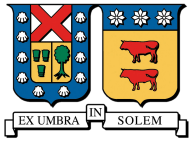

Universidad Técnica Federico Santa María
Sistemas Distribuidos
Primer Semestre 2025

Laboratorio 3: Sistemas Distribuidos

Sistema Distribuido de Emparejamiento Multijugador Avanzado

Profesor: Jorge Díaz
Ayudantes de Lab: Felipe Marchant y Maximiliano Tapia

Junio 2025

1 Objetivos del Laboratorio

- Familiarizarse con los conceptos asociados a la replicación y consistencia de datos en Sistemas Distribuidos.
- Aplicar consistencia eventual en un sistema distribuido.
- Aplicar modelos de consistencia centrados en el cliente.
- Profundizar en el uso de **Golang**, **gRPC** y **Docker**.

2 Introducción

Dentro de los sistemas distribuidos, la replicación pasa a ser una buena alternativa para manejar la tolerancia a fallos, además de beneficiarse en cuanto al rendimiento debido a la división de la carga entre las diferentes réplicas que se manejen del sistema. Sin embargo, la replicación tiene un trade-off y es que introduce problemas de consistencia, ya que cuando se actualiza una réplica, entonces se vuelve diferente de las otras. Para solucionar esto, se han propuesto modelos de consistencia. Para poner esto en práctica, se propone el siguiente problema en donde, haciendo uso de gRPC, Golang, y modelos de consistencia centrados en los datos y el cliente, deberán simular un sistema de emparejamiento (matchmaking). Para este laboratorio, se hará uso del modelo de consistencia eventual junto a Read Your Writes.

3 Tecnologías Utilizadas

- El lenguaje de programación a utilizar es **Go**.
- Para la comunicación se utilizará **gRPC**.
- Para la definición de mensajes síncronos se usará **ProtocolBuffers**
- Para distribuir el programa se utilizará **Docker**

4 Laboratorio

4.1 Contexto General

El sistema a desarrollar simulará una plataforma de emparejamiento (matchmaking) para un juego multijugador en línea. El objetivo principal es asignar jugadores que buscan una partida a servidores de juego disponibles, considerando diferentes modos de juego (ej. 1v1, 2v2, 3v3 - aunque para simplificar nos centraremos en un tipo de partida genérico de 1v1). El sistema debe ser robusto, capaz de manejar fallos en los servidores de juego, y mantener una visión consistente del estado global. Un cliente administrador permitirá la supervisión y gestión del sistema. Los principales desafíos radican en:

- **Coordinación Distribuida:** Asegurar que el Matchmaker pueda tomar decisiones informadas basadas en el estado potencialmente dinámico de los jugadores y servidores.
- **Consistencia de Estado:** Garantizar que el estado del sistema (quién está en cola, qué servidores están libres/ocupados/caídos) sea gestionado con un modelo de consistencia eventual utilizando relojes vectoriales, para mantener la causalidad y permitir la convergencia del estado.
- **Experiencia del Usuario:** Proveer a los jugadores una vista consistente de su propio estado (Read Your Writes).
- **Tolerancia a Fallos:** Permitir que el sistema continúe operando (quizás con capacidad degradada) incluso si algunos Servidores de Partida fallan.
- **Trazabilidad:** Registrar eventos importantes para auditoría o depuración.

4.2 Arquitectura del Sistema

El sistema estará compuesto por las siguientes 7 entidades lógicas, cada una ejecutándose en su propio contenedor Docker:

- **Jugadores (2 instancias):** Aplicaciones cliente que simulan jugadores humanos buscando partidas.
- **Servidores de Partida (3 instancias):** Entidades que simulan la ejecución de las partidas de juego.
- **Matchmaker Central (1 instancia):** El cerebro del sistema, responsable de recibir solicitudes de los jugadores, gestionar el estado de jugadores y servidores, y asignar partidas.
- **Cliente Administrador (1 instancia):** Una aplicación cliente que permite a un administrador supervisar y gestionar el sistema a través del Matchmaker.

Flujo General de Información (Simplificado):

1. Un **Jugador** envía una solicitud para unirse a una partida al **Matchmaker Central**.
2. El **Matchmaker Central** consulta su estado interno (que incluye el estado de los **Servidores de Partida** y otros jugadores en cola).
3. Si hay suficientes jugadores para una partida y un **Servidor de Partida** disponible, el **Matchmaker** asigna la partida.
4. El **Matchmaker** notifica a los **Jugadores** involucrados y al **Servidor de Partida** seleccionado.
5. El **Servidor de Partida** simula la partida. Durante este tiempo, informa al **Matchmaker** que está ocupado.
6. El **Cliente Administrador** puede conectarse en cualquier momento al **Matchmaker** para consultar el estado del sistema o realizar acciones de gestión.

