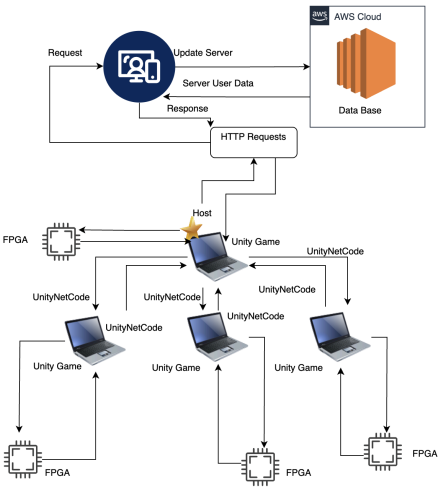
Information Processing Report Group 11

The purpose of our system

The purpose of our system is to facilitate a client-server solution for a multiplayer dogfight game with FPGA controllers. The system has multiple nodes that can process data, including that captured by an accelerometer, and can interact with a cloud server in order for information to be exchanged.

The architecture of our system



Our system uses a Peer-to-Peer model for the game communication, with a master node acting as a server for the game itself as well as a proxy for communication with the database.

Design Decisions

FPGA

During the development of the project several design decisions were made around the implementation of the FPGA in order to meet the system requirements.

We first took the decision to assume that the FPGA had no flashed hardware and no software downloaded at the start of each game session. This is so that no user preparation would have to take place prior to entering the game, enabling them to use any DE10-Lite board. This led to the development of the FlashFPGA.py and NiosClient.py scripts to flash hardware and download/run software respectively. FlashFPGA.py uses code found from the Quartus II Scripting Reference Manual and NiosClient.py utilises a mixture of the Nios II Command Shell though a Python subprocess and the intel-jtag-uart Python Library. The reason we chose to use this mix was because after testing we found that the Command Shell could download the .elf file (containing the software) onto the FPGA much faster than the library. However the Python library was robust, much easier to use and had better built in error handling than the Command Shell.

We also decided that as part of the project the FPGA would need to be able to be capable of simultaneous two-way communication with its respective computer. This is so that the FPGA state could be updated in real time at the request of the game while also streaming its accelerometer data back to the computer to control the in-game plane. At first, we looked into the possibility of using two scripts, one to collect the accelerometer data from the FPGA and one to send state update requests. We found that this was not possible, as only a single program on the computer can use the jtag for communication at any one time. The decision we made to resolve this was to merge the two scripts into one and run them concurrently using Python threading so that data could be streamed from the FPGA and state updates could be sent in the same program .

The software that ran on the FPGA came with its own set of design decisions, one being how the FPGA would parse and validate incoming update requests. We decided to wrap incoming state requests in a header and footer to differentiate them from anydata the FPGA might detect on the jtag, giving each type of state request a different header to be parsed by the FPGA. To detect incoming requests we initially looked to implement the alt_getchar() function used in lab 4. This function proved capable of detecting incoming characters to be parsed but halted the program at each use. This presented a problem as it would prevent the FPGA from executing the rest of its code (like sending the controller accelerometer output). To resolve this we opted to poll the jtag buffer register directly using the IORD() function which would allow the program to continue executing if no requests were detected at the time. One consideration we had implementing this approach was that the frequent polling in the program could lead to the execution of our program slowing down. After testing this however no difference in speed could be detected.

```
private void StartServer()
{
    try
    {
        // Create a socket object for receiving from Nios
        IPAddress niosReadAddr = IPAddress.Parse("127.0.0.1");
        int niosReadPort = 49152;
        niosReadCon = new TcpListener(niosReadAddr, niosReadPort);
        niosReadCon.Start();
        Debug.Log("Waiting for a connection from Nios...");

        niosReadClient = niosReadCon.AcceptTcpClient();
        Debug.Log("Connection established with Nios: " + niosReadClient.Client.RemoteEndPoint);

        // Create a socket object for sending to Nios
        IPAddress niosWriteAddr = IPAddress.Parse("127.0.0.1");
        int niosWritePort = 49153;
        niosWriteCon = new TcpClient();
        niosWriteCon.Connect(niosWriteAddr, niosWritePort);
        Debug.Log("Connected to Nios");

        // Start a thread for sending to Nios
        sendThread = new Thread(Send2Nios);
        sendThread.Start();

        // Start a thread for receiving from Nios
        receiveThread = new Thread(ReceiveFrmNios);
        receiveThread.Start();
    }
    catch (Exception e)
    {
        Debug.LogError("An error occurred while starting server: " + e.Message);
    }
}
```

