Universidad Europea

MEMORIA DEL JUEGO

“Try to Get Free”

Redes y entornos multijugador

Leonardo David Alpire Villarroel

Cochabamba-Bolivia
Febrero 2025

Índice

Memoria de juego

1. Introducción

El entorno multijugador de "Try to Get Free" ha sido diseñado utilizando Unity Netcode for GameObjects en combinación con Unity Relay Server para proporcionar una conexión estable y facilitar la sincronización en tiempo real entre los jugadores. Se ha optado por un enfoque cooperativo en el que dos jugadores deben colaborar para superar desafíos activando palancas en distintos niveles.

Este documento detalla los aspectos clave del multijugador, incluyendo:

- Variables de red utilizadas.
- Implementación de RPCs y su funcionalidad.
- Motivaciones para la elección del framework y capa de transporte.
- Descripción detallada de interacciones multijugador.
- Flujo del juego y gestión de escenas.
- Objetos de red utilizados y sus componentes.
- Uso de eventos y mensajes en la arquitectura del juego.

2. Variables de red.

Se tienen 1 variables de red, en las “Levers”.

```
[SerializeField] private GameObject leverObject;
public NetworkVariable<bool> isActivated = new NetworkVariable<bool>
    (false, NetworkVariableReadPermission.Everyone, NetworkVariableWritePermission.Server);
```

- Todos los jugadores (clientes) pueden consultar si la palanca está activada o no.
- Sin embargo, solo el servidor tiene permiso para modificar su estado.
- Esto es importante para evitar que un cliente malintencionado cambie el estado de la palanca sin control.
- La activación de la palanca se realiza a través de un ServerRpc, asegurando que solo el servidor pueda validar la acción.

3. RPCs

3.1. ChatUserList:

```
[ServerRpc(RequireOwnership = false)]
1 reference
private void AddConnectedClientServerRpc(ChatUserdata newChatUserData)
{
    chatUsers.Add(newChatUserData);
    UpdatUserConnectedListClientRpc(chatUsers.ToArray());
}

[ClientRpc]
3 references
private void UpdatUserConnectedListClientRpc(ChatUserdata[] userList)
{
    chatUsers = userList.ToList();
}
```

En este caso, al ingresar un jugador, su nombre se añade a la lista:

```
[SerializeField] private List<ChatUserdata> chatUsers = new List<ChatUserdata>();
```

El mensaje es enviado al servidor, el cual después lo reproduce en todos los clientes para que tengas la lista actualizada.

3.2. ChatManager:

```
[ServerRpc(RequireOwnership =false)]
1 reference
private void SendMessageServerRpc(string messageToSend)
{
    SendMessageClientRpc(messageToSend);
}
[ClientRpc]
1 reference
private void SendMessageClientRpc(string messageToSend)
{
    chatLog.text += "\n" + messageToSend;
}
```

Los mensajes son enviados desde los clientes hacia el servidor, el cual después reproduce los mensajes en todos los demás clientes en el ChatLog.text.

3.3. WinGame:

```
[ServerRpc(RequireOwnership = false)]
1 reference
private void PlayerEnteredZoneServerRpc(ulong clientId)
{
    playersInZone.Add(clientId);
    CheckPlayersInZone();
}


[ServerRpc(RequireOwnership = false)]
1 reference
private void PlayerExitedZoneServerRpc(ulong clientId)
{
    playersInZone.Remove(clientId);
}
```

Primeramente, se tiene dos ServerRPC los cuales informan al servidor acerca de los jugadores que entran y salen del objetivo de victoria.

```
[ServerRpc(RequireOwnership = false)]
2 references
private void RequestReturnToLobbyServerRpc(ServerRpcParams rpcParams = default)
{
    Debug.Log($"Jugador {rpcParams.Receive.SenderClientId} solicitó volver al lobby.");
    NetworkManager.Singleton.SceneManager.LoadScene(lobbySceneName, UnityEngine.SceneManagement.LoadSceneMode.Single);
}
```

Después, se tiene el RequestReturnToLobbyRpc, el cual solicita al servidor que regrese a la escena del lobby.

```
[ClientRpc]
1 reference
private void NotifyClientsWinClientRpc()
{
    if (winPanel != null)
    {
        winPanel.SetActive(true); // Activa el panel de victoria en todos los clientes.
    }
}
```

Se les notifica a los clientes del estado de “Victoria”, mostrando el panel que tiene el mensaje y el botón de desconexión.