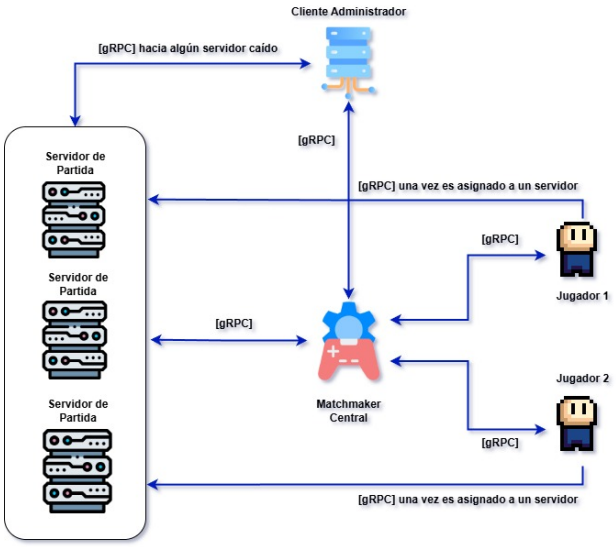

Figure 1: Diagrama conceptual de la arquitectura del sistema distribuido de emparejamiento.

5 Especificación por cada proceso/entidad

A continuación, se detalla el comportamiento esperado, las responsabilidades y las interfaces de cada entidad del sistema.

5.1 Jugadores (Player Clients)

Habrá **dos instancias** de esta entidad, cada una representando a un jugador individual que interactúa con el sistema. Cada instancia será una aplicación de consola, que requiere input.

5.1.1 Responsabilidades Principales

- Permitir al "usuario" (simulado o real a través de la consola) solicitar unirse a una cola de emparejamiento.
- Permitir consultar el estado actual del jugador en el sistema (ej. "Buscando Partida", "En Partida", "Libre"). Es decir, una vez que se une a la cola puede consultar su propio estado, o bien, cancelar su emparejamiento.
- Recibir mensajes del Matchmaker cuando una partida ha sido asignada.
- Implementar la consistencia **Read Your Writes**: si un jugador se une a la cola y luego consulta su estado, debe ver que está en cola.

5.1.2 Interfaz de Comunicación (gRPC Cliente)

El Jugador actúa como cliente gRPC hacia el Matchmaker Central. Deberá implementar llamadas a los siguientes métodos (ejemplos):

- `QueuePlayer(PlayerInfoRequest) returns (QueuePlayerResponse)`: Para solicitar unirse a la cola de emparejamiento.
  - `PlayerInfoRequest`: Debe contener `player_id`, `game_mode_preference`.
  - `QueuePlayerResponse`: Debe contener un `status_code` (éxito/fallo) y un `message`.

- `GetPlayerStatus(PlayerStatusRequest) returns (PlayerStatusResponse)`: Para consultar su estado actual.
  - `PlayerStatusRequest`: Debe contener `player_id`.
  - `PlayerStatusResponse`: Debe contener el estado actual del jugador (ej. una cadena: "IDLE", "IN_QUEUE", "IN_MATCH") y posiblemente el `match_id` y la dirección del servidor de partida si está en una.

5.1.3 Lógica Interna y Comportamiento

- Cada instancia de Jugador debe tener un ID único (ej. "Player1", "Player2"). Este ID se enviará en las solicitudes gRPC.
- Al iniciar, el Jugador podría estar en estado "Libre".
- Debe ofrecer un menú de consola simple para que el usuario elija acciones (ej. 1: Unirse a cola, 2: Consultar estado, 3: Salir).
- Para **Read Your Writes**: Después de una llamada exitosa a `QueuePlayer`, una llamada inmediata a `GetPlayerStatus` debe reflejar que el jugador está en cola. El Matchmaker debe asegurar esto.
- Cuando se le asigna una partida y un servidor, el jugador debe comunicarse directamente con el Servidor de Partida después de la asignación por el Matchmaker, para luego simular la partida (puede ser mediante un log solamente, no es necesario simular una partida real).
- Debe manejar errores de comunicación gRPC (ej. Matchmaker no disponible) de forma elegante, mostrando un mensaje al usuario.

