Implementación y Análisis de un Sistema de Subastas Distribuido con Java RMI y gRPC

Miembros del Equipo:
Camila García Álvarez
Celeste Isabel Alonso García
Jesús Álvarez Sombrerero

Departamento de Computación, Electrónica y Mecatrónica
Universidad de las Américas Puebla (UDLAP)

26 de octubre de 2025

Materia: LIS-4052 Sistemas Distribuidos
Profesor: José Luis Zechinelli Martini

Resumen

Resumen: En este reporte, detallamos nuestro proceso de diseño, migración e implementación de una aplicación de subastas, transformándola de una arquitectura centralizada (MVC) a un sistema distribuido. Para cumplir con los requisitos de la Actividad 4 y obtener las décimas extras, implementamos dos soluciones paralelas utilizando Java Remote Method Invocation (RMI) y gRPC. Resolvimos exitosamente el desafío principal de la práctica, la sincronización de estado en tiempo real, mediante la implementación del patrón observador. Logramos esto en RMI a través de callbacks remotos y en gRPC mediante server-side streaming. Este documento presenta nuestro diseño, un análisis de los datos obtenidos durante nuestra experimentación (basado en la rúbrica SO6) y una comparación técnica de las ventajas y desventajas de RPC, RMI y gRPC para este caso de uso.

Índice

1. Objetivo de la Práctica

Nuestro objetivo en esta práctica fue doble. Primero, aplicar los procedimientos para construir una aplicación distribuida partiendo de una versión centralizada. Para obtener las décimas extras y realizar una comparación más profunda, decidimos implementar la solución completa con dos tecnologías de middleware clave: **Java RMI** y **gRPC**.

Segundo, y como reto central de la actividad, debíamos proponer e implementar una estrategia de software robusta para resolver el problema de la falta de sincronización de datos entre múltiples clientes. Específicamente, teníamos que asegurar que cuando un usuario propone un nuevo precio, las vistas de todos los demás clientes se actualizaran automáticamente.

Como entregable final, preparamos este reporte donde demostramos, acorde a la rúbrica SO6, nuestra capacidad para:

- Diseñar y llevar a cabo una experimentación adecuada.
- Analizar e interpretar los datos que obtuvimos (logs, comportamiento de la GUI).
- Utilizar el juicio de ingeniería para sacar conclusiones válidas.

2. Diseño de la Solución

La base de nuestra práctica fue la aplicación centralizada provista, que sigue el patrón Modelo-Vista-Controlador (MVC). En esta, `SubastaVista` maneja la GUI, `SubastaModelo` contiene la lógica de negocio y el estado, y `SubastaControlador` actúa como puente.

2.1. Arquitectura Distribuida

Nuestra estrategia de migración consistió en identificar el `SubastaModelo` como el componente a ser centralizado y servido.

- **Servidor:** Convertimos la lógica de `SubastaModelo` en el servicio remoto. Este se convirtió en el único componente que mantiene el estado verdadero del sistema.
- **Cliente:** Mantuvimos las clases `SubastaVista` y `SubastaControlador` en la máquina cliente. Modificamos el `SubastaControlador` para que, en lugar de invocar métodos locales del modelo, realizara llamadas remotas (RMI o gRPC) a nuestro servidor.

2.2. Estrategia de Sincronización: Patrón Observador (Push)

El problema central era que una llamada remota (ej. `agregaOferta`) del Cliente A al Servidor no tenía forma de notificar al Cliente B. El *polling* (preguntar al servidor "hay cambios?çada 'x' segundos) nos pareció ineficiente. Por ello, nuestra solución fue implementar el **Patrón Observador (Observer Pattern)**.

El Servidor actúa como el "Sujeto"(Subject) y los Clientes como .ºbservadores"(Observers).

1. **Registro:** Cuando un cliente se conecta, no solo lo registramos como usuario, sino que también lo "suscribimos.ª las actualizaciones del servidor.
2. **Lista de Observadores:** Nuestro servidor mantiene una lista de todos los observadores suscritos (clientes conectados).

3. **Notificación (Push):** Cuando implementamos un evento de cambio de estado (como `agregaProductoALaVen` o `agregaOferta`), el servidor itera sobre su lista de observadores y les .^empuja"(push) una notificación.
4. **Actualización del Cliente:** El cliente, al recibir esta notificación asíncrona, es instruido para recargar su catálogo de productos desde el servidor, refrescando así su GUI.