Finalmente se procede a notificar a todos los clientes sobre el cambio de escena al menú principal.

```
[ClientRpc]
1 reference
private void NotifyClientsToChangeSceneClientRpc(string sceneName)
{
    Debug.Log($"Cambiando a la escena {sceneName} para todos los clientes.");
}
1 reference
private void OnCloseButtonClicked()
{
    if (connectionManager != null)
    {
        connectionManager.out_Disconnect(); // Llamar al método de desconexión
    }
    else
    {
        Debug.LogWarning("ConnectionManager no encontrado.");
    }

    // Cambiar la escena a "Multiplayer"
    ReturnToMultiplayer();
}
```

3.4. Lever:

```
[ServerRpc(RequireOwnership = false)]
1 reference
private void ActivateLeverServerRpc()
{
    isActivated.Value = true; // Sincronizar estado con 'NetworkVariable'
    activatedLeversCount++;

    UpdateLeverStateClientRpc(isActivated.Value); // Actualiza visualmente en todos los jugadores
    UpdateLeverStatusClientRpc(activatedLeversCount, totalLevers); // Actualiza UI en todos los jugadores

    if (activatedLeversCount == totalLevers)
    {
        NotifyAllPlayersToChangeSceneServerRpc();
    }
}
```

Activación de las palancas en el servidor, actualizando su estado y verifica si han sido activadas.

```
[ClientRpc]
1 reference
private void UpdateLeverStateClientRpc(bool state)
{
    isActivated.Value = state;

    if (leverObject != null)
    {
        leverObject.SetActive(!state);
    }
}
```

Sincronización visual del estado de las palancas en todos los clientes.

```
[ClientRpc]
1 reference
private void UpdateLeverStatusClientRpc(int activatedLevers, int totalLevers)
{
    PlayerUIManager.Singleton.UpdateLeverStatus(activatedLevers, totalLevers);
}
```

Actualización de la UI de los jugadores, mostrando cuantas palancas han sido activadas.

3.5. LobbyManager:

```
// ServerRpc para actualizar la lista de jugadores listos
[ServerRpc(RequireOwnership = false)]
1 reference
public void OnPlayerReadyServerRpc(ulong playerId)
{
    if (!playersReady.ContainsKey(playerId))
    {
        playersReady.Add(playerId, false);
        userList.text += playerId + " - Not Ready\n"; // Mostrar ID del jugador con "Not Ready"
        Debug.Log($"Servidor: Jugador {playerId} ha marcado 'Listo'.");
    }

    // Actualiza la lista de jugadores en todos los clientes
    UpdatePlayerListClientRpc(playersReady.Keys.ToArray(), playersReady.Values.ToArray());

    if (playersReady.Count == ConnectionManager.Singleton.connectedClients.Count)
    {
        EnableStartButtonClientRpc(); // Habilitar botón de "Start"
    }
}
```

Notifica al servidor que un jugador marco su estado como “Listo”.

```
[ClientRpc]
2 references
public void EnableStartButtonClientRpc()
{
    startGameButton.SetActive(true);
    Debug.Log("El botón de 'Start' ha sido habilitado para todos los jugadores.");
}
```

Activación de botón “Start”, solo disponible su uso para el Servidor.

```
// Este ClientRpc actualizará la lista de jugadores en todos los clientes
[ClientRpc]
2 references
public void UpdatePlayerListClientRpc(ulong[] playerIds, bool[] readyStatuses)
{
    for (int i = 0; i < playerIds.Length; i++)
    {
        string status = readyStatuses[i] ? "Ready" : "Not Ready";
        userList.text += $"\nJugador {playerIds[i]} ha marcado 'Listo'.";

    }
}
```