NOTA: Considere que usted puede considerar agregar mensajes adicionales a cada servicio según estime conveniente, pero **debe cumplir con el mínimo exigido** que se indica en este enunciado.

5.2 Servidores de Partida (Game Servers)

Habrá **tres instancias** de esta entidad. Cada una simula un servidor capaz de hospedar una partida.

5.2.1 Responsabilidades Principales

- Registrarse con el Matchmaker Central al iniciarse, indicando su disponibilidad y dirección.
- Informar al Matchmaker sobre su estado actual (Ej: "DISPONIBLE", "OCUPADO", "CAIDO").
- Recibir una asignación de partida del Matchmaker.
- Simular la ejecución de una partida (ej. esperar un tiempo aleatorio entre 10 y 20 segundos).
- Informar al Matchmaker que ha vuelto al estado "DISPONIBLE" tras finalizar una partida.
- Simular caídas aleatorias (para probar la tolerancia a fallos del Matchmaker).

5.2.2 Interfaz de Comunicación (gRPC Servidor y Cliente)

- **Como Servidor gRPC (expone al Matchmaker):**
  - `AssignMatch(AssignMatchRequest) returns (AssignMatchResponse)`: Invocado por el Matchmaker para iniciar una partida.
    - `AssignMatchRequest`: Contendría `match_id`, lista de `player_ids`.
    - `AssignMatchResponse`: Podría contener un `status_code` (éxito/fallo).
- **Como Cliente gRPC:**

- Hacia el **Matchmaker Central**:
  - `UpdateServerStatus(ServerStatusUpdateRequest) returns (ServerStatusUpdateResponse)`: Para informar cambios de estado (ej. DISPONIBLE, OCUPADO, CAIDO). Al inicio, se usa para registrarse.
    - `ServerStatusUpdateRequest`: `server_id`, `new_status`, `address` (IP/puerto del servicio gRPC del Servidor de Partida).
    - `ServerStatusUpdateResponse`: `status_code`.

5.2.3 Lógica Interna y Comportamiento

- Cada instancia debe tener un ID único (ej. "GameServer1", "GameServer2", "GameServer3") y una dirección (host:puerto) donde su servicio gRPC escucha.
- **Registro:** Al arrancar, contacta al Matchmaker (usando `UpdateServerStatus`) para registrarse e indicar que está "DISPONIBLE".
- **Simulación de Partida:** Cuando el Matchmaker llama a `AssignMatch`:
  1. Cambia su estado interno a "OCUPADO".
  2. Informa al Matchmaker de este cambio (`UpdateServerStatus`).
  3. Simula la duración de la partida (ej. `time.Sleep` por un tiempo aleatorio entre X e Y segundos).
  4. Al finalizar la simulación:
     (a) Cambia su estado interno a "DISPONIBLE".
     (b) Informa al Matchmaker de este cambio (`UpdateServerStatus`).
- **Simulación de Fallos:** De forma aleatoria (ej. con una probabilidad baja después de cada partida o cada cierto tiempo), un Servidor de Partida puede simular una "caída". En este caso:
  - Deja de responder a las llamadas gRPC del Marchmaker Central o cualquier otra entidad que no sea el Cliente Administrador.
  - El Matchmaker deberá detectar esto (ej. por timeouts en las llamadas o porque el servidor no actualiza su estado).
- Debe manejar errores de comunicación gRPC al contactar al Matchmaker.

NOTA: Considere que usted puede considerar agregar mensajes adicionales a cada servicio según estime conveniente, pero **debe cumplir con el mínimo exigido** que se indica en este enunciado.

5.3 Matchmaker Central

Es una **única instancia** que actúa como el coordinador principal del sistema. Es el componente más crítico en términos de lógica y consistencia.

5.3.1 Responsabilidades Principales

- Recibir y gestionar solicitudes de emparejamiento de los Jugadores.
- Mantener el estado actualizado de todos los Servidores de Partida registrados (ID, dirección, estado: DISPONIBLE, OCUPADO, CAIDO).
- Mantener el estado de los jugadores que están buscando partida (en cola) o que están en una partida.
- Implementar la lógica de emparejamiento: seleccionar jugadores de la cola y asignarles un Servidor de Partida disponible.

