Redes de Comunicaciones II

MEMORIA PRÁCTICA 2: SAIMAZOOM

Fecha de entrega: 12 de Abril de 2023
Grupo 2311

Equipo 01
Ismael Iván López Murillo
Óscar Navalón Navarro

1. INTRODUCCIÓN

Este proyecto consiste en la implementación de una aplicación de simulación de entrega de paquetes inspirada en el servicio de Amazon. La aplicación se divide en diferentes componentes que incluyen un cliente, un controlador, robots y el servicio de entrega en sí. La interacción entre estos componentes se realiza mediante colas de mensajes.

El objetivo principal de esta aplicación es proporcionar una simulación del proceso de entrega de paquetes de Amazon(Saimazoom), donde los usuarios pueden enviar solicitudes de entrega y hacer un seguimiento del progreso de sus pedidos. Para lograr este objetivo, se ha desarrollado una arquitectura basada en colas de mensajes, que permite la comunicación y coordinación entre los diferentes componentes de la aplicación, además de facilitar la escalabilidad de la aplicación, aumentar su tolerancia a fallos y simplificar su desarrollo..

En este trabajo, se describe en detalle la arquitectura de la aplicación, incluyendo su diseño y las tecnologías utilizadas. También se presentarán los diferentes componentes de la aplicación y cómo interactúan entre sí para lograr el objetivo final

2. DOCUMENTACIÓN DE DISEÑO

a. Especificación de requisitos funcionales

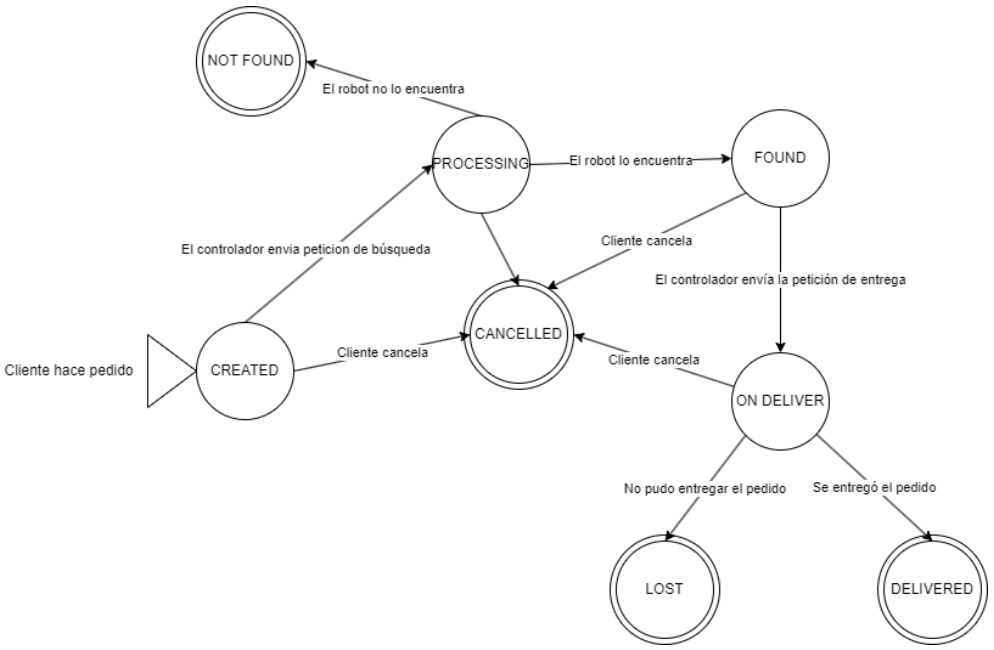
i. Un usuario puede registrarse en el sistema proporcionando nombre, nombre de usuario y contraseña
ii. Para que un usuario pueda hacer pedidos, deberá de acceder con su nombre de usuario y contraseña
iii. Un usuario podrá consultar sus pedidos, mostrando su ID, estado, total y pedidos
iv. Un usuario podrá consultar el detalle de un pedido, donde además de los datos del requisito previo, se podrá ver un registro de los cambios de estado en el pedido
v. Un usuario podrá crear pedidos, en donde se le pedirá escribir en texto plano una cantidad “n” de artículos, siendo “n” personalizable por el usuario.
vi. Un usuario podrá crear cualquier cantidad de pedidos
vii. Por simplicidad, los pedidos tendrán un coste aleatorio
viii. El controlador deberá recibir las peticiones de los usuarios (mediante una cola específica), así como las confirmaciones de robots y repartidores (mediante colas específicas)
ix. El controlador, asimismo, tendrá colas para comunicarse con robots y repartidores sin esperar respuesta (de forma asíncrona)
x. El controlador mantendrá toda la información y su estado, tanto de usuarios y pedidos, en una base de datos relacional
xi. El controlador, al recibir una solicitud de pedido, creará dicho pedido y posteriormente enviará la solicitud de búsqueda al almacén a los robots

