensure that logs are properly ordered. If logs are not ordered sequentially, then changes could be overridden. The distributed systems problem of sequencing is also non-trivial, and can lead to slower system times to get a system to be immediately or eventually-consistent.

We found this architecture of updating replicas with logs to have many aspects that we liked for PaperProto! In particular, the open-ended nature of logging could allow more information to be transmitted between a client and server, and the replay ability lends itself well to gaming. **We thought about how chess games are often logged by moves and could be replayed afterwards.** However, we also worried about the cost of sequencing our logging and the technical

complexity of parsing logs. Lastly, handing over the data logging to generated code makes it far more open-ended then specifying existing functionalities.

We came to a conclusion that we would attempt to draw from both of these distributed architectures to build PaperProto. We would follow the idea of parsing and replaying game updates logs, where the logs would be sent from a client to a server, then redistributed to other active clients in the multiplayer game. We would also utilize gRPC in that we would create a few open-ended messaging formats, such as beginning a game, updating a game, and ending a game. We detail this architecture and approach more thoroughly in our technical implementation, as well as in our code files.

Overall, our architecture looks similar to the following:



Our Technical Approach

Building upon PaperProto from the hackathon, we continued to use Next+JS for our client. We decided to utilize web gRPC for our communication from our client to our server. In our JS client,

we created a utility file called `grpcClient.js` to aid with abstracting our client to server gRPC communication. We then call upon these functionalities within our `GameDisplayMulti` which controls multiplayer game sandboxing. Here are a few examples of handling updates from the client to the server with web gRPC:

```
/**
 * Join Game Room: Send a request to join an existing game
 * @param {string} gameSessionID - The UUID of the game to join
 * @param {string} userID - The UUID of the user joining the game
 * @param {string} username - The username of the user joining the game
 */
export const joinGameRoom = (gameSessionID, userID, username) => {
  return new Promise((resolve, reject) => {
    const request = new JoinGameRoomRequest();

    try {
      request.setGamesessionid(gameSessionID);
      request.setUserid(userID);
      request.setUsername(username);

    } catch (e) {
      console.error('Setting join properties failed:', e);
      // Fall back to direct property assignment
      request.gameSessionID = gameSessionID;
      request.userID = userID;
      request.username = username;
    }

    client.joinGameRoom(request, {}, (err, response) => {
      if (err) {
        console.error('Join gRPC call failed:', err);
        reject(err);
        return;
      }

      console.log("joinGameRoom response:", response)

      resolve({
        status: response.getStatus(),
        message: response.getMessage()
      });
    });
  });
};
```

```
/**
 * Send Game Update: When a game is updated, send a notification to the server of the update.
 * @param {string} fromPlayerID - The UUID of the player making the update
 * @param {string} gameState - A string formatted by the game itself to communicate an update
 * @param {string} gameSessionID - The UUID of the game being updated
 */
export const sendGameUpdate = (fromPlayerID, gameState, gameSessionID) => {
  return new Promise((resolve, reject) => {
    const request = new SendGameUpdateRequest();

    try {
      request.setFromplayerid(fromPlayerID);
      request.setGamestate(gameState);
      request.setGamesessionid(gameSessionID);
    } catch (e) {
      console.error('Setting sendGameUpdate properties failed:', e);
      // Fall back to direct property assignment
      request.fromPlayerID = fromPlayerID;
      request.gameState = gameState;
      request.gameSessionID = gameSessionID;
    }

    client.sendGameUpdate(request, {}, (err, response) => {
      if (err) {
        console.error('Send game update gRPC call failed:', err);
        reject(err);
        return;
      }

      resolve({
        status: response.getStatus(),
        message: response.getMessage()
      });
    });
  });
};
```

In the beginning, we thought that web gRPC and standard gRPC in Python would be able to communicate directly and identically - and we were quite wrong!!! While the setup is the same between our proto files on the client and server end, we encountered many issues in bridging web gRPC with our Python server. For many reasons including compatibility issues and handling HTTP/HTTPS variations, we learned that it is necessary to have a proxy server to bridge web gRPC and gRPC. We decided to use Envoy & Docker to create a proxy server that would forward messages from our web client to our backend server. The process of creating this connection was non trivial and definitely a new learning experience for both of us!