- Propagar y reconciliar el estado del sistema mediante consistencia eventual usando **relojes vectoriales**. Cada modificación al estado se marca con un reloj vectorial para mantener la causalidad entre eventos y facilitar la convergencia del estado en el sistema distribuido.
- Detectar (implícita o explícitamente) fallos en los Servidores de Partida y actualizar su estado.
- Proveer una interfaz para que el Cliente Administrador pueda consultar el estado del sistema y realizar acciones de gestión.

5.3.2 Interfaz de Comunicación (gRPC Servidor y Cliente)

- **Como Servidor gRPC (expone a Jugadores y Cliente Administrador):**
  - `QueuePlayer(PlayerInfoRequest) returns (QueuePlayerResponse)`: (Definido en Jugador) Añade un jugador a la cola de espera.
  - `GetPlayerStatus(PlayerStatusRequest) returns (PlayerStatusResponse)`: (Definido en Jugador) Devuelve el estado actual de un jugador.
  - `AdminGetSystemStatus(AdminRequest) returns (SystemStatusResponse)`: Para el Cliente Administrador. Devuelve el estado de todos los servidores y las colas de jugadores.
    - `SystemStatusResponse`: Debe contener listas de `ServerState` (id, status, address, current_match_id) y `PlayerQueueEntry` (player_id, time_in_queue).
  - `AdminUpdateServerState(AdminServerUpdateRequest) returns (AdminUpdateResponse)`: (Opcional) Para el Cliente Administrador, para forzar el estado de un servidor (ej. marcar como DISPONIBLE o CAIDO).
    - `AdminServerUpdateRequest`: `server_id`, `new_forced_status`.
- **Como Cliente gRPC (hacia Servidores de Partida):**
  - `AssignMatch(AssignMatchRequest) returns (AssignMatchResponse)`: (Definido en Servidor de Partida) Para ordenar a un servidor que inicie una partida.
  - `PingServer(ServerId) returns (PingResponse)` para verificar la salud de un servidor si no hay actualizaciones de estado recientes.

5.3.3 Lógica Interna y Comportamiento

- **Gestión de Estado Interno:**
  - **Servidores de Partida:** Mantener una estructura de datos (ej. un mapa) con la información de cada servidor: `ServerID -> {Address, Status, LastHeartbeat, CurrentMatchID}`. El estado puede ser `DISPONIBLE`, `OCUPADO`, `CAIDO`, `DESCONOCIDO`.
  - **Cola de Jugadores:** Mantener una o más colas (ej. FIFO) para los jugadores esperando partida. Cada entrada podría ser {`PlayerID, TimestampEntradaCola, Preferencias`}.
  - **Partidas Activas:** Un registro de las partidas en curso: `MatchID -> {PlayerIDs, ServerID, StartTime}`.
- **Consistencia Eventual:**
  - Todas las operaciones que modifican las estructuras de estado deben estar marcadas con un reloj vectorial.
  - Las operaciones entrantes se fusionan mediante el algoritmo de comparación de relojes vectoriales para mantener la causalidad.
  - Las lecturas reflejan el estado eventualmente consistente según la propagación de los relojes vectoriales. **Se recomienda que cada respuesta de estado incluya el vector causal más reciente conocido por el Matchmaker**.

- **Registro y Actualización de Servidores de Partida:**
  - Cuando un Servidor de Partida llama a `UpdateServerStatus` (que el Matchmaker expone a través de un servicio interno o recibe como cliente de un servicio de status del GameServer):
    * Si es un nuevo ID, lo registra.
    * Actualiza su estado y `LastHeartbeat`.
- **Lógica de Emparejamiento:**
  1. Periódicamente (o cuando un nuevo jugador entra en cola / un servidor queda disponible), intenta formar una partida.
  2. Verifica si hay suficientes jugadores en cola (ej. 2 para una partida 1v1).
  3. Busca un Servidor de Partida en estado `DISPONIBLE`.
  4. Si ambos se cumplen:
     (a) Remueve los jugadores de la cola.
     (b) Genera un `MatchID` único.
     (c) Marca el Servidor de Partida como `OCUPADO` (internamente primero).
     (d) Llama a `AssignMatch` en el Servidor de Partida seleccionado.
     (e) Si `AssignMatch` tiene éxito:
       – Actualiza el estado de los jugadores a `IN_MATCH`.
       – Registra la partida activa.
     (f) Si `AssignMatch` falla (ej. el servidor no responde):
       – Marca el Servidor de Partida como `CAIDO`.
       – Vuelve a poner a los jugadores en la cola (o al inicio de la misma).
       – Intenta encontrar otro servidor.
- **Detección de Fallos de Servidores de Partida:**
  - Si una llamada a `AssignMatch` falla.
  - Si un Servidor de Partida en estado `OCUPADO` no actualiza su estado a `DISPONIBLE` después de un tiempo máximo esperado para una partida (timeout).
  - Al detectar un fallo, el estado del servidor se actualiza a `CAIDO`. Si tenía una partida asignada que no pudo comenzar o que se interrumpió, los jugadores afectados deberían ser notificados o devueltos a la cola.
- **Manejo de Solicitudes del Cliente Administrador:** Provee los datos solicitados o ejecuta las acciones de gestión, siempre respetando la consistencia secuencial.

