# UNIVERSIDAD AUTÓNOMA DE MADRID

## ESCUELA POLITÉCNICA SUPERIOR



**Grado en Ingeniería Informática**

# MEMORIA DE LA PRÁCTICA 2

**REDES DE COMUNICACIONES II**

**Javier Valero Velázquez**
**Sergio Larriba Moreno**

# ÍNDICE

# 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 repartidores. La interacción entre estos componentes se realiza mediante colas de mensajes RabbitMQ.

El objetivo de esta aplicación es proporcionar un sistema software de pedidos donde los usuarios pueden enviar solicitudes de entrega y hacer un seguimiento de sus pedidos. Para lograr este objetivo, se ha desarrollado una arquitectura basada en colas de mensajes.

Gracias a ellas, es posible 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 documento, se describe en detalle la arquitectura de la aplicación, incluyendo su diseño y el porqué, las tecnologías utilizadas y diferentes diagramas de explicación del software.

# 2 - Diagramas

## 2.1 - Diagrama de Clases

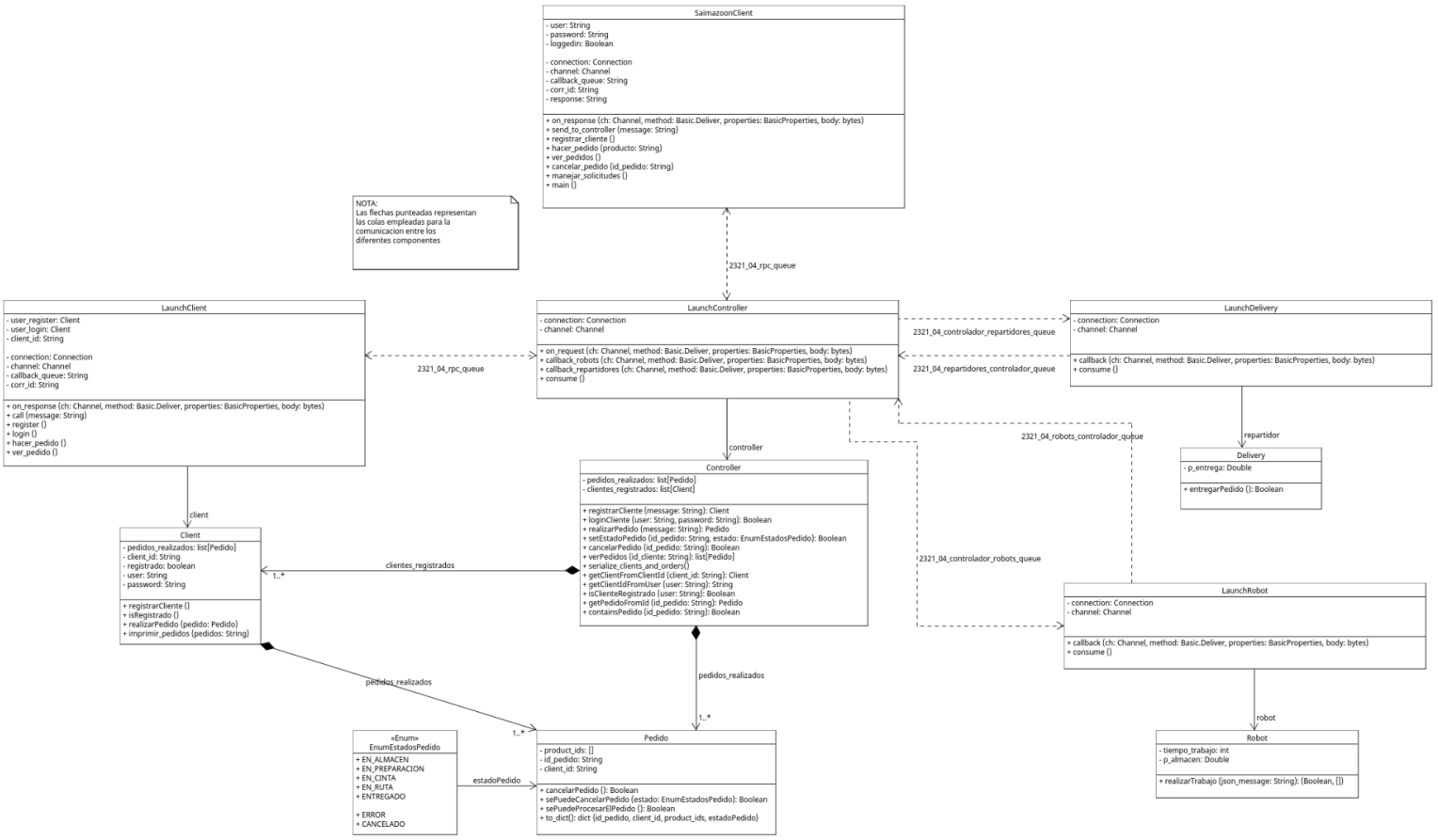
En el diagrama de clases se representan los diferentes componentes de la aplicación. Se pueden distinguir 2 tipos de clases, aquellas relacionadas con el lanzamiento de los componentes (empiezan por el prefijo “Launch”) y aquellas relacionadas con la gestión de la funcionalidad interna.