Implementamos este patrón de forma idiomática en cada tecnología:

- **En RMI:** Lo implementamos mediante **Callbacks**. Definimos una nueva interfaz remota, `ClienteCallbackRM` que el cliente implementa. El cliente envía su propio stub al servidor, el cual almacenamos en un `Vector`. Cuando hay una actualización, el servidor invoca el método `notificarActualizacion()` en cada stub de cliente.
- **En gRPC:** Lo implementamos mediante **Server-side Streaming**. Definimos un RPC en el `.proto` llamado `suscribirseNotificaciones` que retorna un `stream` de mensajes. El cliente llama a este método una vez. El servidor almacena el objeto `StreamObserver` del cliente y, cuando hay una actualización, llama a `onNext()` en cada *observer* para enviar la notificación.

3. Principales Pasos para Desarrollar la Solución

El desarrollo de ambas soluciones (RMI y gRPC) siguió un proceso metodológico similar, partiendo de la base de código MVC. Los pasos generales que seguimos fueron:

1. **Definir el Contrato:** Primero definimos la interfaz entre cliente y servidor. Para RMI, fue una `java.rmi.Remote interface`. Para gRPC, fue el archivo `.proto`.
2. **Implementar la Lógica del Servidor:** Adaptamos el `SubastaModelo.java` original para que implementara el contrato remoto y gestionara el estado y la lista de observadores.
3. **Adaptar el Cliente:** Modificamos el `SubastaControlador` para que utilizara el stub (RMI o gRPC) y realizara llamadas remotas en lugar de locales.
4. **Implementar la Sincronización:** Programamos la lógica de suscripción y notificación (callbacks en RMI, streams en gRPC) en el cliente y el servidor.

Como parte de los requisitos de las décimas extras, los detalles técnicos, fragmentos de código y pasos específicos para *cada* implementación (RMI y gRPC) se encuentran documentados en el **Anexo A**.

4. Aspectos de Seguridad en el Desarrollo

En nuestro análisis, consideramos los aspectos de seguridad de ambas plataformas, aunque su implementación completa (como el cifrado) estaba fuera del alcance de esta práctica.

- **RMI:** Encontramos que la seguridad en RMI es compleja y anticuada. Requiere un `SecurityManager` y archivos de políticas (`.policy`) para definir permisos, lo cual es verboso. La funcionalidad de `codebase` es una vulnerabilidad conocida si no se configura adecuadamente, y la comunicación por defecto no está cifrada.

- **gRPC:** En contraste, gRPC está construido sobre HTTP/2 y promueve la seguridad como una característica central. Se integra nativamente con SSL/TLS para cifrar toda la comunicación. Para nuestras pruebas, utilizamos `usePlaintext()`, pero reconocemos que habilitar el cifrado en producción es trivial. Además, gRPC soporta mecanismos de autenticación modernos, como tokens JWT.

5. Presentación de los Datos: Interfaz del Sistema

Decidimos no modificar la interfaz de usuario (`SubastaVista.java`) visualmente, ya que el reto de la práctica era de backend y arquitectura, no de frontend. La GUI sigue presentando las cuatro zonas funcionales descritas en las instrucciones:

1. **Iniciar sesión:** Captura del nombre de usuario.
2. **Poner producto a la venta:** Nombre del producto y precio inicial.
3. **Obtener lista de productos:** Lista de productos y precio actual.
4. **Hacer una oferta:** Monto ofrecido.

El cambio fundamental que logramos no es visual, sino funcional. Gracias a nuestra solución de sincronización (callback/stream), la "Zona de lista de productosz el "Precio actual.ªhora se actualizan de forma reactiva. Cuando un cliente B realiza una oferta, nuestro servidor envía la notificación. El cliente A la recibe y su controlador recarga la lista y el precio, reflejando el nuevo estado del sistema sin intervención del usuario.

6. Demostración de Ejemplos y Análisis (SO6)

Acorde a la rúbrica SO6 ("Diseño de experimentosz .Análisis de datos"), diseñamos un experimento para validar nuestra solución al problema de sincronización.