NOTA: Considere que usted puede considerar agregar mensajes adicionales a cada servicio según estime conveniente, pero **debe cumplir con el mínimo exigido** que se indica en este enunciado.

5.4 Cliente Administrador

Es una **única instancia** de una aplicación de consola que permite a un administrador humano interactuar con el sistema.

5.4.1 Responsabilidades Principales

- Conectarse al Matchmaker Central para obtener información del sistema.
- Mostrar de forma clara el estado de los Servidores de Partida (IDs, direcciones, estados, partidas actuales).
- Mostrar el estado de las colas de jugadores (quién está esperando, cuánto tiempo).
- Permitir al administrador realizar acciones como forzar el estado de un Servidor de Partida.

5.4.2 Interfaz de Comunicación (gRPC Cliente)

Actúa como cliente gRPC hacia el Matchmaker Central.

- `AdminGetSystemStatus(AdminRequest) returns (SystemStatusResponse)`: Para obtener una vista completa del estado del sistema.

Actúa como cliente gRPC hacia algún servidor caído.

- `AdminUpdateServerState(AdminServerUpdateRequest) returns (AdminUpdateResponse)`: Para modificar el estado de un servidor.

5.4.3 Lógica Interna y Comportamiento

- Debe ofrecer un menú de consola para las acciones disponibles (ej. 1: Ver Estado de Servidores, 2: Ver Colas de Jugadores, 3: Cambiar Estado de Servidor, 4: Salir).
- Al solicitar el estado del sistema, debe formatear y presentar la información recibida del Matchmaker de manera legible.
- Debe manejar errores de comunicación gRPC con el Matchmaker.

6 Modelos de Consistencia a Implementar

La correcta implementación de los modelos de consistencia es crucial para el funcionamiento predecible y correcto del sistema.

6.1 Consistencia Eventual con Relojes Vectoriales

- **Definición Aplicada:** Las operaciones que modifican el estado del sistema distribuido (estado de los Servidores de Partida, colas de jugadores, partidas activas) no se aplican en un orden total global, sino que cada nodo (Matchmaker, Cliente Administrador, Jugadores) mantiene su propia vista local del estado. Para garantizar la convergencia del estado a lo largo del tiempo, se utilizan relojes vectoriales para rastrear la causalidad entre eventos distribuidos.
- **Relojes Vectoriales:**
  - Cada entidad del sistema (Jugadores, Servidores, Matchmaker) mantiene un vector de enteros de tamaño N (donde N es el número total de entidades relevantes).
  - Cada vez que una entidad realiza una operación que modifica el estado, incrementa su propia posición en el vector y adjunta el vector actualizado en los mensajes gRPC.
  - Cuando una entidad recibe un mensaje con un vector, actualiza el suyo **tomando el máximo componente por posición** (merge), y en caso de conflicto concurrente, aplica una **política de resolución** (por ejemplo, *last-writer-wins* con *tie-break* por ID). Se sugiere utilizar su creatividad para implementar una correcta política de resolución. Debe indicarlo claramente en su archivo README.
- **Escenario Clave:**
  1. El Servidor de Partida **S1** actualiza su estado a "DISPONIBLE" e incrementa su vector local: $VC_{S1} = [0, 1, 0, ...]$.
  2. El Jugador **P1** se une a la cola de emparejamiento e incrementa su vector local: $VC_{P1} = [1, 0, 0, ...]$
  3. El Matchmaker recibe ambos mensajes. Compara los relojes vectoriales y puede determinar si los eventos son concurrentes o causales.
  4. En caso de concurrencia (no hay orden total definido), el Matchmaker puede postergar la decisión de emparejamiento o aplicar una política de resolución (por ejemplo, priorizar el jugador con menor ID).

