

Sistemas Distribuidos

Trabajo práctico 3: RMI

Fecha de entrega: 30 de abril de 2019

Carlos Andrés Reyes Evangelista 157068

Mariela Sánchez Alvarez 157930

Erick Siordia Nagaya 157504

Contenido

[Plan del Documento 3](#_Toc7560813)

[Objetivo 3](#_Toc7560814)

[Implementación 3](#_Toc7560815)

[Diseño 3](#_Toc7560816)

[Estructura final 4](#_Toc7560817)

[Paquete Mediator 5](#_Toc7560818)

[Paquete RemoteObjects 6](#_Toc7560819)

[Paquete RemoteInterfaces 6](#_Toc7560820)

[Paquete Client 7](#_Toc7560821)

[Paquete Server 13](#_Toc7560822)

[Paquete Style 14](#_Toc7560823)

[Discusión y conclusiones 15](#_Toc7560824)

# Plan del Documento

El presente trabajo se divide en 3 secciones principales: objetivo, implementaicón y discusión. En el objetivo se presenta el motivo que impulsa el desarrollo de este proyecto. Después, en la implementación se explica el proceso para el desarrollo de la aplicación distribuida, como también el funcionamiento de los componentes de software. Finalmente, en la última sección se presentan las lecciones aprendidas durante el desarrollo del proyecto, y las conclusiones a las que el equipo llegó.

# Objetivo

El proyecto desarrollado tiene como objetivo demostrar el uso del mecanismo Remote Method Invocation (RMI) de Java, para la implementación de una aplicación de subastas donde los clientes se registren en un servidor para ofrecer productos y hacer pujas a los productos ofrecidos por otros usuarios. Llevando a cabo esta tarea, podremos observar en la práctica los conceptos teóricos analizados en el curso como callback, invocaciones remotas, serialización y marshalling/unmarshalling, entre otros.

# Implementación

## Diseño

Para poder implementar la aplicación, primero se pensaron en varias formas de distribuir la aplicación centralizada de ofertas. El equipo tuvo 2 ideas para llevar a cabo la distribución:

* Propuesta 1:
  1. El servidor administra diferentes objetos *servants*. Cada uno de ellos contiene una estructura de datos para almacenar ya sea los datos de los usuarios o los productos.
  2. El cliente contiene objetos locales del Model, View y Controller. El objeto Model contiene los atributos clientes y productos, que hacen referencia a los objetos *servant* administrados por el servidor. El controller conecta un objeto local de Model y de View.
* Propuesta 2:
  1. El servidor administra un solo objeto *servant* que contiene estructuras de datos donde se guarda la información de los clientes y de los productos ofrecidos.
  2. El cliente contiene objetos locales de tipo Controller y de View. El objeto de tipo Controller conecta el objeto View, y un stub del objeto Model obtenido a través del servidor de nombres de Java.

Después de plantear las ventajas y desventajas de cada propuesta, se decidió implementar una mezcla de ambas propuestas. El resultado fue el siguiente:

* Propuesta implementada:
  1. El servidor administra un objeto *servant* que contiene estructuras de datos donde se guarda la información de los clientes y de los productos ofrecidos.
  2. El cliente contiene objetos locales de tipo Controller y de View. El objeto tipo Controller conecta el objeto View, y un stub del objeto Model obtenido a través del servidor de nombre de Java.
  3. El cliente contiene también un objeto que actúa como mediador entre él mismo y el servidor. Esto fue resultado de un análisis donde se concluyó que es necesario tener un mediador debido a que se necesita conocer el nombre del usuario que está operando la aplicación en el cliente. El mediador se encarga de adaptar las llamadas del cliente (que no tienen el nombre del usuario), a llamadas donde el servidor puede identificar al cliente (que contienen el nombre del usuario).