6.1. Diseño del Experimento

El diseño experimental que aplicamos fue idéntico para RMI y gRPC, para poder comparar los resultados:

1. **Inicio:** Lanzamos la aplicación del Servidor.
2. **Conexión Cliente A:** Iniciamos un cliente (ej. Çliente 1: Camila") y nos conectamos. Verificamos en el servidor que se registra la conexión y la suscripción.
3. **Conexión Cliente B:** Iniciamos un segundo cliente (ej. Çliente 2: Celeste") y nos conectamos. Verificamos que el servidor registra al segundo suscriptor.
4. **Acción (Cliente A):** El Cliente A pone un producto a la venta (ej. Ïmpresora"por 3500).
5. **Verificación (Cliente B):** Verificamos que la GUI del Cliente B se actualiza *automáticamente*, mostrando el nuevo producto Ïmpresoraz su precio, sin haber presionado ningún botón.
6. **Acción (Cliente B):** El Cliente B realiza una oferta por la Ïmpresora"(ej. 3600).
7. **Verificación (Cliente A):** Verificamos que la GUI del Cliente A actualiza el precio actual de la Ïmpresora.ª 3600, de nuevo, sin acción manual.

6.2. Análisis de Datos (SO6)

El experimento fue un éxito en ambas plataformas. **Los datos principales** que obtuvimos fueron los logs de las terminales y las capturas de pantalla de las GUI.

Como se demuestra en las capturas de pantalla detalladas en el **Anexo A** (ver Figuras 2, 3 y 5), los logs del servidor y del cliente confirman la recepción de notificaciones asíncronas (la línea ÇALLBACK RECIBIDO.^{en} RMI y los logs de .^{En}viando notificación.^{en} gRPC). Los datos de la GUI (específicamente la GUI del Cliente 2 en la Fig. 5 y el callback en la Fig. 2) validan inequívocamente que nuestra implementación del patrón observador resolvió el problema de sincronización.

7. Análisis y Comparación: RPC, RMI y gRPC

Tal como lo solicita la práctica, realizamos un análisis comparativo entre las tecnologías, suponiendo un RPC tradicional.

- **RPC (Tradicional):**
  - **Ventajas:** Es el concepto más simple y es agnóstico al lenguaje (si usa un IDL).
  - **Desventajas:** Es un paradigma puramente síncrono (solicitud-respuesta). Resolver el problema de sincronización de esta práctica habría requerido *polling* (el cliente preguntando "¿hay cambios?çada segundo). Consideramos que esto es altamente ineficiente, genera alta latencia y no escala.
- **Java RMI (Implementado):**
  - **Ventajas:** Su integración con Java es nativa. Nos permitió pasar objetos Java serializables completos. El patrón de callback fue una solución natural y elegante para nosotros dentro de un ecosistema puramente Java.
  - **Desventajas:** Es una tecnología legada, limitada exclusivamente a Java. Su configuración (`rmiregistry`, `codebase`, políticas de seguridad) nos pareció anticuada y verbosa en comparación con gRPC.
- **gRPC (Implementado):**
  - **Ventajas:** Es una tecnología moderna de alto rendimiento (HTTP/2, Protocol Buffers). Es agnóstica al lenguaje (definida por `.proto`). Su soporte nativo para **streaming** (unidireccional y bidireccional) fue la herramienta perfecta para este caso de uso. Nos permitió una comunicación asíncrona y reactiva de forma muy eficiente.
  - **Desventajas:** Requiere un paso de compilación/generación de código. La gestión de la comunicación asíncrona (streams, observadores) añadió una capa de complejidad al código cliente que no tuvimos con RMI.

8. Conclusiones (Síntesis SO6)

Hemos cumplido en su totalidad los objetivos de la práctica. Logramos migrar exitosamente la aplicación MVC centralizada a dos arquitecturas distribuidas funcionales, implementando la solución en RMI y gRPC.

La *síntesis de la información* (rúbrica SO6) que obtuvimos de nuestros experimentos (Sección 6 y Anexo A) nos permite concluir que el problema de sincronización de estado fue resuelto exitosamente

en ambas plataformas. Nuestra estrategia del patrón observador fue validada: los callbacks de RMI y los streams de gRPC demostraron ser mecanismos eficaces para .empujar"(push) las actualizaciones de estado a los clientes en tiempo real.

Aplicando nuestro *juicio de ingeniería* (rúbrica SO6), concluimos que **gRPC es la solución tecnológicamente superior**. Aunque nuestra implementación de RMI fue funcional y elegante dentro de su ecosistema cerrado, gRPC nos ofreció una solución más robusta, de mayor rendimiento e interoperable, cuyo soporte nativo para streaming se alinea perfectamente con los requisitos de las aplicaciones distribuidas modernas y reactivas.