After getting our proxy server to connect to our web application and backend server, we built our server in Python with many inspirations taken from our chat design challenges. We utilized the same framework and structure with a different set of proto services and messages, and we were excited to feel quite confident and comfortable with this new implementation. For example, rather than streaming new chat messages, we streamed game state updates in a very similar way. Utilizing gRPC

on the server-side, we also store the current, authoritative game state for each game on the server. At the moment, we have set up a future of being fault tolerant, yet due to the complexity of having LLM generated endpoints, we decided to narrow our project's scope. However, our server currently contains Heartbeat functionalities, which allow clients to ensure connections with the server, setting up a future of fault tolerance.

```
# MARK: Simple Heartbeat
def Heartbeat(self, request, context):
    """
    Handles heartbeat RPC requests to check server status and status of players.

    Parameters:
        request (HeartbeatRequest): Contains the request details.
            - requestor_id (str): The id of the client/entity sending the heartbeat.
            - server_id (str): The id of the intended recipient of the heartbeat.
        context (RPCContext): The RPC call context.

    Returns:
        HeartbeatResponse: The response to the heartbeat request.
            - responder_id (str): The id of the server responding.
            - status (str): Description of the state.
    """
    try:
        logger.info(f"Received heartbeat from {request.requestor_id}")
        return service_pb2.HeartbeatResponse(
            responder_id=self.server_id,
            status="Heartbeat received"
        )

    except Exception as e:
        logger.error(f"Error occurred in Heartbeat request: {e}")
        return service_pb2.HeartbeatResponse(
            responder_id=self.server_id,
            status=f"Error: {str(e)}"
        )
```

Now, one of the most critical technical components of our project was bridging the gap between our containerized games and our client, so that the client could deterministically control sending calls to the server. In order to accomplish this task, we needed to instruct the LLM to generate games that could create game logs, send game logs through an event handler, and finally parse game logs to update game state. Next, we had to create a way for a client to handle these events from the subclient and propagate the messages. We accomplished this with a substantial amount of prompt-engineering, and through leveraging `postMessage` functionalities of iFrames in our `GameDisplayMulti.js.` Our current prompt for our multiplayer game generation is stored in `Prompts.js` within the client folder.

Obstacles

We faced many challenges and obstacles to overcome in the creation of our project. Many of our challenges involved bridging the uncertainties of generated code endpoints with the highly

structured nature of communicating between clients and our server. We wanted to highlight a few of the key challenges we encountered, coming from the poster that we also presented at the SEAS Design Fair:

Communication with a Containerized Game Environment. AI-generated code requires containerized execution to prevent potential security and system integrity vulnerabilities. This sandboxing introduces communication complexity, as direct server-game interactions would be optimal for performance but problematic. Our architecture must balance security requirements with latency concerns while managing communication from the container to the client prior to the server.

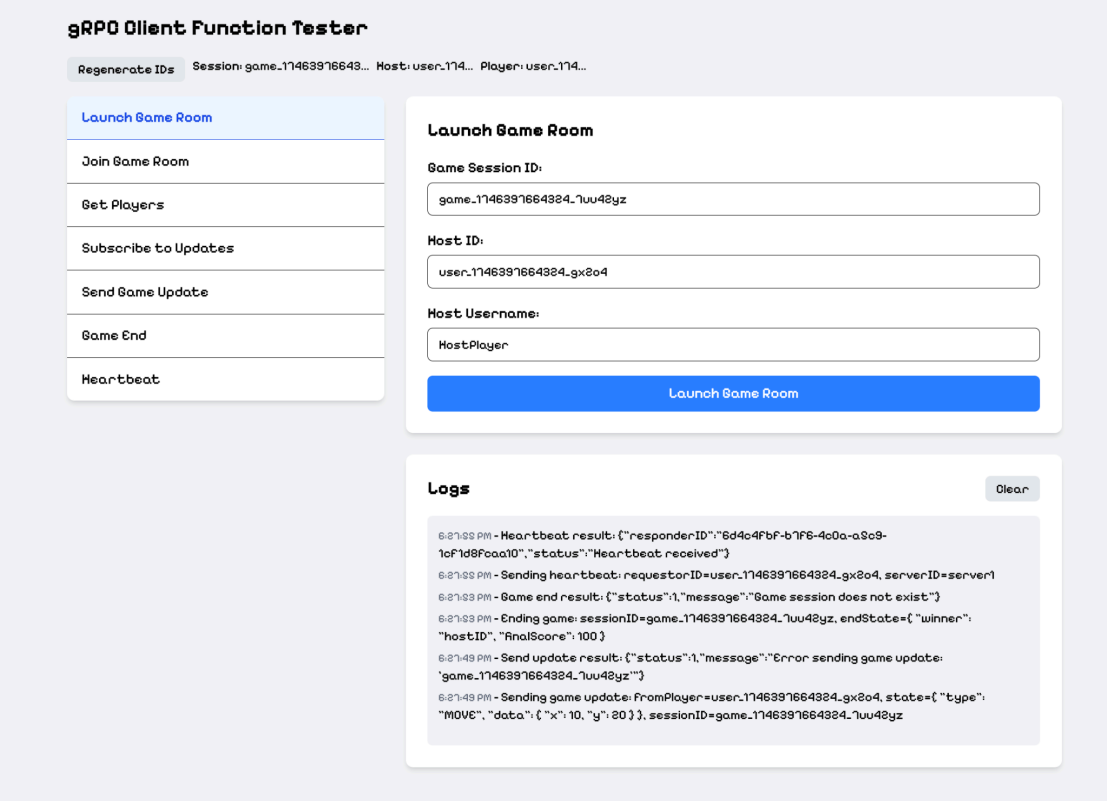
Protocol Design: Flexibility vs. Standardization. Designing communication protocols proposes the challenge of balancing flexibility and consistency. Our gRPC-based protocol framework must be restrictive enough to standardize communication between games, yet robust enough to permit a wide variety of game play mechanics. We've attempted to address this by implementing a minimalist yet open-ended protocol structure - using session ID, user ID, function name, and JSON-formatted arguments.