## Estructura final

El proyecto final está conformado por 6 paquetes, cada uno de ellos encapsula un conjunto de clases o interfaces que conciernen a una abstracción independiente cuyas clases se relacionan entre sí. A continuación, será presentada una relación de cada una de las clases implementadas, su función, su razón de ser y las relaciones que mantiene con las demás clases. Es importante destacar que la descripción de estas clases será realizada únicamente en aspectos globales y relacionados directamente a la clase, si se requiere más detalle acerca del funcionamiento específico de alguno o todos los componentes, se exhorta al lector a acudir ya sea a la documentación en formato *Java Docs* adjunta en este proyecto o directamente al código también anexado que cuenta con comentarios descriptivos en cada una de las clases, interfaces y métodos.

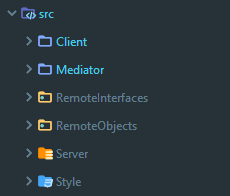
**

Ilustración 1: Estructura final

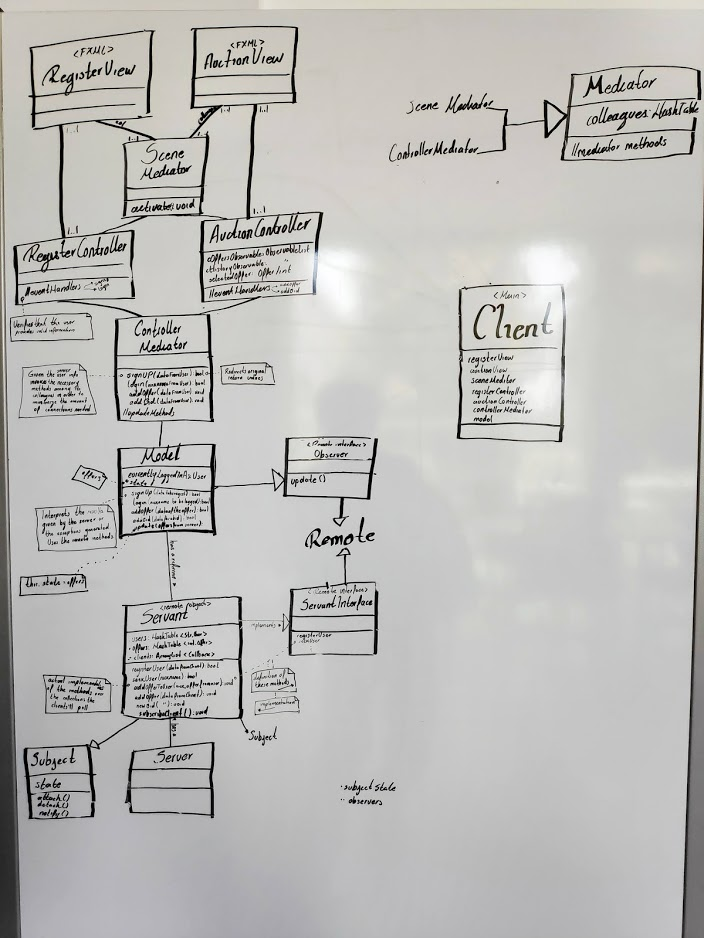


Ilustración : Arquitectura

## Paquete Mediator

Este paquete cuenta únicamente con una clase abstracta homónima que define los métodos y atributos necesarios para ser implementados por mediadores concretos. Esta abstracción corresponde a un patrón de diseño de software de nombre Mediador, en esencia este objeto se encarga de definir un objeto que controla como un conjunto de otros objetos, denominados colegas, interactúa entre sí.

El objetivo de este patrón de diseño es aligerar la carga de conexiones que puede vincular a estos objetos al sustituir asociaciones de ‘muchos a muchos’ por asociaciones de ‘uno a muchos’, estas últimas son mucho más fáciles de controlar y de variar independientemente. Los objetos mediadores contarán con referencias a todos los objetos implicados y estos únicamente contarán con una referencia al mediador respectivo, de esta manera, todas las invocaciones corren por un solo objeto quien las redirecciona a los objetos necesarios.