5. La estructura interna del Matchmaker, incluyendo colas y estados de servidores, se actualiza siguiendo la causalidad observada en los relojes vectoriales.
6. El Cliente Administrador que consulta el estado ve una instantánea coherente eventual, que refleja la convergencia del sistema a medida que los relojes vectoriales se propagan.

Por considerar:

- Puede existir un retardo antes de que todas las entidades converjan al mismo estado, pero la divergencia será eventualmente resuelta mediante la fusión de relojes vectoriales.
- Las lecturas (como **GetPlayerStatus** o **AdminGetSystemStatus**) pueden incluir el último vector causal conocido para informar al cliente sobre la "frescura" del estado (con "frescura" nos referimos a qué tan reciente hubo un cambio en el estado).

6.2 Read Your Writes (Jugadores)

- **Definición Aplicada:** Después de que un Jugador realiza una operación que modifica su propio estado a través del Matchmaker (ej. unirse a la cola de emparejamiento), cualquier consulta subsecuente de su estado realizada por ese mismo Jugador al Matchmaker debe devolver un estado que refleje esa modificación (o una posterior).
- **Escenario Clave:**
  1. Jugador P1 envía una solicitud `QueuePlayer` al Matchmaker.
  2. El Matchmaker procesa la solicitud, añade P1 a su cola interna y confirma la operación a P1 (ej. devuelve una respuesta exitosa).
  3. Inmediatamente después, P1 envía una solicitud `GetPlayerStatus` al Matchmaker.
  4. La respuesta a `GetPlayerStatus` **debe** indicar que P1 está "EN_COLA" (o un estado posterior si, por una increíble rapidez, ya fue asignado a una partida). No debe mostrar el estado anterior (ej. "LIBRE").

Nota: Este modelo puede convivir con consistencia eventual del sistema si cada entidad mantiene una copia local de su estado y el Matchmaker responde con el vector causal que garantiza que la operación de escritura del jugador fue procesada.

6.3 Ejemplo: "Camino Feliz" para sistema con 3 entidades

Supongamos que sólo contamos con 3 entidades para el sistema completo:

- Jugador 1 (o Player1)
- Matchmaker
- Servidor de Partida 1 (o GameServer1)

Cada entidad tiene un reloj vectorial de tamaño 3 (considerando al jugador, matchmaker y el servidor). Los relojes se inicializan en `[0, 0, 0]`, en el orden `[Jugador1, Matchmaker, GameServer1]`. El flujo sigue tal que:

1. Player1 quiere unirse a la cola
   (a) Incrementa su propio reloj: `[1, 0, 0]`
   (b) Envía `QueuePlayerRequest` al Matchmaker con este vector.
2. Matchmaker recibe el mensaje
   (a) El Matchmaker tiene `[0, 0, 0]`, recibe `[1, 0, 0]`

(b) Hace merge: `merge([1, 0, 0], [0, 0, 0]) = [1, 0, 0]`.

(c) Incrementa su propio reloj: `[1, 1, 0]`

(d) Añade a Player1 a la cola

(e) Envía `QueuePlayerResponse` con `[1, 1, 0]`

3. (Opcional) Player1 consulta su estado

   (a) Player1 tiene `[1, 0, 0]`, recibe `[1, 1, 0]`

   (b) Hace merge: `merge([1, 0, 0], [1, 1, 0]) = [1, 1, 0]`

   Ahora **Player1** ve su propio estado reflejado ("**EN COLA**"), cumpliendo **Read Your Writes**, incluso con consistencia eventual!.

4. Matchmaker inicia emparejamiento

   (a) Suponiendo que GameServer1 está disponible, Matchmaker llama a `AssignMatch` en GameServer1

   (b) Antes de enviar, incrementa su reloj: `[1, 2, 0]`. Pues ahora Matchmaker generó un cambio en el sistema

   (c) Envía este vector con el match

5. GameServer1 recibe asignación

   (a) Tiene `[0, 0, 0]`, recibe `[1, 2, 0]`

   (b) Hace merge: `merge([0,0,0], [1,2,0]) = [1,2,0]`

   (c) Incrementa su posición: `[1, 2, 1]`

   Observación: puede notar que ahora GameServer1 es capaz de saber que:

   - Player1 hizo una solicitud
   - Matchmaker la procesó
   - GameServer1 (asímismo) está ejecutando una acción causalmente posterior

6. GameServer1 termina la partida

   (a) Incrementa: `[1, 2, 2]`

   (b) Informa al Matchmaker que está disponible nuevamente

7. Matchmaker actualiza estado

   (a) Tiene `[1, 2, 0]`, recibe `[1, 2, 2]`

   (b) Merge: `[1, 2, 2]`

   (c) Incrementa: `[1, 3, 2]`

   Observación: Ahora el estado interno del Matchmaker **refleja todos los eventos previos**, y puede responder a nuevos jugadores con una vista actualizada sin necesidad de orden total global!.