Cada clase “Launch” tiene asociada su clase de gestión interna. Las clases de tipo “Launch” sirven para toda la inicialización, envío y recepción de mensajes a través de las colas de mensajes, mientras que las otras clases sirven para funcionalidades más específicas.

Por ejemplo, la clase “LaunchClient” tiene toda la gestión del envío de mensajes por parte del cliente al controlador y la recepción de las respuestas de éste, sin embargo, dicha clase como tal no realiza ninguna funcionalidad interna (registro de un cliente por ejemplo), para ello se emplea la clase “Client”.

De esta forma conseguimos modularizar mucho más el código y separar las funcionalidades con el objetivo de hacer un código mucho más legible, modificable y entendible para cualquier nuevo integrante que desee formar parte del proyecto.

A continuación se adjunta el diagrama de clases realizado:

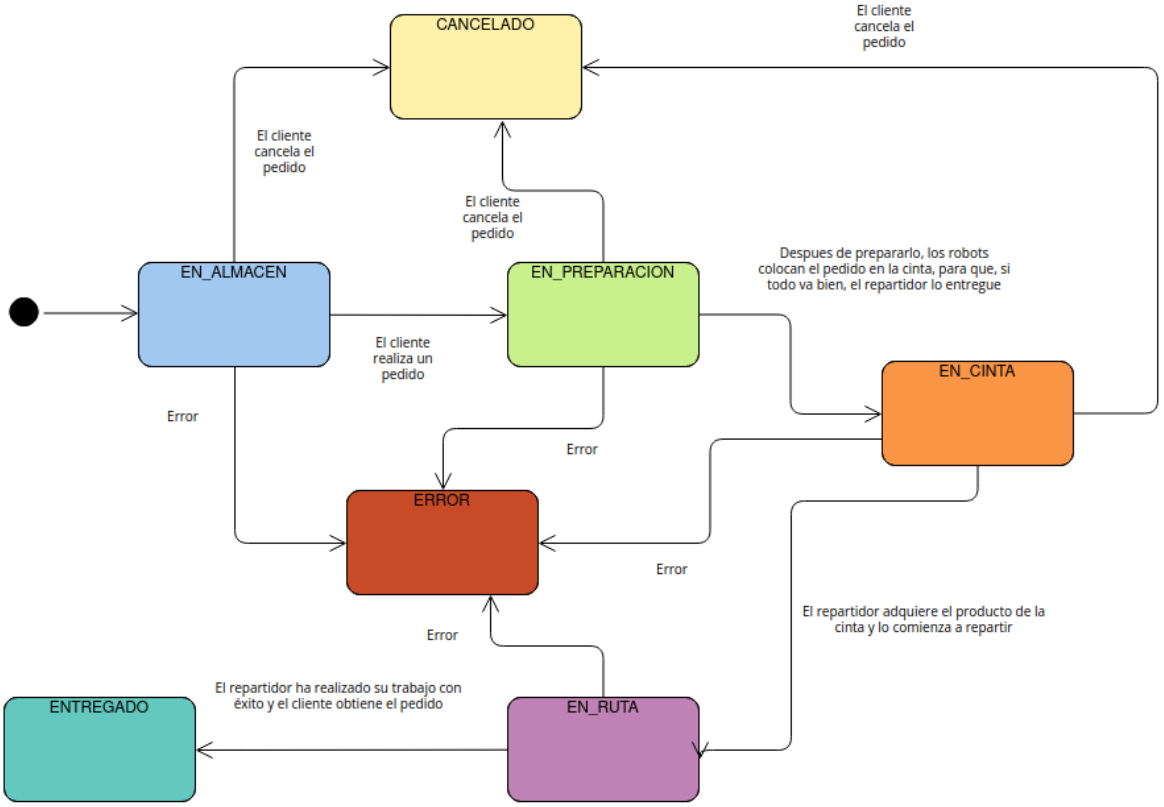



Si se desea ver el diagrama de clases mejor, se adjunta un archivo .png en la ruta desde la raíz del proyecto gitlab: Diseño/Diagrama_De_Clases/Diagrama_De_Clases.png

Como se puede observar, las colas, representadas por flechas discontinuas, conectan las diferentes clases de tipo “Launch”, y la funcionalidad como tal de la aplicación la realizan el resto de clases, cada una siendo accesible desde su respectiva clase “Launch”.

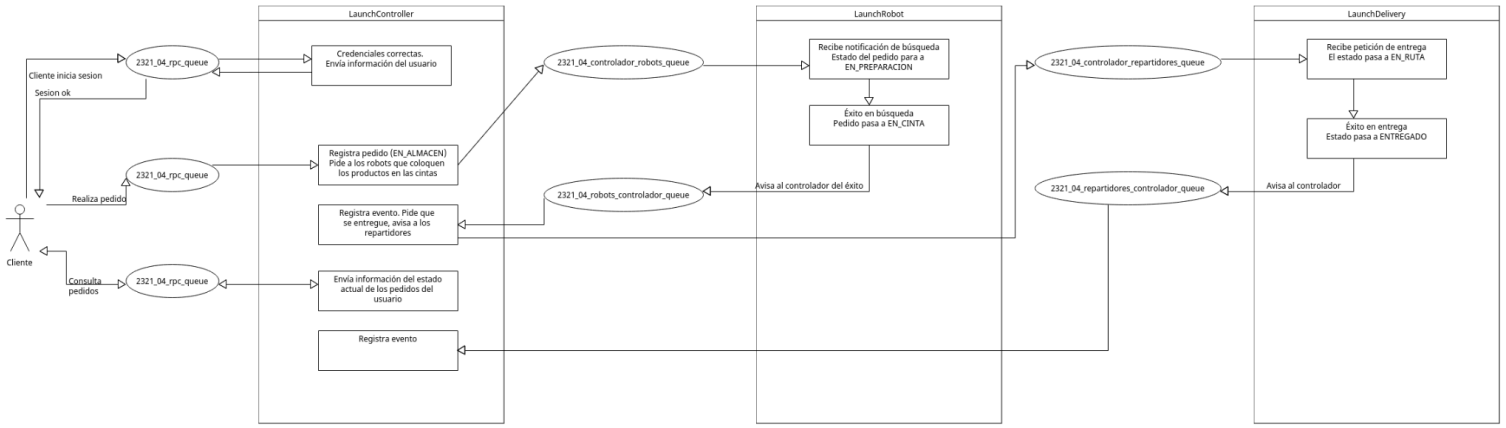
La clase “SaimazoonClient” corresponde al archivo commandline_client.py, que simula la interfaz que recibiría un cliente por la línea de comandos. Dicha clase también es de tipo “Launch” y la clase que gestiona su funcionamiento interno es “Client”, la misma para la clase “LaunchClient”.

