(Unofficial)

Online Multiplayer

Authored by: chandler14362, pythonology

Download our fork!

https://github.com/pythonology/YARG/releases/tag/online-alpha-v0.2.2

How to use it:

- Simply download the build and connect.
- Supports creating both public and private lobbies (with invite codes).
- Make sure all players in the lobby have the song you want to play.

Service and infrastructure code can be found here:
https://github.com/chandler14362/YARG.Online

Table of Contents

Introduction
Overview
    Screenshots
Networking Model
    Transport
    Clock Synchronization
    Prediction & Rollback
    Packet Types

LAN
Infrastructure
- Lobby Service
- Game Service

UI Design

Introduction

Hello! Let's start by introducing ourselves. We are two passionate rhythm game players that absolutely **love** YARG. In our opinion, it is currently the best RB/GH clone out there hands down. We mostly implemented this feature for ourselves and our friends, but anyone is welcome to use and enjoy it.

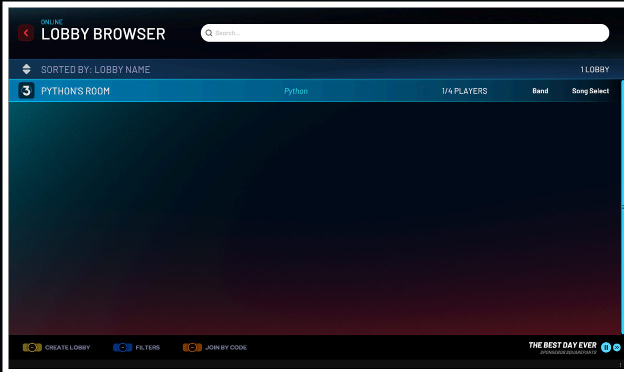
Here are a few relevant projects we've built/maintain to give us some more credibility:

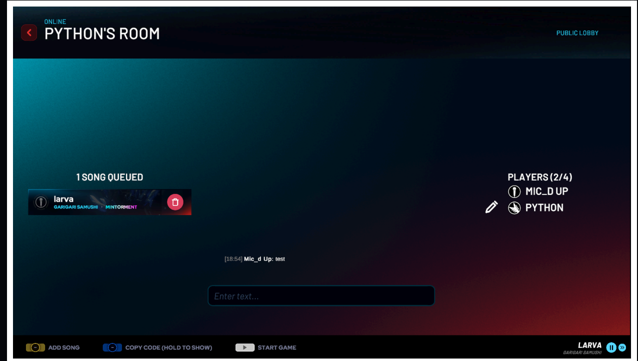
BeatTogether (GitHub): A custom BeatSaber online multiplayer implementation with cross-play between standalone Quest + PC. Widely used by the community for the last 5 years.
Toontown Archive (Website): A preservation project for the late Disney's Toontown Online.

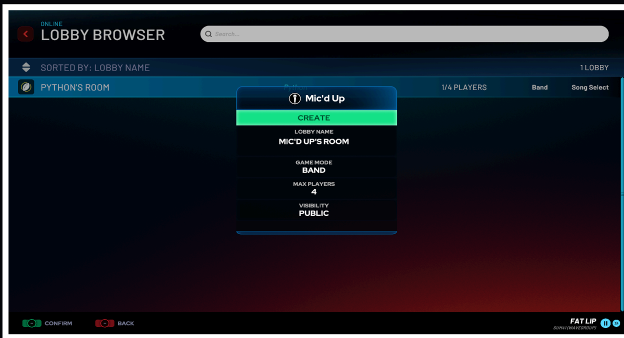
Overview

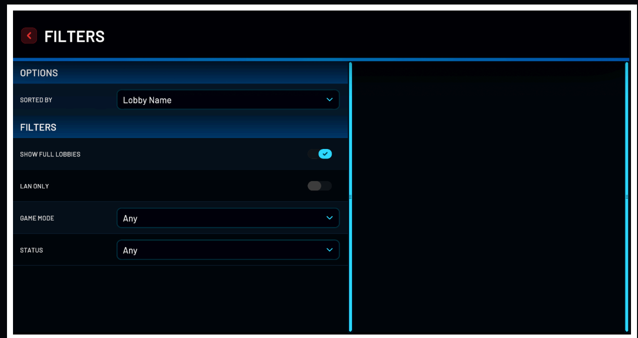
This document is an informal technical design document describing our working implementation of YARG online multiplayer. We're looking for feedback on the approach and are interested in collaborating toward an official integration if possible. Regardless, we intend to host and maintain this as long as it is needed.

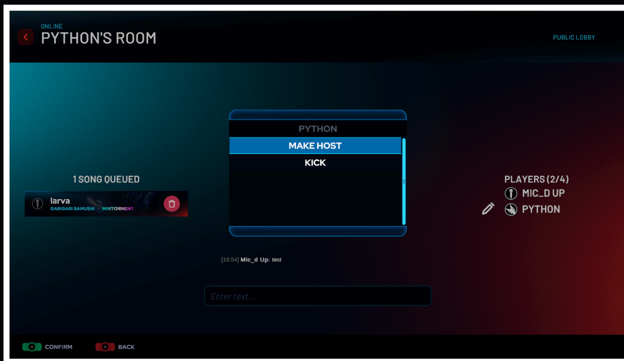
Screenshots











Networking Model

Transport

All gameplay networking runs over UDP via LiteNetLib, allowing for both reliable and unreliable (lower latency) communication. The lobby browser uses WebSocket/HTTP fallback via ASP.NET SignalR.