Protocol Implementation with Stochastically Generated Code. Once we have designed a communication protocol, we must engineer a verifiable system that ensures the generated game code follows the protocol. We must both minimize the probability of failure, as well as implement a feedback loop to ensure the game code is regenerated correctly.

Game State Synchronization. The most critical element of a game is managing game state. When the difference between one point can make-or-break the game, the management of game state is vitally important. With synchronization being a key area of challenge with distributed systems, the ordering of events and timing across different processes is a non-trivial task. We leverage the use of logical clocks and verification systems on the client and server-side to ensure a consistent gameplay experience.

Tests

Due to the LLM-generated nature of our games, testing PaperProto is quite a challenge. While we expect a non-zero percentage of our games to not work, we aimed to create a robust enough architecture that buggy or malicious generated code will not compromise the integrity of our client nor our server. In order to test our connection to our server from the client, we started by actually creating a UI interface where we could create mock data and see if we received what we expected. While this is less of a formal test, it is robust and allowed us to attempt to pass malicious code/inputs, where we could verify that our client and server would not fail. You can find this full integration testing at https://www.paper-proto.com/grpc-test.



In addition to this full integration test, we implemented a number of server-side unit tests to validate PaperProtoServer's core functionality. We tested as many edge cases as possible, including room creation, player management, real-time update streaming, and session termination. In conjunction with the aforementioned integration tests, we have done our best to test this non-deterministic system in a comprehensive manner.

Client/Server Hosting

This was an unnecessary (and very painful) element of our project that we wanted to explore! We wanted to share our project with others, as we were incredibly proud of our results. However, we knew that we could not leave our proxy server and python server running on our laptops indefinitely. Thus, we decided to use Vercel to host our client, and AWS EC2 to host our proxy server and python server. Setting up our infrastructure with running the proxy server and the python server in EC2 was quite challenging. One of the biggest challenges was fighting against "Mixed Content" errors, since our Vercel client uses HTTPS, yet EC2 instances by default use HTTP. Thus, we had to connect our proxy hosted on AWS to our domain: proxy.paper-proto.com. **After many hours of debugging errors with TLS handshakes, errors with docker containers, and needing to reboot EC2 and reconnect via SSH, we were able to successfully host our full project on https://www.paper-proto.com/. We would love for you to take a look!**

Conclusion

Thank you so much for reading about and taking a look at our project. We had an absolutely amazing time building PaperProto and creating a new architecture that supports multiplayer game play & generation! We hope that you enjoyed it, and **thank you for a wonderful semester.**

AI Disclosure:

There is a LOT of AI being used in our project, including in the creation of it! **One key use of AI is that our games are completely generated by Claude :)**

During the hackathon, we generated most of our UI with the help of Claude, and we also utilized Claude to assist with the style of our site to make it cute & fun!

For our distributed systems components, we did *not* use AI for the designing of the architecture. We did use it to assist with the technical challenges of creating a proxy server and understanding how to bridge web gRPC and our Python server.

We also used AI to help us set up our AWS servers and to help with some of the complexity of bridging HTTP & HTTPS across domains.

Lastly, we wrote down most of our engineering notebook as bulleted thoughts/problems we encountered as we went, so we utilized Claude to help us format these into more formal thoughts, as well as consolidate a lot of our process to meet the page restriction.