7 Aspecto (no menos) Importante

7.1 Tolerancia a Fallos

El sistema debe ser capaz de manejar la caída de los Servidores de Partida.

- **Detección de Fallos por el Matchmaker:**

- **Fallo en Asignación:** Si el Matchmaker intenta asignar una partida a un Servidor de Partida (`AssignMatch`) y la llamada gRPC falla (timeout, servidor no alcanzable), el Matchmaker debe considerar ese servidor como potencialmente caído.
- **Timeout de Partida:** Si un Servidor de Partida está `OCUPADO` con una partida, pero no informa de su finalización (cambiando su estado a `DISPONIBLE`) dentro de un período de tiempo razonable (ej. `MAX_MATCH_DURATION`), el Matchmaker puede considerarlo caído.

- **Acciones del Matchmaker ante un Fallo Detectado:**
  - Marcar el Servidor de Partida afectado como `CAIDO` (o `UNAVAILABLE`) en su estado interno. Este servidor no será considerado para futuras asignaciones hasta que se recupere.
  - Si el fallo ocurrió durante la asignación de una nueva partida, los jugadores que iban a ser asignados a ese servidor deben ser devueltos a la cola de emparejamiento (preferiblemente al inicio de la cola o con su prioridad original). El Matchmaker debería intentar entonces encontrar otro servidor para ellos.
  - Si el fallo ocurrió mientras una partida estaba en curso (más difícil de manejar sin que los jugadores informen), para este laboratorio, se asume que la partida se pierde. El foco está en que el Matchmaker no siga intentando usar el servidor caído.

8 Uso de Docker

Todas las entidades del laboratorio deben ejecutarse en contenedores Docker dentro de las máquinas virtuales y deben estar aisladas en contenedores separados. Además, se pone **COMO EJEMPLO (no obligatorio)** seguir la siguiente distribución para las entidades en las máquinas virtuales:

- MV1: Jugador 1 / Servidor 1
- MV2: Jugador 2 / Servidor 2
- MV3: Servidor 3
- MV4: Cliente Administrador / Matchmaker Central

Recalcamos que, si bien esto es solo un ejemplo, es **mandatorio** que tanto Jugador 1 y 2, junto con los 3 servidores a implementar, queden en máquinas diferentes.

9 Restricciones

Para asegurar un enfoque en los conceptos fundamentales del curso y facilitar la evaluación, se establecen las siguientes restricciones en cuanto a las librerías de Go permitidas:

`bufio`, `context`, `fmt`, `log`, `math`, `math/rand`, `net`, `os`, `os/signal`, `strconv`, `strings`, `sync`, `time`.

Importante: Todo uso de librerías externas que no se han mencionado en el enunciado debe ser consultado con los ayudantes.

10 Consideraciones Adicionales

- **Impresiones por Pantalla (Logging Detallado):**
  - Es **fundamental** que cada entidad imprima mensajes detallados en su consola (salida estándar/error estándar) sobre su estado, acciones realizadas, mensajes gRPC enviados y recibidos, errores encontrados, etc.

  - Utilizar prefijos distintivos para cada entidad en sus logs (ej. `[Matchmaker] INFO: ...`, `[Player1] DEBUG: ...`).
  - Este logging es crucial para la depuración durante el desarrollo y para la evaluación del funcionamiento correcto del sistema. Se espera ver claramente el flujo de operaciones y la aplicación de los modelos de consistencia.
  - Se recomienda que cada mensaje gRPC de escritura o actualización incluya el reloj vectorial actual, y que este se loguee junto con el evento para facilitar trazabilidad y depuración causal.