Mediator es implementada por *SceneMediator* y *ControllerMediator* en el paquete Cliente*.*

## Paquete RemoteObjects

Este paquete únicamente contiene clases que representan objetos concretos que serán utilizados tanto por el lado del cliente como del lado del servidor. Específicamente se trata de los objetos:

* User
* Offer
* Bid

Cada uno representa una abstracción diferente requerida específicamente para esta aplicación, sus implementaciones no tienen métodos dignos de discusión más que la mención de que todos ellos implementan la interfaz Serializable pues copias de ellos serán transmitidas mediante RMI a lo largo de la ejecución del programa.

**User** representa un solo usuario, contiene información personal de ese usuario y una lista de las ofertas que ha publicado. **Offer** representa una sola oferta publicada, contiene información relacionada con la oferta y contiene una lista que funge como histórico de los pujes que se han realizado sobre dicha oferta. **Bid** representa un solo puje realizado sobre una oferta específica, únicamente guarda información sobre el nombre de usuario que realizó ese puje y la cantidad establecida.

## Paquete RemoteInterfaces

Este paquete contiene un par de interfaces remotas utilizadas en el proyecto. Ambas interfaces extienden la clase *Remote* y los métodos definidos en ellas arrojan excepciones de tipo remoto lo que permite a las clases que implementen estas interfaces que sus objetos sean susceptibles de recibir llamadas mediante la plataforma RMI. Estas dos interfaces son:

* Observer
* ServantInterface

La interfaz **Observer** define métodos que corresponden al patrón de diseño del mismo nombre, Observador, este patrón de diseño es una abstracción similar a la arquitectura publicador-consumidor, donde un objeto, determinado Asunto (Subject) contiene un atributo de nombre Estado y una lista de Observadores, cuando el Estado cambia, el Asunto debe informar a todos los Observadores de este cambio. Esta modelo se prestó de manera perfectamente adecuada para implementar la técnica de *Callback* en este proyecto. Esta interfaz en particular se encarga de definir un método update() que puede ser llamado de manera remota por el Asunto –que, en este ejemplo particular, será el Servant detallado más adelante–.

Observer es implementada por *Model* en el paquete Cliente.

**ServantInterface** es una interfaz remota que define métodos de tres categorías diferentes: métodos para administrar usuarios, métodos para administrar ofertas y métodos correspondientes al Asunto descrito en el punto anterior. Esta interfaz solía estar separada en cada una de las categorías enumeradas para fomentar la modularización por abstracciones, pero debido a una decisión de diseño elaborada por los miembros del equipo a causa también de una sugerencia superior fue unida en una sola interfaz que permitiese la existencia de un único objeto remoto que incluyera todas las características de los existentes. Los métodos para administrar a los usuarios incluyen registrar y buscar uno, los relativos a las ofertas refieren a añadir un nuevo producto como oferta y hacer un puje a uno ya existente y finalmente, los métodos relacionados al patrón de diseño Observer incluyen subscripciones, desubscripciones de observadores y envío de notificaciones a todos los actualmente suscritos.

ServantInterface es implementada por *Servant* en el paquete Server.

## Paquete Client

Este paquete contiene todas las abstracciones de las clases que atañen a la parte que representa al cliente en este proyecto. La parte Cliente está programada bajo un paradigma *Model – View – Controller* y, además, implementa ciertos *patrones de diseño de software* que permiten al código ser más flexible, modularizado y acorde a estándares ampliamente conocidos. Otra aclaración que vale la pena realzar es que la interfaz gráfica para este proyecto está desarrollada en *Java FX*, razón por la cual los archivos correspondientes a las vistas están almacenados con una extensión *.fxml* en lugar de *.java*. El paquete está conformado por las siguientes clases o módulos -ordenadas en orden de creación por la aplicación-:

* Main
* RegisterView
* RegisterController
* AuctionView
* AuctionController
* SceneMediator
* Model
* ControllerMediator

El módulo **Main** representa la aplicación diseñada para ser ejecutada por el cliente, contiene el método main necesario para ejecutar el programa y está separado de la abstracción e implementación del paquete para fomentar la creación de más aplicaciones Cliente para diferentes requerimientos sin la necesidad de contaminar el código que contiene las abstracciones necesarias para este fin. Este módulo es responsable de crear las vistas de la aplicación Java FX con sus métodos requeridos, extraer los Controladores adjuntos a estas y asignarlos a instancias controlables, crear los Mediadores requeridos y configurarlos, crear una instancia del Modelo y asegurarse de que ésta es registrada en el servidor para ser susceptible de recibir actualizaciones por parte del servidor gracias al *callback* implementado –del que se hablará más adelante también–, mostrar la primera vista al usuario y finalmente, controlar el cierre que al cierre de la aplicación, sea enviada una señal al servidor en seña de que el cliente se desconectará y no necesita recibir más actualizaciones.