The final aspect of integrating the FPGA was allowing unity to read the accelerometer data. This was done by adding functions to the existing player controller game scripts to process what was streamed to unity. The code snippet above creates a webSocket for both receiving data from and transmitting data to the FPGA. The accelerometer and button data received from the FPGA is then (in the SetFPGAData function) converted from x,y,z coordinates into vertical, horizontal, and yaw values that are used to fly the plane and if the buttons are pressed then the functions to shoot bullets or accelerates the plane are called. The initial plane tmiController script was provided from the Unity Simple Airplane Controller Asset and was modified to allow our FPGA to be the controller.

Performance metrics

Sampling rate - The speed at which the accelerometer data is sampled and sent across the JTAG. This should be maximised.
Filter latency- The latency due processing of each filter tap. This should be minimised.
Reliability - The quality and consistency in the initialisation process and during use

Testing

Testing for the FPGA controller was done in a practical fashion. The criteria for success was to have the controller running with a high enough sampling rate and a low enough filter latency to have an indiscernible responsiveness and to initialise without error every run.

The sampling rate was found to be 1000 samples per second and provided a satisfactory responsiveness. The filter latency was minimised at 16 taps while maintaining an appropriate smoothing and low filter latency. During testing our system initialised successfully 14/15 times. As this did not meet our success criteria exactly, implemented in detailed error messages to ensure that the reasoning for initialisation failure was fully understood.

We also put the system under testing during initial integration with unity. We did this by tracking the number of valid packets that were received on the Unity side of the project. We aimed to have 80% of the packers received to be valid so that the Unity system had enough accelerometer data to process. Averaged over 30 seconds we average a rate of 92.53% valid packets, therefore meeting our criteria.

Resources utilised for the DE10-Lite from Quartus

- Accelerometer
- Switches SW[0:9]
- Buttons KEY[0:1]
- LEDs LEDR[0:9]
- 7 segment displays HEX0 - HEX5
- JTAG UART
- On_chip_memory

Game design

Multiplayer netcode

When designing the multiplayer aspect of the game, there were a number of resources that could have been used, thus ultimately it became a matter of evaluating what the benefits of each different method were in order to determine which would be most suitable for this project. In addition to unity's netcode library, we considered using 3rd party networking packages such as mirror and photon. Both of these packages allow for the creation of host servers to run the game on and even allow you to host games on dedicated master servers which avoids having one machine be a host and a client. While this approach would have been more performant, given the fact that multiplayer is only possible over local networks, having the extra performance from the network was deemed unnecessary.

The primary advantages of using unity's own netcode library were as follows, because it is unity's own proprietary library there exists more in-depth documentation which was a significant factor in this case as all members of the team were relatively inexperienced using unity and thus the extra availability of information was very helpful. We also did not need some of the extra features and functionality of the 3rd party libraries, though they may have had more specific tools which would make them better in specified instances, the more simplistic nature of our project meant that these too were not necessary and thus, the more lightweight unity netcode became the more efficient solution to implement. Furthermore, Unity's netcode library made certain aspects of coding the game's functionality simpler and more effective than using 3rd party libraries. This included the availability of sample projects to easily learn and understand how to practically use the library as well as easy import of dependencies across systems, which was important when working on a single project from multiple systems. Ultimately it was determined that due to the advantages mentioned above, unity's netcode library was the best option to develop the netcode of our game.