xii. Los robots recibirán las peticiones de búsqueda de artículos, donde deberán buscar todos los artículos en un tiempo entre [t_min, t_max] con una probabilidad “p” de encontrarlo. Si no encuentran un artículo, todo el pedido quedará como no encontrado
xiii. El controlador recibirá las respuestas de los robots (mediante una cola específica), en donde, dependiendo de el éxito o fracaso, se actualizará la información y se enviará la orden de entrega a los repartidores
xiv. Los repartidores recibirán las peticiones de entrega de órdenes, donde deberán buscar todos los artículos en un tiempo entre [t_min, t_max] con una probabilidad “p” de entrega. Dependiendo de su resultado, se dará aviso al controlador
xv. Los clientes pueden cancelar los pedidos en cualquier momento, aunque esto no implica que estos pedidos serán efectivamente cancelados
xvi. Al realizarse una cancelación, el controlador recibirá el pedido y determinará si se puede cancelar (en caso de no haber sido determinado en un estado final), se avisará tanto a robots como a repartidores para que hagan efectiva dicha cancelación
xvii. En cuanto dicho pedido sea recibido por robots o repartidores, este enviará una confirmación al controlador, quien deberá avisar a todos que dicho pedido ya no necesita ser cancelado.

b. Especificación de requisitos no funcionales

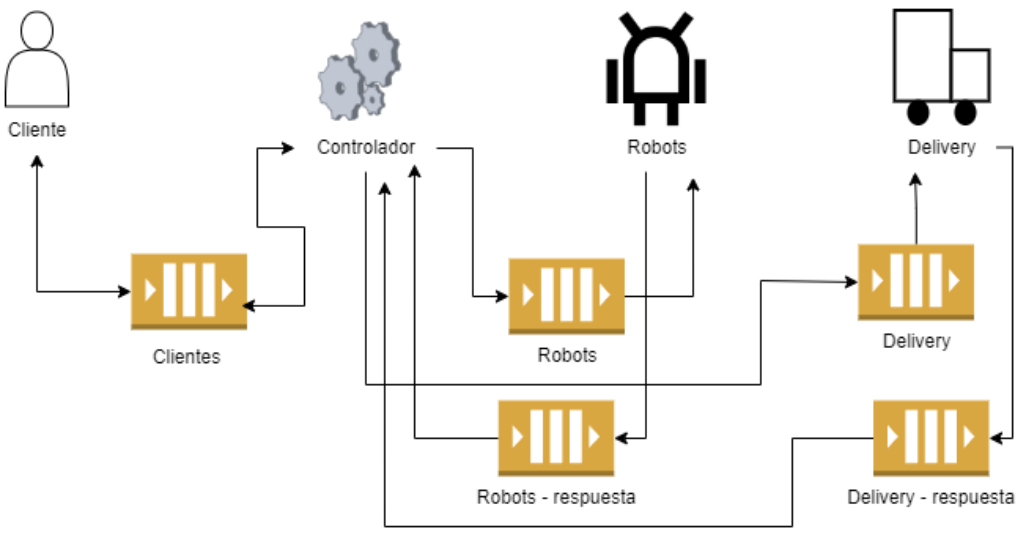
i. Se requiere del intérprete Python 3.6 o superior
ii. Los actuadores pueden ejecutarse ya sea en sistemas Linux o en sistemas Windows
iii. Se requieren tener instaladas las librerías especificadas en requires.txt, además de las librerías estándar de Python,
iv. Se requiere de una conexión a Internet y la red de UAM para correr el programa de forma correcta
v. Se requiere de un espacio mínimo de disco de 100MB para guardar el código fuente y la base de datos Sqlite
vi. El sistema empleado para mantener los datos persistentes será SQLite
vii. Tanto los tiempos de espera como las probabilidades deberán estar definidos en un archivo de configuración de Python
viii. Los mensajes recibidos en la cola serán procesados siempre que haya un actor correspondiente (consumidor) que pueda recibirlo
ix. En caso de no haber un consumidor, el mensaje permanecerá hasta que exista uno disponible
x. Todos los consumidores de mensajes atenderán un mensaje a la vez, asegurando el fair dispatch.
xi. Los mensajes enviados a las colas se conservarán mientras el servidor de colas permanezca activo