El archivo **RegisterView** contienen el código en estilo XML necesario para crear la interfaz gráfica de la primera parte de la aplicación, referente a la registración de nuevos usuarios y el inicio de sesión de aquellos que ya tienen una cuenta activa en el servidor. Esta interfaz gráfica tiene el siguiente aspecto:

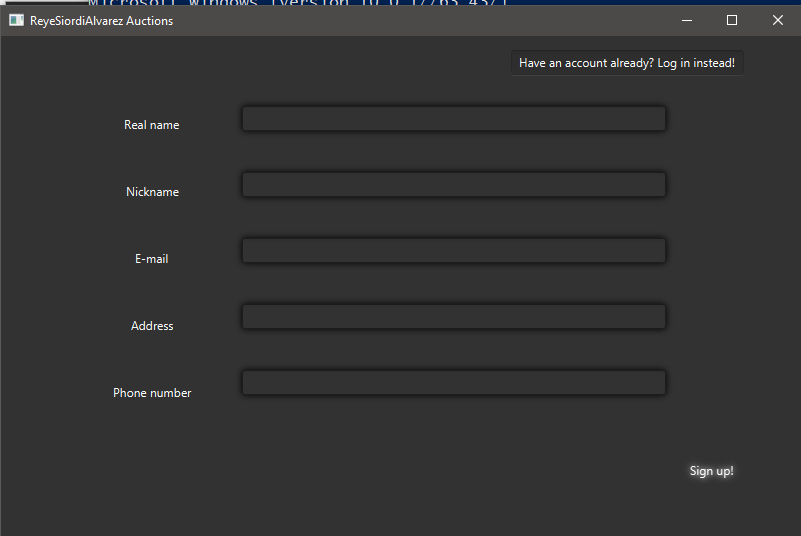


Ilustración : Register View

RegisterView envía datos a la clase: *RegisterController*. Recibe datos de la clase: *Register Controller*.

La clase **RegisterController** se encarga de controlar la entrada del usuario sobre la interfaz gráfica recientemente descrita, por ejemplo, que el usuario no trate de registrarse sin introducir un nombre y un nombre de usuario. Una vez que este Controlador decide que los datos cumplen el formato requerido solicita al Mediador de Controladores que envíe la información al Modelo e interpreta los resultados obtenidos de este para mostrar lo que está pasando al usuario, si el inicio de sesión es correcto, solicita al Mediador de Escenas que active la siguiente vista.

RegisterController envía datos a la clase: RegisterView, *Model* mediante *ControllerMediator*, *AuctionView* mediante *SceneMediator*. Recibe datos de las clases: *SceneMediator* provenientes de *Model*.

El archivo **AuctionView** contiene código en estilo XML necesario para crear la interfaz gráfica de la segunda parte de la aplicación, donde el usuario puede ofertar productos, hacer pujes a los productos ya existentes propuestos por cualquier cliente en la sesión y observar el progreso de sus propios productos. Esta interfaz gráfica tiene el siguiente aspecto en sus múltiples opciones:

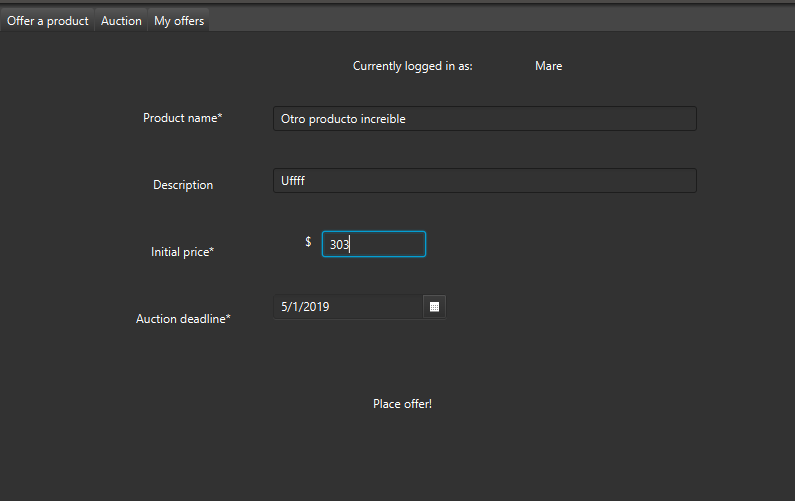


Ilustración : Auction View "Offer a product"

AuctionView envía datos a las clases *AuctionController*. Recibe datos de la clase: *AuctionController.*

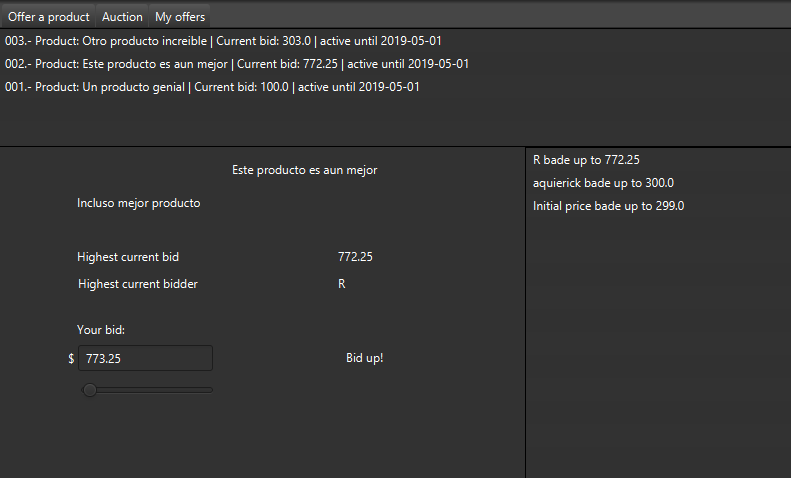


Ilustración : Auction View "Auctions"

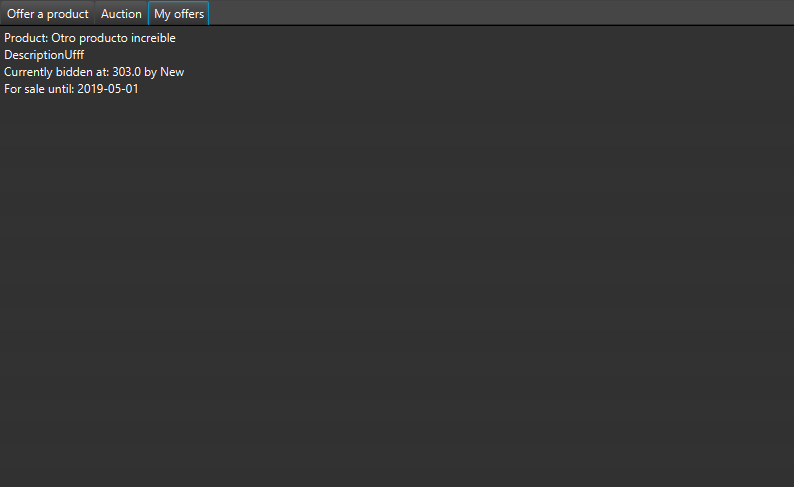


Ilustración : Acution View "My offers"

La clase **AuctionController** se encarga de controlar la entrada del usuario sobre la interfaz gráfica correspondiente a las últimas vistas mostradas, por ejemplo, que el usuario no intente registrar una oferta sin precio inicial o trate de hacer un puje con un valor menor al mayor puje actual o incluso que trate de elegir fechas de termino de oferta anteriores al día en que fueron propuestas. Una vez que el Controlador decide que los datos cumplen el formato requerido solicita al Mediador de Controladores que envíe la información al Modelo e interpreta los resultados recibidos para brindar retroalimentación al usuario.