Players don't connect directly to each other. Instead, all UDP traffic is routed through a custom relay service. This allows for both privacy, and simplicity. The relay service is intentionally left "dumb", with the client handling the majority of the syncing.

Clock Synchronization

Each player's engine runs on their local song clock. To map remote events onto the local timeline properly, and have songs start/end at the same time, the relay server provides a clock syncing mechanism via ping/pong packets. The client measures round-trip time (RTT), and a server time offset is computed.

Prediction & Rollback

The core design goal is making it so remote players feel like they're playing locally. This is extremely fun and interactive, as it feels as if you are playing in a band. A naive approach to fixing the latency problem that comes with this is to visually delay the remote players' track based on their RTT. This works, however, it isn't great. It is quite easy to spot the delay, especially at larger values.

Instead, we've decided to go with a **predictive/reconciliation-based model**. We do this by predicting the remote players' actions, and reconciling when the real result arrives:

1. **Prediction:** When a remote player's note reaches the strike line, the local client guesses whether it was hit or missed based on their recent pattern. If the last note

was a hit, predict a hit. If the last note was a miss, predict a miss. Players tend to streak, so this is accurate the vast majority of the time.

2. **Confirmation:** Remote players broadcast note hit/miss events over unreliable UDP. The result is compared to our prediction.
3. **Rollback:** If the prediction turns out to be wrong, we recalculate the remote player's state from their last authoritative state snapshot, with the new result and future predictions applied. The client will seamlessly update their visual game state to be accurate.

The worst case with this, is a brief visual correction on the remote track when an unexpected miss occurs. This is largely unnoticeable unless you are playing on very high latency. The prediction model will hide latency and feel correct, even up to 200-300ms+ RTT. Alongside this, we also account for network jitter by delaying the next visual note prediction slightly after the prediction mode changes.

Packet Types

Name	Transport	Direction	Purpose
Ping / Pong	Unreliable	Client <-> Server	Clock synchronization
SetLoadout	Reliable	Client -> Server	Instrument/difficulty/engine preset/note speed/modifiers
PeerReady	Reliable	Client -> Server	Player ready to play
GameStart	Reliable	Server -> All	All players ready, start the game
SongMetadata	Reliable	Host -> Server	Song metadata (such as duration for server-side timeout)
GameStartCue	Reliable	Server -> All	Countdown timer
EngineStateSnapshot	Reliable	Client -> Relay -> Others	Authoritative state snapshot of the player's engine

			state (sent periodically)
NoteHit / NoteMissed	Unreliable	Client -> Relay -> Others	Per-note outcome (sent when an actual change occurs from the previous)
StarPowerActivated	Unreliable	Client -> Relay -> Others	Star power activation
Whammy / VocalPitch	Unreliable	Client -> Relay -> Others	Continuous input (whammy/vocal pitch)
GameComplete	Reliable	Client -> Server	Player finished the song
GameEnd	Reliable	Server -> All	All players finished (or a timeout occured)

LAN

We plan to implement a local lobby discovery system for LAN play in the future. This will use the same networking model for now, but we may change it to abuse the low latency.

Infrastructure

We wanted to keep infrastructure both cost-effective to deploy and flexible. Services can be developed locally through just running their code, and Kubernetes is used for orchestration. It is not required to use Kubernetes to develop locally. Pulumi is used to bootstrap the infrastructure in both cloud and local Kubernetes clusters (monitoring, external-dns, etc.) and skaffold is used for deploying services (lobby, game). The current configuration for deploying to Oracle falls completely within the free tier usage. Cloudflare is used for managing DNS, also free. Prometheus and Grafana are used for collecting and

viewing metrics. The cloud deployed Grafana is protected with gated Cloudflare Tunnel access.

Lobby Service

A small WebSocket API with HTTP fallback implemented with SignalR. This service handles authentication, creating/joining lobbies, and any lobby state management outside of gameplay.

Game Service

A UDP relay server that handles routing packets between players in a game session. This service is left intentionally “dumb”, as the core networking model is P2P with a host.

The reason we went with a relay, rather than hole-punching or direct connection is threefold:

1. We don’t want to leak the players IP, especially for streamers.
2. Hole punch can fail. Port forwarding is tedious. This implementation makes it easy for everyone.
3. The predictive model makes latency largely irrelevant, so direct P2P is unnecessary. Even if we used direct P2P, latency is largely based on physical distance from the host regardless.

UI Design

We based the UI on others that exist in the game: namely the music library/popup menus/filters. The lobby UI features a song queue and chat. For private lobbies, we opted for a code-based system to keep things simple, though potentially adding a Discord integration/invite link system would be very cool.