Actualización de la lista de jugadores “Listos” en todos los demás clientes.

```
[ClientRpc]
1 reference
public void DisableMenuCanvasClientRpc()
{
    startGameButton.SetActive(false);
    disconnectButton.SetActive(true);
    // Desactiva todos los paneles del menú y el lobby en todos los clientes, excepto el chatPanel
    if (menuCanvas != null)
    {
        foreach (Transform child in menuCanvas.transform)
        {
            if (child.gameObject != chatPanel && child.gameObject != disconnectButton)
            {
                child.gameObject.SetActive(false);
            }
        }
    }
}
```

Desactivación del menú, manteniendo el chat en todos los clientes al iniciar la partida.

4. Motivaciones.

Para el desarrollo del entorno multijugador de "Try to Get Free", se ha optado por Unity Netcode for GameObjects como el framework principal de red. La elección de este framework y su configuración específica en términos de capa de transporte y protocolo han sido clave para lograr una experiencia estable y fluida. A continuación, se detallan las motivaciones y beneficios de esta elección:

4.1. Elección de Unity Netcode for GameObjects

- Integración nativa con Unity: Netcode for GameObjects es una solución oficial de Unity que facilita la implementación del multijugador sin necesidad de herramientas externas.
- Soporte para múltiples arquitecturas: Permite trabajar con modelos cliente-servidor, peer-to-peer y arquitecturas híbridas.
- Compatibilidad con Unity Relay: Al utilizar Unity Relay Server, podemos evitar problemas con NAT traversal y facilitar la conexión entre jugadores sin necesidad de abrir puertos.
- Sincronización optimizada: El uso de NetworkVariable<T>, ServerRpc y ClientRpc permite una sincronización eficiente con baja sobrecarga de red.Escalabilidad: La infraestructura es adaptable tanto para partidas privadas entre amigos como para entornos con múltiples sesiones simultáneas.

4.2. Capa de transporte utilizada

En cuanto a la capa de transporte, Unity Netcode for GameObjects soporta múltiples opciones. Para este proyecto, hemos seleccionado Unity Transport Protocol (UTP) por las siguientes razones:

- **Fiabilidad y baja latencia:** UTP optimiza la transmisión de datos asegurando una comunicación rápida y precisa entre los jugadores.
- **Seguridad con DTLS:** Se ha configurado el uso de DTLS (Datagram Transport Layer Security) para cifrar la comunicación y prevenir ataques de intermediarios.

- **Soporte para Relay:** UTP permite una integración directa con Unity Relay, lo que evita problemas de conectividad en redes con restricciones de NAT.
- **Menor consumo de ancho de banda:** Permite el uso eficiente de la red mediante la compresión de datos y la optimización en el envío de paquetes.

4.3. Protocolo de transporte

El protocolo de transporte es un factor clave en la experiencia multijugador. Para este proyecto, hemos elegido una combinación de UDP y DTLS:

4.3.1. UDP (User Datagram Protocol).

- Protocolo rápido y eficiente para la transmisión de datos en tiempo real.
- Ideal para enviar información sobre posiciones, movimientos y estado de los objetos en la red sin la sobrecarga de confirmaciones de entrega.
- Funciona bien en entornos donde la pérdida ocasional de paquetes no afecta drásticamente la jugabilidad.

4.3.2. DTLS (Datagram Transport Layer Security).

- Agrega una capa de seguridad al protocolo UDP.
- Cifra la comunicación para evitar accesos no autorizados a los datos transmitidos.
- Protege contra ataques como suplantación de identidad y manipulación de paquetes.

5. Interacciones multijugador y online.

El diseño de "Try to Get Free" se basa en una experiencia multijugador cooperativa, donde dos jugadores deben sincronizar sus acciones para completar los niveles. Para ello, se han implementado diversas interacciones multijugador que permiten la comunicación y sincronización entre los jugadores en tiempo real.

5.1. Interacciones en el juego

Las interacciones principales entre los jugadores y el entorno incluyen:

- **Movimientos y posiciones:** Cada jugador puede moverse libremente dentro del nivel y su posición se sincroniza en todos los clientes mediante NetworkTransform.