c. Diagrama de estados de pedido



d. Diseño del uso de colas y mensajes

Los actores se comunican mediante las siguientes colas:



Al principio podría parecer un diseño redundante, pero cada sección tiene su razón de ser y su arquitectura específica.

1. **Clientes**: Esta cola funciona para que el cliente pueda generar mensajes (productor) que el controlador procese (consumidor), y es tipo RPC, ya que en algunas ocasiones el cliente debe esperar una respuesta antes de proseguir (por ejemplo, en login). Para otros casos, como registro, se sigue haciendo uso de esta cola, pero haciendo una espera asíncrona.
2. **Robots:** Esta cola funciona para enviar mensajes a los robots. En el estilo de Cola de Trabajo simplemente coloca los mensajes y el primero en estar disponible los recibe, permitiendo dividir el trabajo entre múltiples entidades. En modo fanout, todos los robots recibirán el mensaje, esto es útil para cancelaciones, y al ser una cola diferente, no hay problema de prioridades. Estas dos colas podrían

considerarse como “diferentes”, ya que usan distintos mecanismos cada uno.

3. **Robots - respuesta:** Cola exclusiva para actualizar del estado final de la tarea al controlador, ya que de otra manera el cliente tendría que esperar a que el robot termine su trabajo para actualizar estados y proseguir, o en peores casos, perder información en caso de que el controlador muera mientras se está procesando el pedido, el cual ya tiene un estado final y si el mensaje de respuesta falla su entrega, tendría que repetir todo.
4. **Delivery:** Esta cola funciona similar al de robots, para enviar mensajes a los repartidores. En el estilo de Cola de Trabajo simplemente coloca los mensajes y el primero en estar disponible los recibe, permitiendo dividir el trabajo entre múltiples entidades. En modo fanout, todos los repartidores recibirán el mensaje, esto es útil para cancelaciones, y al ser una cola diferente, no hay problema de prioridades. Estas dos colas podrían considerarse como “diferentes”, ya que usan distintos mecanismos cada uno.
5. **Delivery - respuesta:** Cola exclusiva para actualizar del estado final de la tarea al controlador, ya que de otra manera el cliente tendría que esperar a que el repartidor termine su trabajo para actualizar estados y proseguir, o en peores casos, perder información en caso de que el controlador muera mientras se está procesando el pedido, el cual ya tiene un estado final y si el mensaje de respuesta falla su entrega, tendría que repetir todo.