## 2.2 - Diagrama de Estados de un Pedido

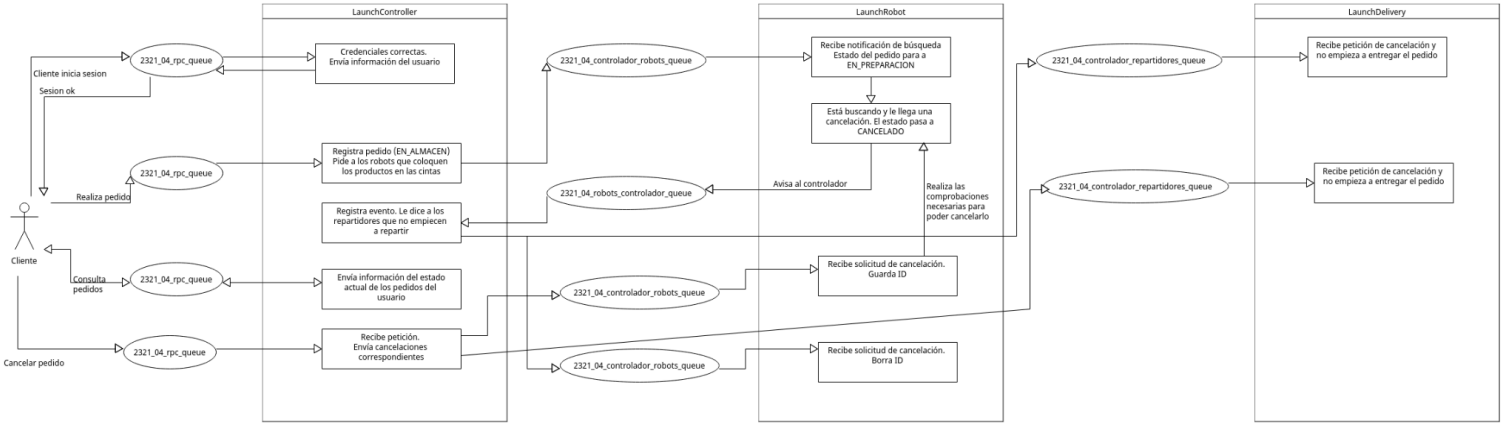



## 2.3 - Diagramas de Casos de Uso

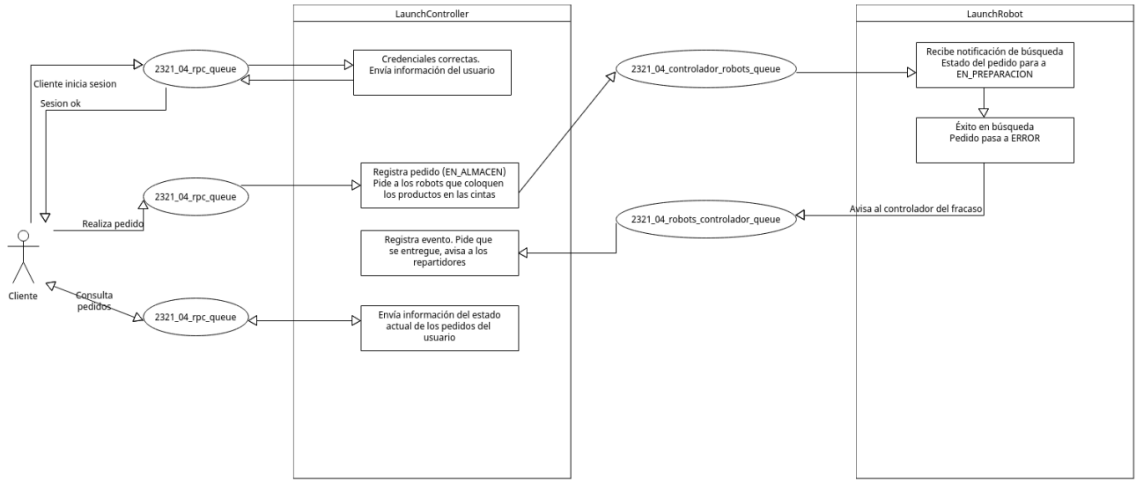
### 2.3.1 - Pedido completado hasta el final



