Informe Técnico: Implementación del Juego UNO Distribuido

Materia: Programación Orientada a Objetos / Sistemas Distribuidos
Tecnologías: Java, RMI, MVC, JavaFX

1. Idea Inicial y Arquitectura

El objetivo del proyecto fue desarrollar una versión multijugador del juego de cartas **UNO** capaz de operar en una red distribuida.

Para lograr esto, se implementó una arquitectura basada en el patrón **MVC (Modelo-Vista-Controlador)** adaptada para sistemas distribuidos mediante **Java RMI (Remote Method Invocation)**.

- **Modelo (Servidor):** Centraliza la lógica del juego (Partida, Mazo, Jugador). Es el "único punto de verdad". Reside en el servidor y extiende ObservableRemoto para notificar cambios.
- **Vista (Cliente):** La interfaz de usuario (JavaFX). Es totalmente pasiva; solo muestra lo que el controlador le indica y captura eventos del usuario.
- **Controlador (Cliente):** Actúa como intermediario. Recibe eventos locales de la Vista y los envía al Servidor RMI. A su vez, escucha eventos del Servidor (Observer) y actualiza la Vista.

Esta separación permite que múltiples clientes (Vistas) se conecten a un único juego (Modelo) y se mantengan sincronizados en tiempo real.

---

2. Desafíos Encontrados y Soluciones

Durante el desarrollo surgieron varios desafíos técnicos y lógicos que fueron resueltos iterativamente:

A. Sincronización y Concurrencia (El problema de las "13 cartas")

- **Problema:** Al inicio, si dos jugadores presionaban "Iniciar Partida" simultáneamente, el servidor recibía dos peticiones y repartía cartas duplicadas, rompiendo el estado del juego.
- **Solución:** Se blindaron los métodos críticos del Modelo (iniciarPartida, jugarCarta, robar) utilizando la palabra clave synchronized y validaciones de estado (if (partidaEnCurso) return;). Esto garantiza que las operaciones sean atómicas y secuenciales.

B. Lógica de Turnos y "Deadlocks" (Caso 2 Jugadores)

- **Problema:** En partidas de 2 jugadores, la carta "Saltarse" o "Cambio de Sentido" debe permitir al mismo jugador jugar de nuevo. El sistema original intentaba validar si el jugador "había robado" antes de pasar turno, generando un bloqueo lógico donde el sistema impedía el avance automático del turno.
- **Solución:** Se separó la lógica de avance de turno en dos métodos:
    1. pasarTurno() (Público): Incluye validaciones estrictas para el usuario (anti-trampa).
    2. avanzarTurnoInterno() (Privado): Mueve el puntero del turno sin restricciones, utilizado exclusivamente por la lógica de cartas especiales (+2, Salto, +4).

C. Seguridad de Identidad (El problema de la Consola)

- **Problema:** Un cliente (especialmente la vista de consola) podía enviar comandos como ROBAR cuando no era su turno, afectando al jugador activo.
- **Solución:** Se implementó una validación de identidad en el ControladorUNO. El controlador verifica localmente (esMiTurno()) si el nombre del cliente coincide con el jugador activo antes de enviar cualquier petición al servidor (RMI).

D. Actualización de Interfaz (JavaFX Threads)

- **Problema:** Las notificaciones RMI llegan en hilos de ejecución distintos al hilo principal de la interfaz gráfica, causando excepciones al intentar actualizar componentes visuales.
- **Solución:** Se encapsularon todas las actualizaciones de la UI dentro de Platform.runLater(() -> { ... }), asegurando la estabilidad de la aplicación gráfica.

---

3. Consideraciones de Implementación

Flexibilidad de Vistas

El sistema soporta múltiples tipos de vistas simultáneas gracias a la interfaz VistaObserver. Se implementaron:

- **Vista Gráfica (GUI):** Renderizado de cartas mediante JavaFX Shapes (sin dependencia de imágenes externas).
- **Vista Terminal (Simulada):** Una interfaz estilo "retro" que emula una línea de comandos pero corre sobre JavaFX para evitar bloqueos del hilo principal (System.in).

Gestión de Estado Visual

Para mejorar la experiencia de usuario (UX), la vista detecta no solo cambios de turno, sino cambios en la mesa.

- Se implementaron "banderas" locales (yaRobe, ultimaCartaRegistrada) para habilitar o deshabilitar botones (Pasar, Robar) dinámicamente, guiando al usuario sobre las acciones permitidas en cada momento.

Ciclo de Vida Completo

Se consideró el flujo completo de la aplicación:

1. **Login:** Registro de nombre.
2. **Lobby:** Sala de espera reactiva.
3. **Juego:** Lógica completa de UNO.
4. **Re-play:** Capacidad de reiniciar la partida manteniendo los jugadores conectados o gestionar desconexiones limpias.

---

4. Conclusión

El proyecto resultante es un sistema robusto, modular y thread-safe. La utilización estricta del patrón MVC junto con la librería RMI permitió desacoplar la lógica de negocio de la presentación, facilitando la implementación de reglas complejas (como las cartas especiales) sin afectar la capa visual.