e. **Sintaxis y formato de los mensajes**

i. **Mensajes de petición (Request):** Tanto para clientes, el controlador, robots y delivery. Usan formato JSON, siguiendo esta sintaxis:

```
{
      “subject”: “strings definidos en Enum”,
      “body”: “{“diccionario” : “parseado a string”}”
}
```

En donde subject determina el tipo de petición a realizar y el body contiene todo lo necesario para procesarlo, por ejemplo, el contenido de una orden, de un usuario, o un ID.

ii. **Mensajes de respuesta (Response):** Implementado para enviar mensajes de confirmación o, por ejemplo, en el caso de los clientes, recibir los detalles de pedidos, datos del usuario, etc.

```
{
      “status”: “strings definidos en Enum ”,
      “message”: “Mensaje adicional, opcional”,
      “body”: “{“diccionario” : “parseado a string”}”
}
```

En donde “status” determina el si hubo éxito o no en la petición, message proporciona datos adicionales del estado y el body contiene información útil sobre la petición, por ejemplo, la lista de pedidos (si se pidieron) o el contenido de un pedido..

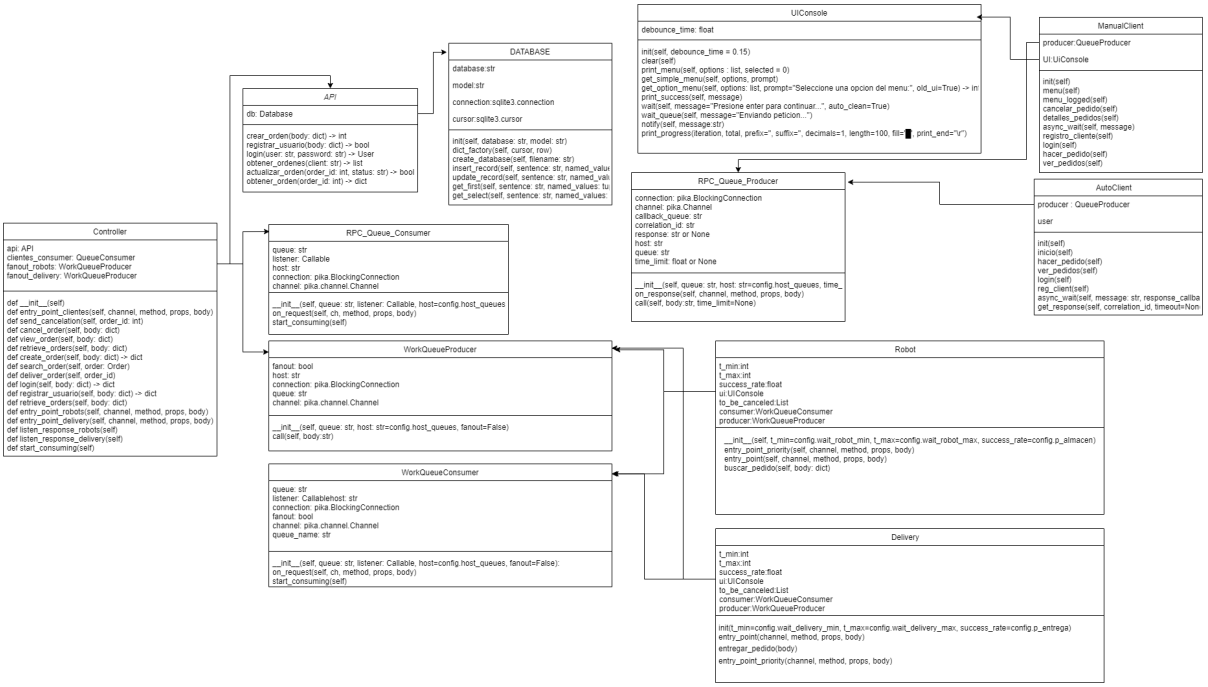
f. **Flujo lógico:** El flujo de la aplicación, a pesar de ser lineal, no es necesariamente ejecutado secuencialmente en una sola línea de tiempo

síncrona, es decir, puede que ciertas porciones se ejecuten de forma continua, pero en algunas haya que esperar un poco a ser ejecutadas, esto porque son ejecutadas de forma síncrona.