## 2.3.2 - Pedido que se cancela antes de empezar el reparto

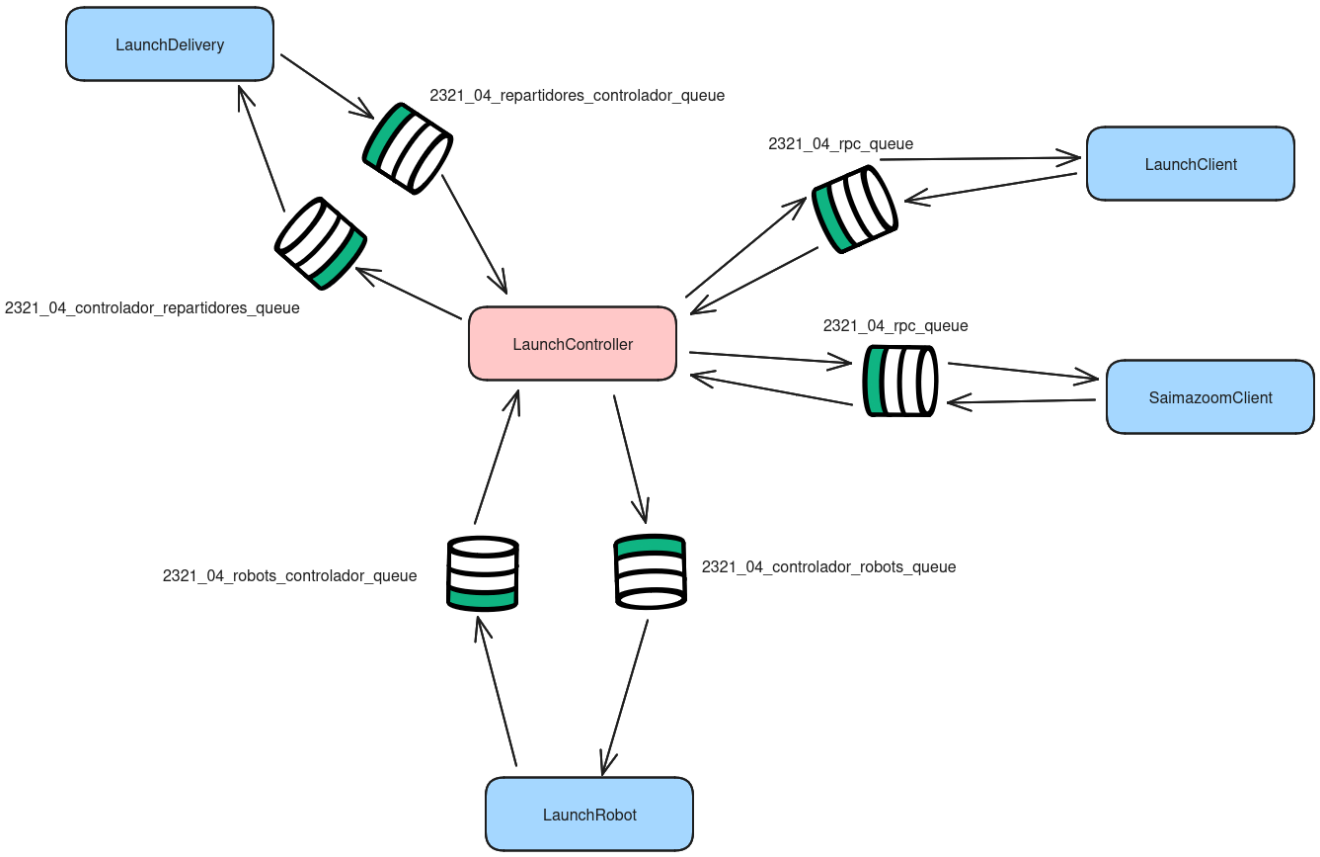



## 2.3.3 - Pedido en el que el robot no encuentra el producto



# 3 - Diseño y uso de las colas de mensajes

## 3.1 - Diagrama de Colas de Mensajes




Hemos usado tanto colas RPC (solicitud - respuesta) como colas de trabajo (distribuir tareas entre diferentes trabajadores).