A. Anexo: DECIMAS EXTRAS (Implementación y Evidencia)

Para la obtención de las décimas extras, nuestro equipo implementó la aplicación completa en **ambas** plataformas (RMI y gRPC). A continuación, detallamos los pasos de implementación y presentamos la evidencia experimental (capturas de pantalla) que demuestra el funcionamiento de ambas soluciones.

A.1. Implementación RMI

Seguimos los siguientes pasos para la versión de RMI:

1. **Definición de Interfaces:** Creamos dos interfaces `java.rmi.Remote`: `SubastaServidorRMI` (para las operaciones principales: ofertar, registrar, etc.) y `ClienteCallbackRMI` (con un solo método, `notificarActualizacion()`).
2. **Serialización:** Modificamos las clases de datos `InformacionProducto` e `InformacionOferta` para que implementaran `java.io.Serializable`.
3. **Implementación del Servidor:** Creamos `SubastaModeloRMI` extendiendo `UnicastRemoteObject` e implementando `SubastaServidorRMI`. Añadimos un `Vector<ClienteCallbackRMI>` para gestionar los callbacks.
4. **Implementación del Cliente:** Modificamos el `SubastaControladorRMI` para buscar el stub del servidor en `rmiregistry`. El cliente también implementó la interfaz de callback y se registró a sí mismo en el servidor al conectarse, enviando su propio stub.

A.2. Implementación gRPC

Para la versión de gRPC, el proceso fue:

1. **Definición de Contrato (IDL):** Creamos el archivo `subasta.proto`. En él, definimos los mensajes (ej. `OfertaRequest`, `Producto`) y los servicios (`rpc AgregaOferta`).
2. **Definición del Stream:** El paso crucial fue definir el stream para las notificaciones: `rpc suscribirseNotifi returns (stream NotificacionUpdate);`.
3. **Generación de Código:** Usamos Gradle y el plugin de `protobuf` para generar el código stub de Java.
4. **Implementación del Servidor:** Creamos `SubastaServicioImpl` extendiendo la clase base generada por gRPC. Migramos la lógica del modelo aquí, incluyendo un `ConcurrentHashMap` para almacenar los `StreamObserver` de los clientes suscritos.
5. **Implementación del Cliente:** Creamos `ClienteGPRC`, que instancia un `ManagedChannel` para conectarse al servidor. El `SubastaControladorGPRC` usa el stub bloqueante para acciones (como ofertar) y el stub asíncrono para suscribirse al stream de notificaciones en un hilo separado.

A.3. Evidencia Experimental RMI

Las siguientes figuras demuestran el funcionamiento de nuestra implementación RMI.

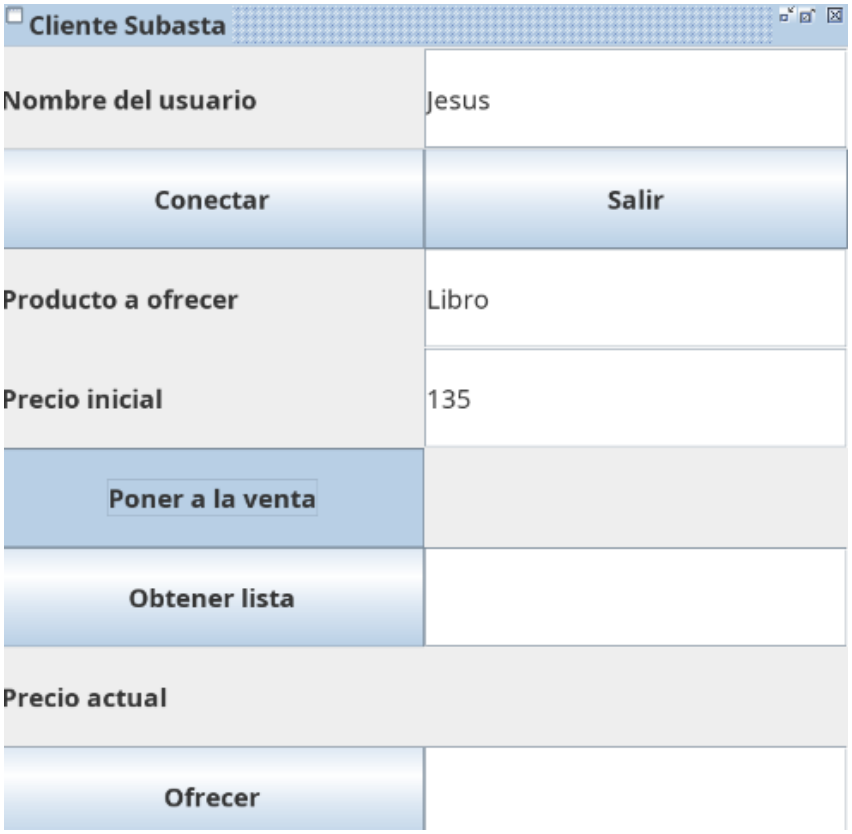

Figura 1: RMI: Cliente 1 ("Jesus") poniendo a la venta un "Libro"por 135.