- **Activación de palancas:** Ambas palancas deben ser activadas por los jugadores para avanzar al siguiente nivel. La activación se gestiona mediante RPCs y variables de red para garantizar su sincronización.
- **Cambio de nivel:** Una vez que ambos jugadores han activado sus respectivas palancas y están en la zona de activación, el servidor inicia el cambio de escena.
- **Chat entre jugadores:** Los jugadores pueden comunicarse mediante un chat de texto sincronizado en todos los clientes mediante ServerRpc y ClientRpc.

5.2. Arquitectura e infraestructura

El juego utiliza una arquitectura cliente-servidor gestionada mediante Unity Relay Server, lo que permite una conexión estable sin necesidad de que los jugadores configuren sus redes manualmente.

5.3. Estructura del servidor y clientes

- **Host (Servidor):** Se encarga de sincronizar el estado del juego, incluyendo posiciones, activaciones de palancas y transiciones de niveles.

Gestiona todas las variables de red (NetworkVariable<T>) y ejecuta los ServerRpc.

- **Clientes:** Reciben los cambios del servidor mediante ClientRpc.

Pueden enviar solicitudes al servidor mediante ServerRpc, por ejemplo, para activar una palanca o enviar un mensaje de chat.

5.4. Uso de Unity Relay Server

- Evita problemas de NAT traversal: Los jugadores pueden conectarse sin necesidad de abrir puertos en su router.
- Optimiza la latencia: El relay elige la mejor ruta de comunicación para minimizar el retraso.
- Facilita la conexión: Permite que los jugadores se unan a partidas sin necesidad de una IP pública.

6. Flujo del juego y gestión de escenas

El flujo de juego en "Try to Get Free" sigue una secuencia clara y bien definida, asegurando que los jugadores experimenten una progresión lógica desde el inicio hasta la finalización de la partida.

6.1. Estructura del flujo de juego

- Pantalla de Inicio: Los jugadores tienen la opción de crear una partida, unirse a una existente o configurar ajustes.
- Lobby: Una vez en el lobby, los jugadores pueden marcar su estado como "Listo". La partida solo inicia cuando ambos jugadores están listos.
- Carga de Nivel: El servidor asigna el nivel inicial y sincroniza la carga de la escena en todos los clientes.
- Juego Activo: Los jugadores pueden moverse e interactuar con el entorno, incluyendo la activación de palancas.
- Cambio de Nivel: Cuando las palancas han sido activadas y ambos jugadores están en la zona de activación, se inicia la transición al siguiente nivel.
- Pantalla de Victoria: Al completar el último nivel, se muestra un mensaje de victoria y se da la opción de regresar al lobby o salir del juego.

6.2. Gestión de escenas en Netcode for GameObjects

La gestión de escenas en Unity Netcode se realiza a través del NetworkSceneManager. El servidor es responsable de cargar las escenas y asegurarse de que todos los clientes cambien a la nueva escena de manera sincronizada.

- **Carga inicial de la escena:** Cuando se inicia una partida, el servidor asigna la primera escena y notifica a los clientes.
- **Cambio de escena tras completar un nivel:** Cuando los jugadores activan las palancas y se encuentran en la zona de activación, el servidor ejecuta el cambio de nivel, además, un ClientRpc puede utilizarse para notificar a los clientes que la escena está cambiando.

6.3. Sincronización de estados al cambiar de nivel.

Para evitar inconsistencias entre los jugadores al cambiar de nivel, se utilizan NetworkVariable<T> para sincronizar estados clave, Al cambiar de nivel, el servidor actualiza la variable y la propaga a todos los clientes.

7. Objetos de red y sus componentes.

En "Try to Get Free", los objetos de red utilizados son fundamentales para sincronizar el estado del juego entre todos los jugadores. A continuación, se describen los principales objetos de red y sus componentes clave:

7.1. Jugador (PlayerController).

Cada jugador en la partida es un objeto de red (NetworkObject) que debe sincronizar su estado con los demás clientes. Los componentes principales incluyen:

- **NetworkObject** → Permite que Unity Netcode gestione la identidad del jugador en la red.
- **NetworkTransform** → Sincroniza la posición y rotación del jugador.
- **PlayerController** → Maneja el movimiento del jugador y las interacciones con el entorno.
- **NetworkVariable<Vector3> playerPosition** → Almacena y actualiza la posición del jugador en la red.