Dependiendo de la petición, más o menos actores son involucrados, pero en un caso general de un pedido, se sigue este flujo:

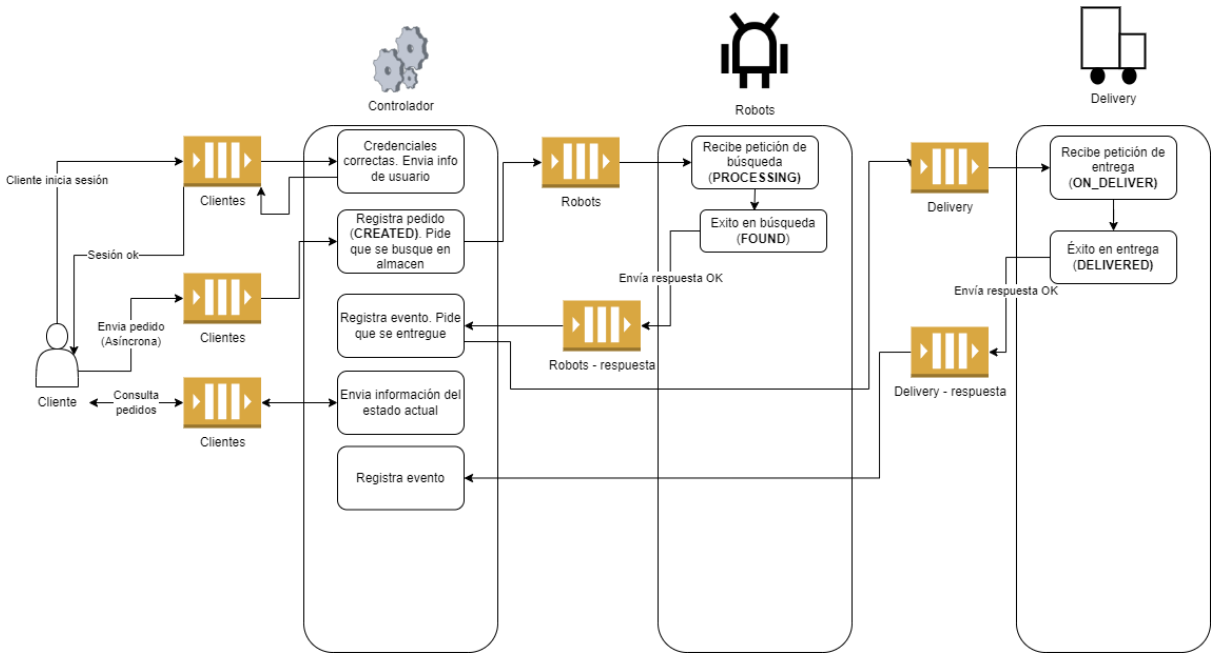
i. El cliente hace una petición de crear un pedido (puede ser registrarse, ver pedidos, etc.)
ii. El controlador recibe las peticiones del cliente y las procesa según sea el caso, si fuera login o registro este mismo solamente responde, pero si fuera un pedido, se actualiza la base de datos y se envía el pedido a búsqueda a los robots
iii. Los robots reciben la solicitud de búsqueda, la cual empiezan a procesar. Al momento de tener un resultado (éxito o fracaso), comunican este al controlador
iv. El controlador recibe dicha respuesta, y según sea el caso actualiza el estado y además comunica al repartidor, si fuera exitoso
v. El repartidor recibe el mensaje, lo proceso, y sea cual sea su respuesta, la comunica al controlador
vi. El controlador recibe dicha respuesta y actualiza en la base de datos.
vii. Cabe mencionar que en cualquier momento el usuario puede cancelar el pedido (otra cosa diferente es que este sea cancelado exitosamente) y además consultar los pedidos.