AuctionController envía datos a las clases: *Model* mediante *ControllerMediator* y *AuctionView*. Recibe datos de las clases Model mediante ControllerMediator.

Las clases **SceneMediator** y **ControllerMediator** sonimplementaciones concretas del patrón de diseño de nombre Mediador, las características y ventajas de esta implementación son explicadas en puntos anteriores. En este caso particular, cada Controlador debería tener referencias a su propia vista, al otro Controlador y al Modelo, mientras que este tendría conexiones a todos los demás objetos implicados, sin contar las referencias entre las vistas requeridas; con Mediador implementado, cada clase tiene una única referencia a su mediador requerido. En este ejemplo tal vez represente una ventaja ínfima, pero conforme el proyecto crezca, la ventaja también lo hará de manera potencial.

Finalmente, la clase **Model** implementa la lógica aplicativa de cada cliente para las funciones de registrar un usuario, iniciar sesión, añadir un nuevo producto como oferta y hacer pujes a ofertas ya existentes. Es el Modelo quien cuenta con una referencia al objeto remoto *Servant* y es su responsabilidad realizar las llamadas pertinentes al mismo con los datos recibidos por los Controladores y reenviar a estos resultados a interpretar con base en los resultados o excepciones de las llamadas al servidor. Es el Modelo también quien hace uso de los métodos que el servidor ofrece para subscribirse y desubscribirse para recibir *Callbacks,* del mismo modo implementa los métodos requeridos para que el servidor -quien contará con una referencia a él- pueda mandar las actualizaciones pertinentes. Dado que la clase Model es subclase de UnicastObjectRemote, cada vez que una instancia de esta clase es creada, ésta ya fue automáticamente exportada como objeto remoto y funciona como una referencia RMI inmediatamente.

Model implementa *Observer* y extiende *UnicastRemoteObject*. Recibe datos de: *RegisterController* y *AuctionController* mediante *ControllerMediator*, envía datos por el mismo medio a los mismos destinatarios. Se comunica de manera remota con: *Servant*, de quien recibe información a su vez.

## Paquete Server

Este paquete incluye abstracciones relevantes al servidor de la aplicación. Está conformado por las clases:

* Main
* Servant
* Server

El módulo Main corresponde a la aplicación de comando de líneas diseñada para ser ejecutada de lado del servidor, contiene el método main requerido para que el servidor sea lanzado y se encarga de la creación de objetos relacionados con el servidor controlada. Cuando esta aplicación es lanzada se es creada una instancia del servidor de nombres de RMI, RMIRegistry, en el puerto 1099 (puerto por defecto), declara una instancia del servidor (Server) que automáticamente se inicializará a sí misma y al Servant requerido –que ya habrá sido exportado automáticamente– y obtiene la referencia al Servant que contiene ese Servidor para su posterior vinculación en el RMIRegistry ya creado en ese punto. A continuación, el servidor dará su trabajo por terminado y únicamente esperará por llamadas externas ante las cuales ya tendrá forma de responder.

La clase Servant representa la implementación de los servicios que este servidor pretende proveer. Servant mantiene colecciones de objetos donde almacena los datos de la ejecución del actual servidor; entre estas colecciones posee: una lista de usuarios registrados (objetos tipo User), una lista de ofertas publicadas (objetos tipo Offer) y finalmente una lista de Observadores (referencias remotas a objetos de tipo Model a quienes puede realizar llamadas remotas de sus métodos).