7.2. Palancas (Lever)

Las palancas son elementos clave en el juego, ya que los jugadores deben activarlas para avanzar de nivel. Están sincronizadas en la red y utilizan variables de red y RPCs para su activación.

- **NetworkObject** → Identifica la palanca en la red.
- **NetworkVariable<bool> isActivated** → Sincroniza el estado de activación de la palanca.
- **ServerRpc ActivateLeverServerRpc()** → Se ejecuta en el servidor para actualizar el estado de la palanca.
- **ClientRpc UpdateLeverStateClientRpc()** → Notifica a los clientes sobre la activación de la palanca.

7.3. Zona de activación (MoveSceneManager)

La zona de activación detecta cuando los jugadores han activado las palancas y están listos para avanzar al siguiente nivel.

- **NetworkObject** → Se utiliza para gestionar la interacción en la red.
- **HashSet<ulong> playersInZone** → Lista de jugadores en la zona de activación.
- **ServerRpc PlayerEnteredZoneServerRpc()** → Se ejecuta cuando un jugador entra en la zona.
- **ServerRpc PlayerExitedZoneServerRpc()** → Se ejecuta cuando un jugador sale de la zona.
- **CheckConditions()** → Verifica si los jugadores han activado las palancas y están en la zona para cambiar de nivel.

7.4. Sistema de chat (ChatManager)

El chat permite la comunicación entre los jugadores y utiliza RPCs para enviar y recibir mensajes.

- **ServerRpc SendMessageServerRpc()** → Envía un mensaje al servidor.
- **ClientRpc SendMessageClientRpc()** → Distribuye el mensaje a todos los clientes.

8. Uso de eventos y mensajes en la red

El uso de eventos y mensajes en la red es fundamental para coordinar la interacción entre los jugadores y sincronizar el estado del juego de manera eficiente. En "Try to Get Free", se emplean eventos personalizados y mensajes de red mediante ServerRpc y ClientRpc para garantizar que todos los jugadores estén sincronizados. A continuación, se detallan los eventos más importantes utilizados en el juego y su propósito:

8.1. Evento de activación de palancas

Cuando un jugador activa una palanca, se debe notificar a todos los clientes que el estado ha cambiado. Esto se logra mediante un ServerRpc y un ClientRpc.

- **ActivateLeverServerRpc()** → Se ejecuta en el servidor para cambiar el estado de la palanca.
- **NotifyClientsLeverActivatedClientRpc()** → Envía la actualización a todos los clientes para sincronizar visualmente el estado de la palanca.

8.2. Evento de cambio de escena

Cuando los jugadores han cumplido las condiciones para avanzar al siguiente nivel, el servidor inicia un cambio de escena y notifica a los clientes.

- **ChangeSceneServerRpc()** → Solo el servidor puede iniciar un cambio de escena.
- **NotifyClientsSceneChangingClientRpc()** → Se usa para informar a todos los jugadores que la escena está cambiando, lo que puede usarse para mostrar pantallas de carga o animaciones de transición.

8.3. Evento de mensajería en el chat

El chat permite la comunicación entre los jugadores mediante ServerRpc y ClientRpc.

- **SendMessageServerRpc()** → Envía el mensaje al servidor, quien se encarga de distribuirlo.
- **SendMessageClientRpc()** → Todos los clientes reciben el mensaje y lo muestran en pantalla.

8.4. Evento de victoria.

Cuando los jugadores han activado todas las palancas y están en la zona de activación, el servidor muestra el mensaje de victoria en todos los clientes.

- **CheckWinConditionServerRpc()** → Verifica en el servidor si los jugadores cumplen las condiciones para ganar.
- **NotifyClientsWinClientRpc()** → Activa el panel de victoria en todos los clientes.