g. Diagrama de clases

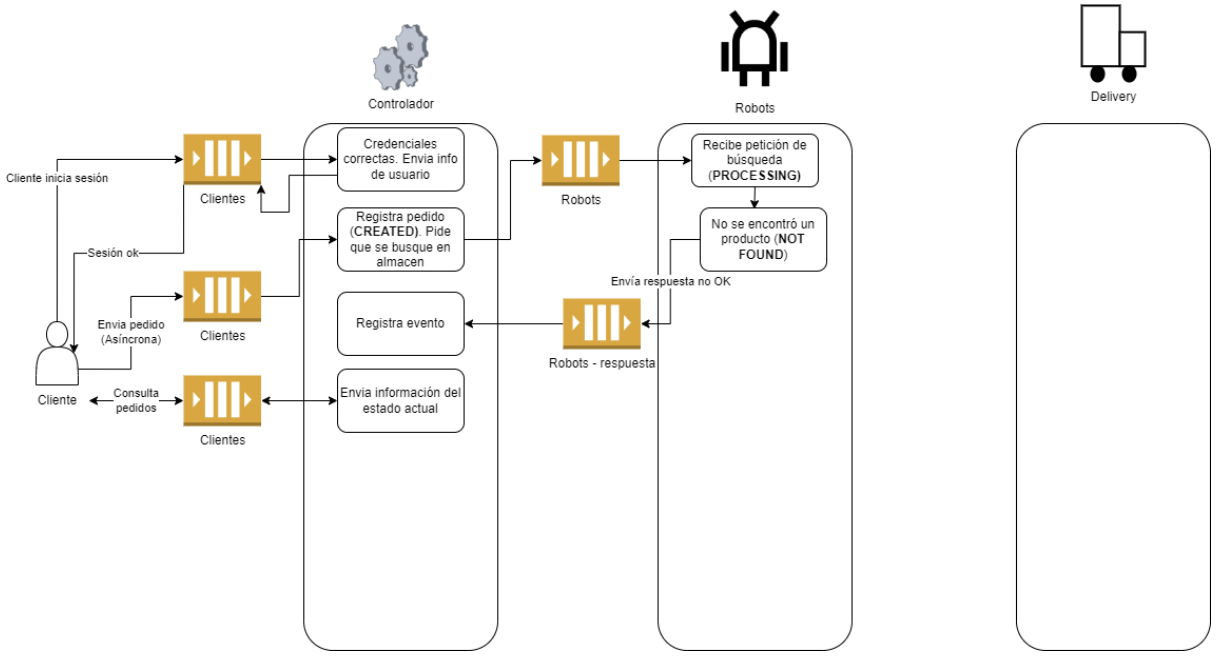


h. Casos de uso

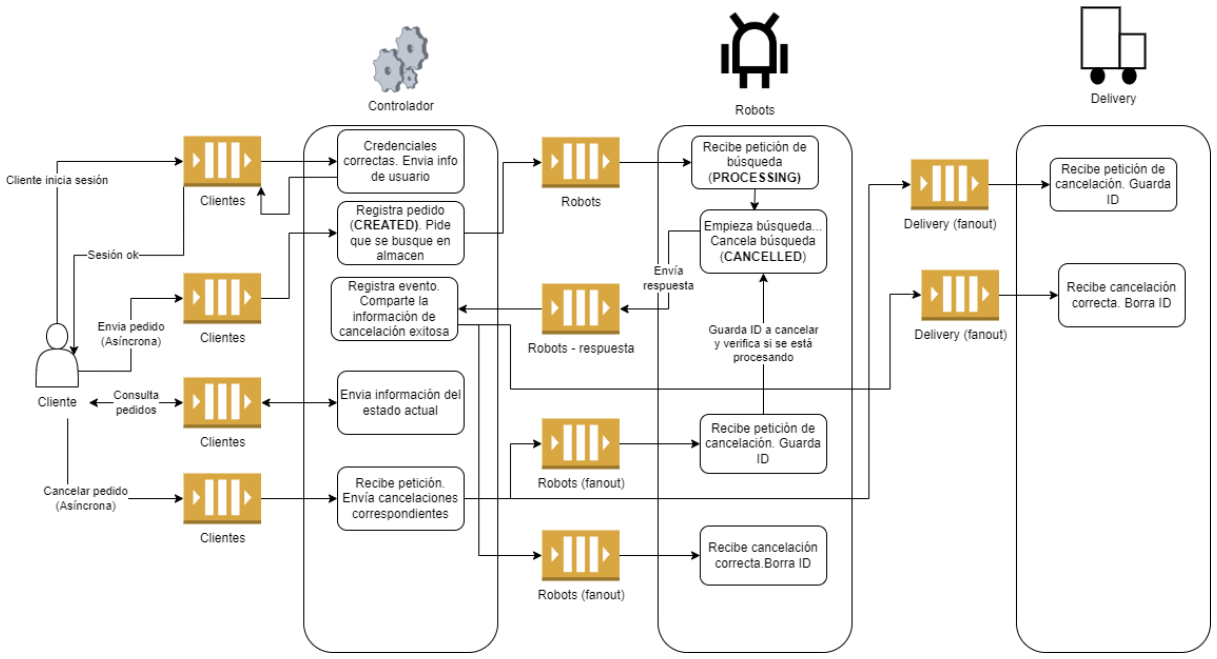
i. Pedido completado hasta el final



ii. Pedido en el que el robot no encuentra el producto



iii. Pedido que se cancela antes de empezar el reparto



3. DESARROLLO TÉCNICO E IMPLEMENTACIÓN

Durante el desarrollo del proyecto, se optó en primera instancia por utilizar el sistema de colas de mensajes para enviar y recibir mensajes básicos. Posteriormente, se desarrolló en paralelo la gestión de las bases de datos, el controlador y una interfaz de línea de comandos para el cliente. A medida que se avanzaba en el proyecto, se logró implementar un sistema funcional de registro de usuarios, inicio de sesión y generación de órdenes de compra, sin embargo, estas órdenes no eran procesadas adecuadamente.

Con el objetivo de solucionar esta situación, se procedió a desarrollar una primera versión de los programas de robots y delivery para que las órdenes fueran procesadas correctamente y entregadas a los clientes. A medida que se avanzaba en el proyecto, se identificaron diferentes matices y problemas, como el control de errores del sistema, la cancelación de pedidos y la elaboración más profunda del sistema de cliente manual.

Finalmente, se resolvieron estas dificultades hasta lograr un sistema de comercio electrónico completamente funcional. Posteriormente, se implementó el "cliente automático": launch_client. Con esto, se dio por concluido el desarrollo del proyecto.

Descripción de mensajes

Como se vió, los mensajes dentro del sistema siguen un formato único para peticiones y otro para respuestas, los cuales están en formato JSON y requieren de un parseo previo a envío y posterior a recepción.