Initial Game Design

We began with a simple game design, where the aim was to get multiplayer functioning and after that, began adding more functionality to the game and improving the design.

Unity's inbuild networking package includes a network manager script which is able to handle basic running of different tasks that facilitate local network multiplayer functionality, including initialising the host server, managing client connections and disconnections from the host server, as well as dealing with any network errors that could occur during runtime.

Another crucial task which the network manager performs is synchronising various network objects so that they are the same across all instances of the game for all players. To do this, the script is told which axes it must care about, it serialises the transform information in those axes, and sends it to the other clients connected to the server who can then deserialize it and are able to apply the appropriate values to the appropriate game objects.

Performance metrics

Performance of the server was assessed in a practical way checking to see if the user experience was acceptable, based on factors such as ping and packet loss

Testing

Testing for the netcode was done practically. It was undertaken in steps as outlined below. The criteria for success was to have the game running with the desired functionality while having acceptable operational standards (latency, packet loss).

- Test whether we could instantiate a host server
- Test whether we could successfully connect a player client to the host server from the same machine
- Test whether we could then connect a separate player in a different game instance running from the same machine
- Test whether we could connect a player running a game instance on a different machine using the local ip address
- Test to ensure that the positional data for both players is synchronised across both game instances

These tests were conducted on the network and it was able to pass them successfully and show that our game was functioning. Overall, latency did not prove to be an issue as we had relatively few objects and players within the system combined with the fact our game was run across a single local network.

Random Position Spawner

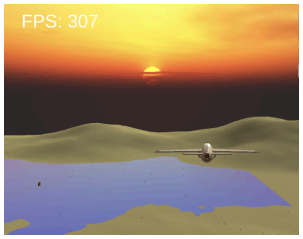
```
using System;
using System.Collections.Generic;
using Unity.Netcode;
using UnityEngine;
using Random = UnityEngine.Random;

[RequireComponent(typeof(NetworkManager))]
public class RandomPositionPlayerSpawner : MonoBehaviour
{
    NetworkManager m_NetworkManager;

    int m_RoundRobinIndex = 0;

    [SerializeField]
    SpawnMethod m_SpawnMethod;

    [SerializeField]
    List<Vector3> m_SpawnPositions = new List<Vector3>() { Vector3.zero };

    /// <summary>
    /// Get a spawn position for a spawned object based on the spawn method.
    /// </summary>
    /// <returns>The spawn position.</returns>
    public Vector3 GetNextSpawnPosition()
    {
        switch (m_SpawnMethod)
        {
            case SpawnMethod.Random:
                var index = Random.Range(0, m_SpawnPositions.Count);
                return m_SpawnPositions[index];
            case SpawnMethod.RoundRobin:
                m_RoundRobinIndex = (m_RoundRobinIndex + 1) % m_SpawnPositions.Count;
                return m_SpawnPositions[m_RoundRobinIndex];
            default:
                throw new NotImplementedException();
        }
    }

    private void Awake()
    {
        var networkManager = gameObject.GetComponent<NetworkManager>();
        networkManager.ConnectionApprovalCallback += ConnectionApprovalWithRandomSpawnPos;
    }

    void ConnectionApprovalWithRandomSpawnPos(NetworkManager.ConnectionApprovalRequest request, NetworkManager.ConnectionApprovalResponse response)
    {
        // Here we are only using ConnectionApproval to set the player's spawn position. Connections are always approved.
        response.CreatePlayerObject = true;
        response.Position = GetNextSpawnPosition();
        response.Rotation = Quaternion.identity;
        response.Approved = true;
    }
}

enum SpawnMethod
{
    Random = 0,
    RoundRobin = 1,
}
```

The code above allows us to provide fixed spawn points all around the map, and randomly spawn at one of those positions in the vector of spawn positions provided or use a round robin approach for spawning.