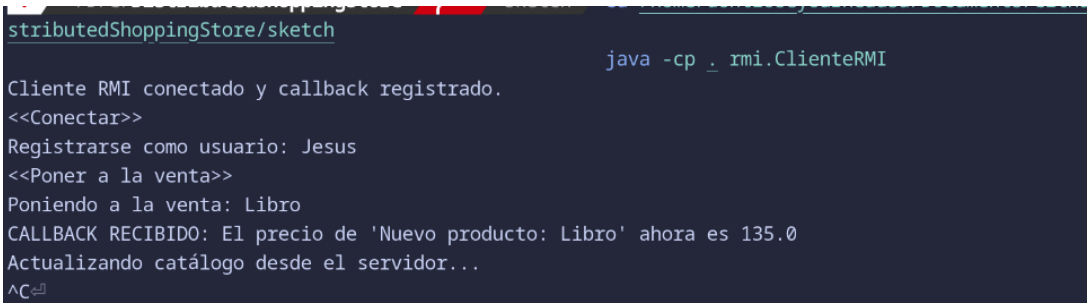

Figura 2: RMI: Terminal del Cliente 1. La línea ÇALLBACK RECIBIDO"demuestra que el servidor notificó exitosamente al cliente sobre el nuevo producto.

A.4. Evidencia Experimental gRPC

La siguiente figura captura la demostración completa de nuestra implementación gRPC, mostrando al servidor y dos clientes interactuando.

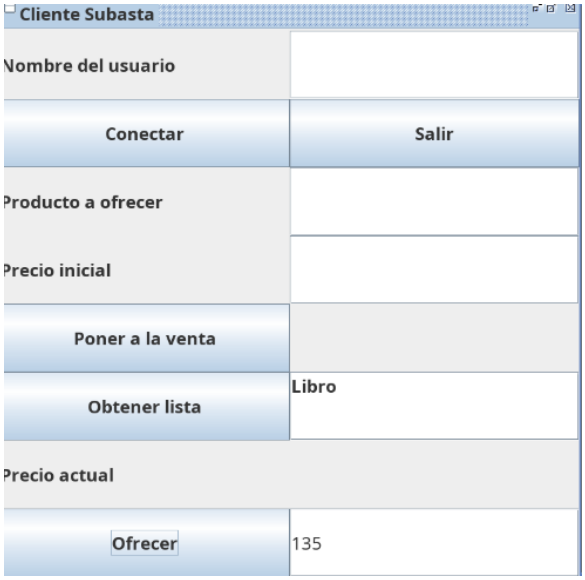

Figura 3: RMI: Cliente 2 visualiza el "Libroz prepara una oferta de 135.

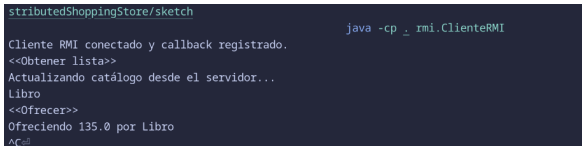

Figura 4: RMI: Terminal del Cliente 2, registrando la acción .ºfrecer".

Figura 5: gRPC: Demostración completa. El servidor (abajo) registra a Çamilaz Çeleste". Camila (derecha) vende una İmpresora". El servidor envía la notificación y la GUI de Celeste (izquierda) se actualiza automáticamente .

Referencias

[1] A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms (4th Edition)*. Pearson, 2024.

[2] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design (5th Edition)*. Addison-Wesley, 2011.

[3] The gRPC Authors, "gRPC Documentation: Core Concepts, Architecture and Lifecycle."`https://grpc.io/docs/what-is-grpc/core-concepts/` [Consultado: 25 de octubre de 2025].

[4] Oracle Corporation, "Java RMI Documentation: An Overview."`https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html` [Consultado: 25 de octubre de 2025].