Tal y como se explica, las peticiones tienen un subject o asunto, el cual puede ser cualquiera de los siguientes, y su body es el siguiente:

- **REGISTRO**: Mensaje de cliente a controlador para registro de un usuario
  - Body: Username, name, password
- **LOGIN**: Mensaje de cliente a controlador de inicio de sesión de un usuario
  - Body: Username, password
- **CREATE_ORDER**: Mensaje de cliente a controlador para crear una orden
  - Body: ID, description, total, client
- **SEARCH_ORDER**: Mensaje de controlador a robot para buscar clientes
  - Body: Orden
- **VIEW_ORDERS**: Mensaje de cliente a controlador para listar órdenes de un usuario
  - Body: Cliente
- **DELIVER_ORDER**: Mensaje de controlador a repartidor para entregar pedido
  - Body: Orden
- **CANCEL_ORDER**: Mensaje de cliente a controlador para cancelar una orden
  - Body: Orden
- **ORDER_FOUND**: Mensaje de robot a controlador para avisar que los artículos fueron encontrados
  - Body: Orden
- **ORDER_NOT_FOUND**: Mensaje de robot a controlador para avisar que los artículos no fueron encontrados
  - Body: Orden
- **ORDER_DELIVERED**: Mensaje de repartidor a controlador para avisar que el pedido fue entregado
  - Body: Orden
- **ORDER_LOST**: Mensaje de repartidor a controlador para avisar que el pedido no pudo ser entregado
  - Body: Orden
- **ORDER_CANCELED**: Mensaje de repartidor o robot al controlador para avisar que el pedido fue correctamente descartado
  - Body: Orden
- **VIEW_ORDER**: Mensaje de cliente a controlador para solicitar los detalles de un pedido
  - Body: Orden (ID)
- **CLEAR_CANCELATION**: Mensaje de cliente a robot y repartidor para avisar que un pedido fue correctamente cancelado y por lo tanto pueden dejar de tomarlo en cuenta
  - Body: Orden (ID)
- **ON_DELIVER**: Mensaje de repartidor a robot para indicar que ha comenzado el reparto
  - Body: Orden

Y, para las respuestas, en general el body contiene el contenido solicitado, el message es solo informativo (y puede estar vacío, en cuyo caso se recibe un string vacío), pero el estado puede ser:

- **ERROR**: La solicitud no pudo ser concretada con éxito, es decir, hubo un login incorrecto o no se pudo crear la orden, por ejemplo.
- **OK**: La solicitud fue realizada con éxito
- **MALFORMED**: La solicitud no pudo ser procesada por error del cliente, es decir, no está en formato adecuado.

4. CONCLUSIONES

a. Conclusiones técnicas

En primer lugar, se ha podido comprobar la utilidad y eficacia de la implementación de colas de mensajes en el desarrollo de aplicaciones distribuidas. El uso de este patrón de comunicación permitió separar la lógica de los diferentes componentes del sistema, lo que facilitó su desarrollo y su escalabilidad.

Por otra parte, se destaca la importancia de la implementación de un sistema de control de errores y excepciones en una aplicación compleja como esta. Gracias a la detección y manejo adecuado de estos errores, se logró reducir el impacto de posibles fallos en el sistema y mantener su estabilidad y fiabilidad.

Otro punto a destacar es la implementación de la funcionalidad de cancelación de pedidos, el cual hizo que se tuviese que pensar el proyecto de forma holística, lo cual ayudó a entender mejor cómo funcionan los sistemas basados en microservicios.

b. Conclusiones personales

En primer lugar, es importante destacar el valor de trabajar en equipo y la importancia de una buena planificación y organización. A lo largo del desarrollo de Saimazoom, se encontraron diversos desafíos técnicos que se pudieron superar gracias al trabajo en equipo y al esfuerzo conjunto.

Por otro lado, el uso de herramientas y tecnologías actuales, como el sistema de colas de mensajes, permitió desarrollar un proyecto moderno y con un alto nivel de escalabilidad.

Asimismo, la implementación de buenas prácticas de programación, como el uso de patrones de diseño y la gestión adecuada de las bases de datos, contribuyeron a la calidad del proyecto y facilitaron su mantenimiento.

Finalmente, el proyecto Saimazoom fue una experiencia enriquecedora que permitió poner en práctica los conocimientos adquiridos a lo largo del curso y desarrollar habilidades técnicas y de trabajo en equipo que serán útiles en el futuro.