In the Awake function (initialises the game state before starting), we then call the ConnectionApprovalWithRabdomSpawnPos function, which handles the connectional approval process, successfully spawning the plane in a random position when called.

Performance metrics

- Game FPS: The FPS of the game stays between 200-400 fps
- Game control: Whether the player could successfully control their avatar
- Game feature functionality: random positional spawning



Libraries Used:

- UnityNetCode for GameObjects
- Simple Airplane Controller Asset
- Fantasy SkyBox
- Terrain Skin

Testing

- Ran test build of the game to determine performance of the game
- Tested controlling the player jet with keyboard to test if the controller script allowed player movement control
- Tested controlling the player jet using accelerometer on FPGA
- Tested rerunning the game to see if the spawn position was randomised each time

AWS

Choice of SQLite3 for Database Management Over DynamoDB:

1.SQLite3 is easy to set up and doesn't require the overhead of managing a DynamoDB instance. This is particularly beneficial in the early stages of development or for projects with limited resources.
2.SQLite3 is free and runs locally, eliminating the costs associated with DynamoDB, such as read/write throughput provisioning and storage costs.
3. For initial testing and small-scale applications, SQLite3's performance might be entirely adequate. It also simplifies the development process, allowing for rapid prototyping without worrying about database scaling or management.
4. If a game's data interactions are primarily transactional and relational (e.g., updating scores, retrieving leaderboard rankings), SQL-based databases like SQLite can be more straightforward to work with compared to the document-oriented structure of DynamoDB.

JavaScript was a natural choice for web-based game development, enabling dynamic interactions between the client and the server. Using JavaScript facilitates the development of a responsive user interface for the leaderboard and the game itself, allowing for real-time updates without needing to refresh the page.

Hosting the server on an AWS EC2 instance provides the flexibility to scale resources as the game's popularity grows. The choice of location (Northern Europe, Sweden) for the EC2 instance was likely made to optimise latency, ensuring that the round-trip time for messages is within acceptable limits for a real-time gaming experience.

The decision to use both HTTP and WebSocket protocols is strategic for balancing efficiency in communication. HTTP is used for standard client-server communication, such as querying the leaderboard, where immediate real-time interaction is not crucial.WebSocket is utilised for real-time communication between the server and clients. This is essential for live game data updates and interactions, providing a seamless gaming experience by reducing latency and enabling a constant connection between client and server.

Database

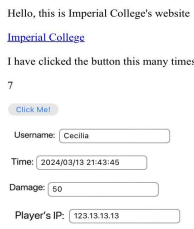
The objective of this project was to develop a server-side solution for a dogfight game that generates and displays a leaderboard at the end of the game and stores the data in a database. The initial solution involves creating a simple web game to finish the initial test without the integration of Unity. The whole design consists of aiohttp for web server creation, SQLite3 for database management, and JavaScript for client-server communication. We first deploy this simple web game on the server in order to test the basic storage and update functions of the created database. Before Unity is completed, the architecture of the long-term storage data module is written in parallel and tested with this simple web game to ensure its efficiency.

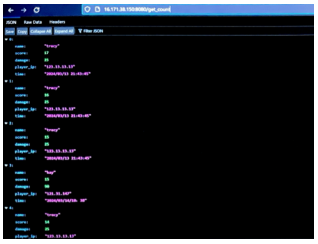
Use of aiohttp for Web Server Creation:
aiohttp is an asynchronous HTTP client/server framework. The decision to use aiohttp allows for efficient handling of simultaneous client connections, which is crucial for games that require real-time updates and interactions. This choice ensures that the server can handle multiple requests concurrently without blocking, making it ideal for the game's leaderboard and interaction mechanisms.