- **2321_04_rpc_queue:** Esta cola funciona para que el cliente pueda generar mensajes, productor, que el controlador procesa, consumidor. Es de tipo RPC ya que en algunas ocasiones el cliente debe esperar una respuesta antes de continuar con su ejecución (por ejemplo al hacer login, hasta que no se loguea no puede hacer pedidos). Para otros casos, como el registro, se sigue haciendo uso de esta cola pero de manera asíncrona. Además, el objetivo principal que nos ha llevado a usar una RPC es para que el controlador sepa a qué cliente le está mandando la respuesta a su solicitud correspondiente.
- **2321_04_controlador_robots_queue:** Esta cola sirve para enviar mensajes desde el controlador a los robots. Es una cola de trabajo (work queue), es decir, simplemente coloca los mensajes y el primer robot en estar disponible los recoge, permitiendo así dividir el trabajo entre múltiples entidades.
- **2321_04_robots_controlador_queue:** Esta cola sirve exclusivamente para que los robots informen al controlador sobre el éxito o fracaso en la puesta de los productos solicitados en la cinta, al igual que la cola de ida, también es una cola de trabajo.

- **2321_04_controlador_repartidores_queue:** Esta cola tiene un funcionamiento muy similar a la cola “2321_04_controlador_robots_queue”, y al igual que ella, también es una cola de trabajo, pero en vez de ser robots, esta vez son repartidores.
- **2321_04_repartidores_controlador_queue:** Esta cola tiene un funcionamiento idéntico a la cola “2321_04_robots_controlador_queue” pero orientado a los repartidores. Simplemente sirve para avisar por parte de los repartidores al controlador del éxito o fracaso en la entrega.

# 4 - Descripción de los mensajes: sintaxis y formato

---

Todos los mensajes intercambiados entre las diferentes colas de mensajes tienen formato JSON. Es un formato ligero y fácilmente interpretable tanto por humanos como por máquinas, lo que agiliza el procesamiento y la manipulación de la información.

**Mensajes desde el cliente al controlador**

- **Registro:**
  message = {"accion": "REGISTER",
  "user": "String con el nombre de usuario",
  "password": "String con la contraseña"
  }

- **Login:**
  message = {"accion": "LOGIN",
  "user": "String con el nombre de usuario",
  "password": "String con la contraseña"
  }

- **Hacer pedido:**
  message = {"accion": "ORDER",
  “product_ids”: [Array con los productos que pide el cliente en el pedido],
  “user”: “String con el nombre de usuario”
  }

- **Ver pedido:**
  message = {"accion": "SEE",
  “user”: “String con el nombre de usuario”
  }

- **Cancelar pedido:**
  message = {"accion": "CANCEL",
  “id_pedido”: “String con el identificador del pedido a cancelar”,
  “user”: “String con el nombre de usuario”
  }

**Mensajes desde el controlador al cliente (Respuesta a las solicitudes del cliente)**

- **Confirmación exitosa del registro:**
  message = {"accion": "REGISTERED",
  “client_id”: “String con el id del cliente registrado”,
  "user": "String con el nombre de usuario",
  "password": "String con la contraseña"
  }

- **Fallo en el registro:**
  message = {"accion": "NOT-REGISTERED",
  "user": "String con el nombre de usuario",
  "password": "String con la contraseña",
  “causa”: “El cliente ya existe”
  }

- **Confirmación exitosa del login:**
  message = {"accion": "LOGGED",
  “client_id”: “String con el id del cliente registrado”,
  "user": "String con el nombre de usuario",
  "password": "String con la contraseña"
  }