Esta clase implementa todos los servicios declarados en la interfaz Servant, por lo que tiene la capacidad de agregar usuarios, buscarlos, agregar ofertas y vincularlas al usuario indicado, añadir pujes a los históricos de las ofertas precisadas y además brinda las implementaciones a los métodos attach() y detach() que permitirán a los Clientes subscribirse para recibir las notificaciones que se generarán cuando ocurra un cambio a la colección de ofertas publicadas que desencadenará el método notifyClients() que, basado en la lista de clientes suscritos, utilizará el método update() definido en Observer (paquete Observer) e implementado en Model (paquete Client) para enviar a todos los clientes las versiones actualizadas de las ofertas publicadas y, además, a cada cliente le pedirá el nombre de usuario del cliente que actualmente inició sesión para enviarle su propia versión de ese usuario que seguramente habrá cambiado del de la copia local que cada cliente tiene de su usuario, incluyendo información sobre nuevos pujes hechos a sus ofertas. Derivado del hecho de que esta clase extiende UnicastRemoteObject, al utilizar su constructor, el objeto que devuelve ya ha sido exportado de manera automática, por lo que está listo para ser vinculado en el RMIRegistry.

Servant implementa ServantInterface y extiende UnicastObjectRemote. Se comunica remotamente con Model de quien recibe referencias remotas y a quien envía actualizaciones.

Para finalizar, la clase Server funciona como un caparazón que protege y administra a los Servants que implementan sus servicios, en este caso es únicamente uno, así que su existencia podría parecer banal, pero esta estructura permite al proyecto crecer a voluntad en futuras actualizaciones. Esta clase está implementada con una estrategia llamada Singletón, que impide que más de una instancia de esta clase sea creada y, en caso de haber sido intentado, se es regresada la instancia ya previamente creada.

## Paquete Style

Este paquete únicamente incluye un archivo de estilo CSS que modifica la apariencia de la aplicación, esto permite que en el futuro se incluyan temas de manera fácil y efectiva. Este estilo modifica la apariencia original, mostrada a continuación, por la desplegada previamente.

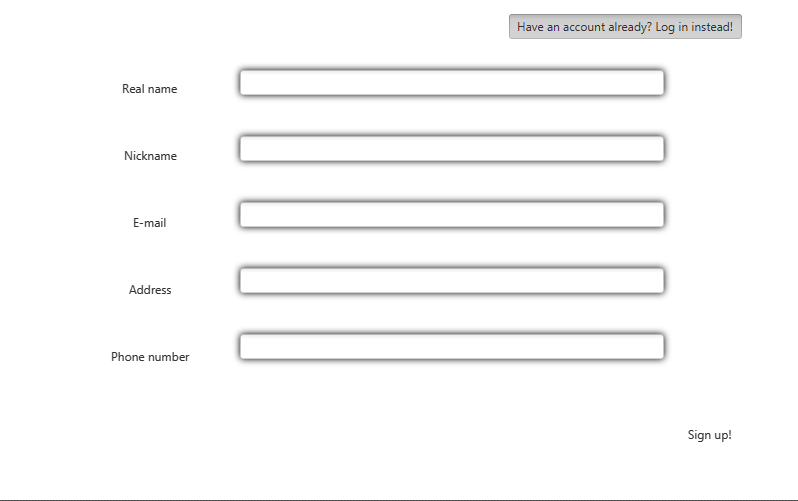


Ilustración : Apariencia Original

# Discusión y conclusiones

Durante el desarrollo del presente proyecto se tuvieron que afrontar varios obstáculos debido a los diferentes puntos de vista de los integrantes sobre las decisiones de diseño de la aplicación.

Una de las lecciones importantes durante el proyecto fue el hecho de conocer que existen ocasiones donde existe más de una idea válida. El analizar todas las propuestas de diseño para saber cuál era la más conveniente para el contexto de nuestra apicación fue lo que nos llevó a implementar la combinación de ambas propuestas debido a que las dos proponían ideas que facilitaran una implementación más optima.

Además, se pudo experimentar con el mecanismo de RMI de la plataforma de Java, y se logró implementar una aplicación Java RMI que implementara un Callback entre el cliente y el servidor. Gracias a lo anterior, se logró tener una mejor comprensión de los casos de uso que puede tener un sistema distribuido basado en esta tecnología.

Nota: dentro de los archivos adjuntos se encuentra un archivo README que especifica la forma de correr el programa para realizar las pruebas tanto de manera local como de manera distribuida.

Requerimientos:

Java 8 (si se posee una versión superior, asegúrese de contar con un JDK que incluya JavaFX o instale OpenJFX)