- **Documentación (`README.md`):**
  - Se debe entregar un archivo `README.md` exhaustivo en la raíz del repositorio del proyecto.
  - Este archivo debe incluir: **Nombres y Roles de los Integrantes del Grupo**, **Instrucciones Detalladas de Compilación y Ejecución**, **Consideraciones adicionales**.
  - El `README.md` debe ser claro y suficientemente detallado para que una persona externa pueda entender, compilar, ejecutar y evaluar el proyecto sin dificultades.

- **Idempotencia:**
  - Para el Matchmaker: si un jugador envía `QueuePlayer` múltiples veces, ¿qué sucede? Idealmente, si ya está en cola, las solicitudes subsecuentes no deberían alterar su posición o crear entradas duplicadas, sino quizás devolver un error o un estado de "ya en cola".

- **Archivos de Configuración vs. Variables de Entorno:**
  - Se recomienda utilizar variables de entorno para configurar los contenedores Docker (direcciones de otros servicios, puertos, IDs, etc.), ya que es una práctica estándar y se integra bien con Docker Compose. Evitar archivos de configuración hardcodeados dentro del código o que requieran montaje de volúmenes complejos si no es estrictamente necesario.

11 Reglas de Entrega

- La tarea se entrega en grupos de 2 personas previamente asignados en aula. Con la excepción de algunos casos puntuales.
- La fecha de entrega es el día **viernes 27 de junio 23:59 hrs. sin posibilidad de extensión de plazo**.
- La tarea se revisará en las máquinas virtuales, por lo que los archivos necesarios para la correcta ejecución de esta deben estar en ellas. Recuerde que el código debe estar identado, comentado, sin warnings y sin errores. **Si hay alguna entidad que no esté subida a las VMs, se calificará con nota 0 y SIN POSIBILIDAD DE RECORRECCIÓN.**
- Se aplicará un descuento de 5 puntos al total de la nota por cada Warning, Error o Problema de Ejecución.
- Además de los códigos en las máquinas virtuales y github, deberán subir un archivo comprimido que contenga todos los códigos desarrollados en carpetas separadas por entidad en formato ***.zip*** con el nombre **GrupoXX-LabY.zip**. Donde XX es el número de su grupo e Y es el número del laboratorio. Ejemplo: *Grupo08-Lab3.zip*. El no cumplimiento (no tener los archivos en el repositorio) implica un descuento del **50% de la nota final**.
- **Debe** dejar un `MAKEFILE` o similar en cada máquina virtual asignada a su grupo para la ejecución de cada entidad. Este debe manejarse de la siguiente forma **(EJEMPLO)**:
  - `make docker-jugador1-servidor1`: Iniciará el código hecho en Docker para el jugador 1 y el servidor 1.

  - `make docker-jugador2-servidor2`: Iniciará el código hecho en Docker para el jugador 2 y el servidor 2.
  - `make docker-servidor3`: Iniciará el código hecho en Docker para el servidor 3.
  - `make docker-admin-matchmaker`: Iniciará el código hecho en Docker para el cliente administrador y el matchmaker central.

- Debe dejar un **README** en la entrega asignada a su grupo con nombre y rol de cada integrante, además de la información necesaria para ejecutar los archivos. La no entrega de este archivo, o un archivo incompleto (sin nombres, roles, instrucciones de ejecución, etc.) conlleva un descuento de 20 puntos en la nota final.

- No se aceptan entregas que no puedan ser ejecutadas desde una consola de comandos. Incumplimiento de esta regla significa nota 0.

- Cada hora o fracción de atraso se penalizará con un descuento de 10 puntos.

- El uso de **Docker** es **obligatorio**. Teniendo nota 0 aquellas implementaciones que no hagan uso de Docker.

- Copias serán evaluadas con nota 0. Serán notificadas al profesor y las autoridades pertinentes.

- **Simplificación de Ejecución en VMs:** A la hora de levantar los contenedores en las máquinas virtuales para la revisión, el proceso debe ser lo más simple posible. Idealmente, un único comando `make` (ej. `make deploy-vmX`) por cada VM debería ser suficiente para iniciar las entidades correspondientes en esa VM. No debería ser necesario ejecutar múltiples comandos manuales (como `make clean`, creación de redes Docker manualmente, etc.) para una ejecución estándar. Los cambios de directorio (`cd`) dentro de un script `make` son aceptables. El incumplimiento de esta simplicidad (requerir múltiples comandos manuales por VM) conllevará un descuento de **5 puntos por comando extra** y podría llevar a que la entrega sea considerada para recorrección con penalización.