- **Fallo en el login:**
  message = {"accion": "NOT-LOGGED",
  “client_id”: “String con el id del cliente registrado”,
  "user": "String con el nombre de usuario",
  "password": "String con la contraseña",
  “causa”: “Usuario o contraseña incorrectos”
  }

- **Confirmación exitosa del pedido:**
  message = {"accion": "ORDERED",
  “id_pedido”: “String con el identificador del pedido solicitado”,
  “product_ids”: Array con los productos que contiene el pedido,
  “client_id”: “String con el id del cliente registrado”
  }

- **Confirmación exitosa de la cancelación de un pedido:**
  message = {"accion": "CANCELLED",
  “id_pedido”: “String con el identificador del pedido solicitado”,
  “client_id”: “String con el id del cliente registrado”
  }

- **Confirmación no exitosa de la cancelación de un pedido (el pedido está ya entregado o en reparto):**
  message = {"accion": "NOT-CANCELLED",
  “id_pedido”: “String con el identificador del pedido solicitado”,
  “client_id”: “String con el id del cliente registrado”,
  “causa”: “El pedido no existe o ya está en reparto”
  }

- **Confirmación no exitosa de la cancelación de un pedido (el cliente no está registrado):**
  message = {"accion": "NOT-CANCELLED",
  “id_pedido”: “String con el identificador del pedido solicitado”,
  “client_id”: “String con el id del cliente registrado”,
  “causa”: “El cliente no está registrado”
  }

- **Confirmación exitosa de ver pedidos:**
  message = {"accion": "SEEN",
  “client_id”: “String con el id del cliente registrado”,
  “pedidos”: “String con todos los pedidos del cliente”
  }

- **Confirmación no exitosa de ver pedidos:**
  message = {"accion": "NOT-SEEN",
  “client_id”: “String con el id del cliente registrado”,
  “causa”: “El cliente no está registrado”
  }

**Mensajes desde el controlador a los robots**

- **Puesta a los robots en marcha:**
  message = {"accion": "MOVE",
  “id_pedido”: “String con el identificador del pedido solicitado”,
  “product_ids”: Array con los productos que contiene el pedido,
  “client_id”: “String con el id del cliente que ha hecho el pedido”,
  “estado”: “String con el estado de EnumEstadosPedido del pedido”
  }

**Mensajes desde el controlador a los repartidores**

- **Puesta a los repartidores en marcha:**
  message = {"accion": "DELIVERY",
  “id_pedido”: “String con el identificador del pedido solicitado”,
  “product_ids”: Array con los productos que contiene el pedido,
  “client_id”: “String con el id del cliente que ha hecho el pedido”,
  “estado”: “String con el estado de EnumEstadosPedido del pedido”
  }

**Mensajes desde el repartidor al controlador**

- **Confirmación del pedido entregado con éxito:**
  message = {"accion": "DELIVERED",
  “id_pedido”: “String con el identificador del pedido solicitado”,
  “product_ids”: Array con los productos que contiene el pedido,
  “client_id”: “String con el id del cliente que ha hecho el pedido”,
  }

- **Confirmación de que el pedido no ha sido entregado con éxito:**
  message = {"accion": "NOT-DELIVERED",
  “id_pedido”: “String con el identificador del pedido solicitado”,
  “product_ids”: Array con los productos que contiene el pedido,
  “client_id”: “String con el id del cliente que ha hecho el pedido”,
  }

**Mensajes desde el robot al controlador**

- **Confirmación del pedido movido con éxito:**
  message = {"accion": "MOVED",
  “id_pedido”: “String con el identificador del pedido solicitado”,
  “product_ids”: Array con los productos que contiene el pedido,
  “client_id”: “String con el id del cliente que ha hecho el pedido”,
  }

- **Confirmación de que el pedido no ha sido movido con éxito:**
  message = {"accion": "NOT-MOVED",
  “id_pedido”: “String con el identificador del pedido solicitado”,
  “product_ids”: Array con los productos que contiene el pedido,
  “client_id”: “String con el id del cliente que ha hecho el pedido”,
  }

# 5 - Conclusiones

## 5.1 - 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.

## 5.2 - 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.