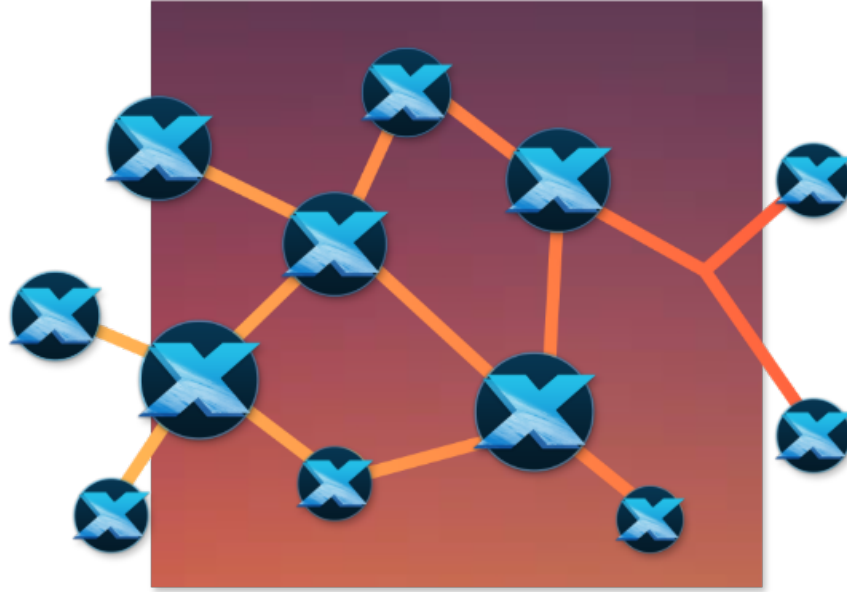


1 Xplane WebRTC



The Xplane WebRTC project takes inspiration from Google Stadia for a streaming based solution for playing video games. But our plan is to build one specifically for flight simulators. We target Xplane 11 for being cross platform and completely offline dependent on sceneries with reasonable documentation on usage of the SDK to use the game. The novelty of the project being all aspects of the project will be completely open source. There are already major segments of the project complete and have been tested which will be mentioned in the appropriate section for the necessary details. I am also investing in my PhD to familiarize myself to work with slim down kernels on HPC scenarios (This will help me extend the Xplane to run on a distributed scenario). This project is fun long term work and has no heavy deadlines but rather the art of optimizing the current paradigm of how we use heavy workload applications. The following above is a really high level abstraction description of the projects and barely covers the depth of what the project intends to push forward.

The big question is what parts have been rebuilt for the project and how they contribute to the end goal of the project. The first property of the project is that it should be able to run on a p2p network. P2PRC is a p2p orchestrator designed to run applications in a p2p network using Containers initially. There are plans to extend it to run on Uni-kernels and based on

my PhD extend it to run on a Multi-kernel paradigm as well. This will be our custom alternative to Kubernetes which will be used to distribute and run on workloads on p2p effectively. This would make our entire run on Anyone's machine which can reside behind NAT.

We also intend to make an open source solution to distribute a slim down version of X Plane so that nodes can quickly Spawn Xplane instantly with only the required scenery needed. This will be in contrast to running the full scenery of the game which is around 55 GB. Something novel that could be worked on here is a novel approach to only send the scenery of the flight path when distributing the game. This will mean building parsers for the Xplane scenery files and then finding techniques to only get a correct set of scenery files needed in other machines. The techniques are expected to open source but since the scenery files are proprietary they are expected to be public.

The streaming part of the project is expected to use the browser standard WebRTC sockets with the corresponding sockets. This is because there is already massive development of the chromium browser with GPU encoders and decoders for faster performance. We have already built a prototype which has been tested and seems to work as intended.

The PhD will be one of a long term novel approach which will support Multi-kernels with TAG based architecture support for running C++ programs more securely. This might mean most parts of the PhD might not be used. The Multi-kernel approach is definitely an interesting area to experiment on to figure out how the project would use such an approach and this open lot of areas of future research and hopefully better performant flight simulators with better purposed algorithm to offload tasks to devices such as FPGAs or potato machines in abstraction layer similar to speaking nodes in an network.

2 Architecture

This chapter dives into the high architecture design of the project and each module is communicated in detail on the following section below.

2.1 Game allocator

The game allocator stores information about the game sessions. This consists of attributes such as:

- Game Session ID <UUID>

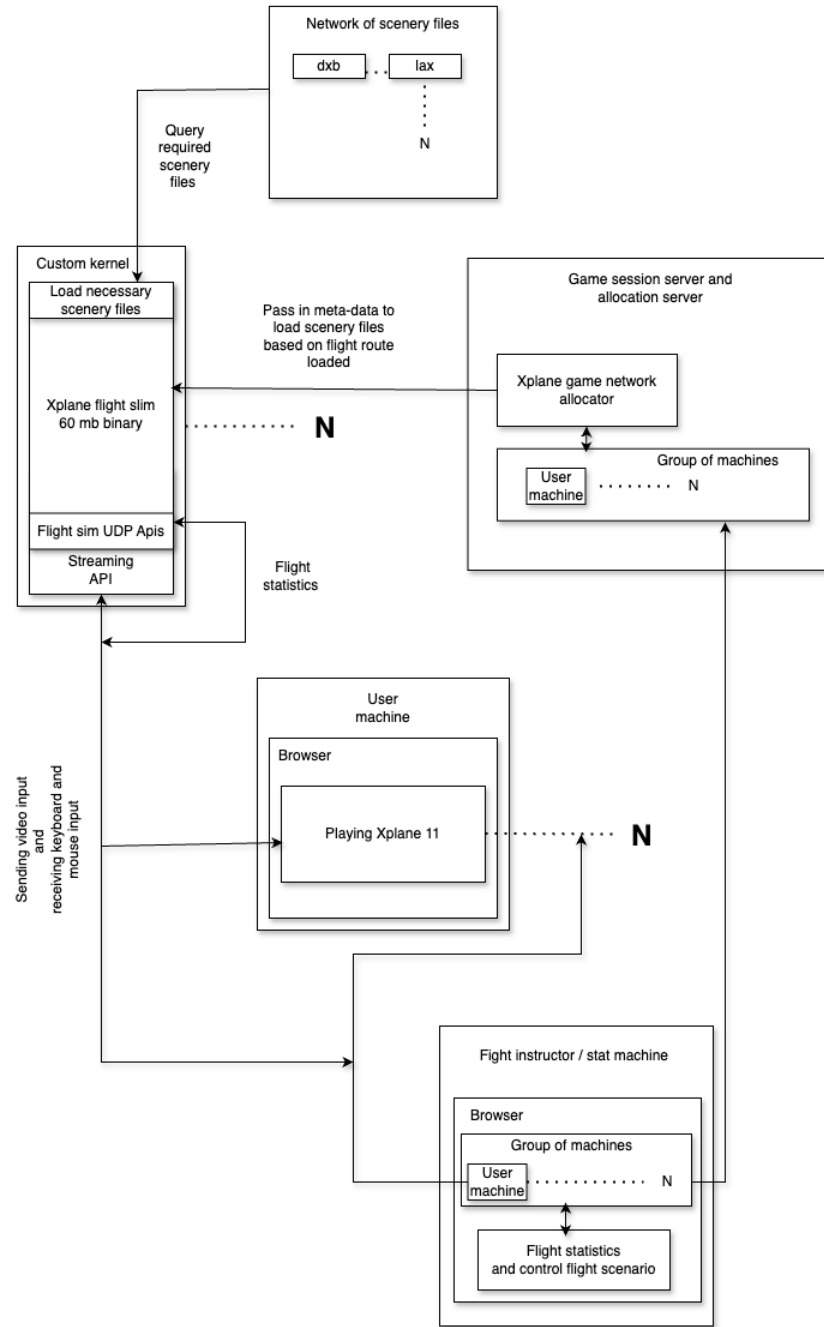


Figure 1: High level architecture of the entire project

```

- Session name <String>
- Nodes Running the game [
    { Rendered Node IP <String>
      Rendered Node Specs <Node Specs>
      Flight Route Path loaded <TBD when network
                               scenery files
                               defined>

      User on Node <UUID>
      Flight Sim API url <String>
    } ..... N ]
- Instructor ID <String>

```

The following above shows a high level data structure for storing session information. A session consists of multiple pilots training with a single instructor. Each pilot is assigned a node to render the game remotely and the instructor can set the scenarios to be trained on.

2.1.1 Interfaces

We will now motivate the higher level interfaces to construct a game allocator this term is inspired from the use of terms like *malloc* and *free* in userspace for allocating memory in a kernel.

```
Instructor = AddInstructor(<Instructor object>)
```

This function creates a instructor in database.

```
Player = AddPlayer(<player object>)
```

Creates a player (i.e trainer) to the database.

```
NodePlayer = AllocateNode(<player object>)
```

Finds a free node variable and adds allocates a player to it based on least latency.

```
FreeNode(NodePlayer)
```

Frees the player from the node. Normally called after the end of the flight session.

```
Node = AddNode(<register node information>)
```

Adds a node that can render the flight sim into the network.

`FreeNode(Node)`

Removes the flight sim render node from the network.

`Session = CreateSession([Player],...n],[Instuctor,...n])`

Create session of players mapped and adds instructors to the session. This function is a high level function that encapsulates *AllocateNode* and maps it to *Instructors*.

`FreeSession(Session)`

Free the entire session created.

3 Render machine

Note: We do not talk about how the game itself is deployed here and we assume that the game is available to execute.

This node is in charge of computing the game in its CPU and GPU. This layer does not distinguish if the game is running bare-metal or on a virtualised environment but rather focuses on game itself is rendered and is pass through the *user machine*. Xplane is called using the binary. locally on the machine and the instructions relating to which window it's running is yet to be decided (This is considered a todo). This section is split into 3 parts:

- Streaming part
- Keyboard and mouse passthrough
- API layer

3.1 Streaming part

The flight sim session is streamed using WebRTC. We will initially hook a chromium browser to detect the screen and over time reduce this to a simple screencapture native program to stream the video feed.

TODO: Specifics to be documented.

3.2 Keyboard and mouse passthrough

We plan to maintain a open source fork of InputLeap. Input Leap is software that mimics the functionality of a KVM switch, which historically would allow you to use a single keyboard and mouse to control multiple computers by physically turning a dial on the box to switch the machine you're controlling at any given moment. Input Leap does this in software, allowing you to tell it which machine to control by moving your mouse to the edge of the screen, or by using a keypress to switch focus to a different system.

TODO: Diagramtic explanation of setup.

3.3 API layer

We use the Xplane API REST server and UDP calls to transmit data back to the *Instructor machine* for further analyses and controlling the flight scenarious. We will implement our own wrapper on top of the Xplane API to create standarised controls no matter the version of the flight sim and we can more fine system control such as new scenery files to pull.

TODO: Internal Xplane APIs to use, Extact routes and relation to transmitted to the instructor server.

4 Network of Scenery files

Not documented until mid 2026.