Web Game Interaction and Score Updating Mechanism: The Unity game and database interfaced via a websocket on port 8080, where they could log in and increment their scores by interacting with a button interface. The button clicks were directly proportional to the players' scores. Each new score, along with associated metadata such as the player's username, timestamp, damage value, and IP address, was systematically stored in the database. These parameters were used in the game to provide players in the lobby with information such as the name of the player with the highest overall damage or score from previous games.

Leaderboard Access and Database Validation: The scores were retrieved using the get_count() method (16.171.38.150:8080/get_count) from the updated database. Upon successful testing, we solidified the foundational database structure, confirming the basic functionality's operational reliability.

The database file was stored in an AWS EC2 instance. This allowed for easy access of data from any location. Communication between server and client was through HTTP websockets. EC2 machine located in northern Europe, Sweden, from London to send a message back and forth in the range of 30-35 milliseconds. 1 second can send about 30 messages, but the game generally refreshes the interface at a frequency of 60 frames per second, so the aim was to try to ensure the rules of the game do not have too high requirements for data synchronization.





Server:

- EC2 instance IP:16.171.38.150
- username:ubuntu
- Password:87989888
- Port:22

Tools and technologies used:

- Python: For server-side scripting, including the handling of HTTP requests and database operations.
- aiohttp: A Python library used for creating a web server capable of handling asynchronous HTTP requests.
- SQLite3 is a lightweight, disk-based database used for storing game scores and other related information.
- HTML and JavaScript: Used for creating the webpage that interacts with the server to display the leaderboard and manage game data.

Implementation Details:

Database Setup and Management:
A SQLite database is named Test.db is created to store the scores, along with additional information such as player damage, player IP, and timestamp. The database class contains all database-related operations, including creating the scores table if it does not exist, inserting new score entries, and fetching scores to generate the leaderboard.

Server-Side Logic:
The aiohttp web server hosts several routes for different functionalities:

- /update_count receives game data from the client, including player name, score, and damage, and inserts this data into the database.
- /get_count returns the top scores from the database in JSON format, limited by an optional max_result_count query parameter.

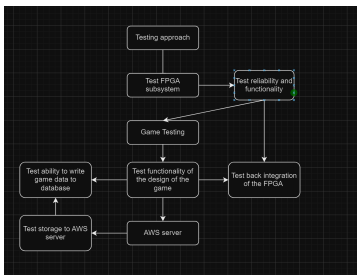
Client-Side Interaction:
The webpage allows players to interact with the game and the leaderboard. Players can start the game, submit their scores, and view the leaderboard. JavaScript is used to handle client-server communication. When a player clicks the button to submit their score, the webpage sends an HTTP request to the /update_count route with the player's data. After submitting the score, players can view the updated leaderboard by sending a request to the /get_count route.

Performance and integration:
The performance and integration between the Python server (utilizing aiohttp and SQLite3) and the client-side HTML/JavaScript code are designed to support a basic but effective leaderboard system for a web-based game.

Explanation of testing:
I uploaded the code server.py to the server to test its functionality:

1. Login to the server by password -by using FileZilla,
2. Then click quick connect and drag all the code inside the ubuntu—server directory into your local machine to start the test
3. Using vscode terminal enter the commands:
    a. ssh ubuntu@16.171.38.150
    b. Enter the password:87989888
    c. tmux a
    d. chmod +x./server.py
    e. ./server.py
4. Then open the simple game webpage to enter different data and click to update: The web page can be tested at http://16.171.38.150:8080/index.html then the updated data will shown in the terminal(method 1 to access database)
5. Change address of the webpage to open the interface 'get_count' to see the database(in Firefox): http://16.171.38.150:8080/get_count (method 2 to access database)
6. To limit the max number of results shown in the table using: http://16.171.38.150:8080/get_count?max_result_count=3 as an example

Overall testing approach



This diagram displays a high level abstraction of the overall approach to testing the system, each individual subsystem was tested separately to ensure functionality within the system and to simplify debugging. Once each system was fully developed we then began integrating them together in order to meet the project goals and create a fully functioning game with FPGA control and AWS integration.

-