2020

Sistemas Distribuidos

Transacciones

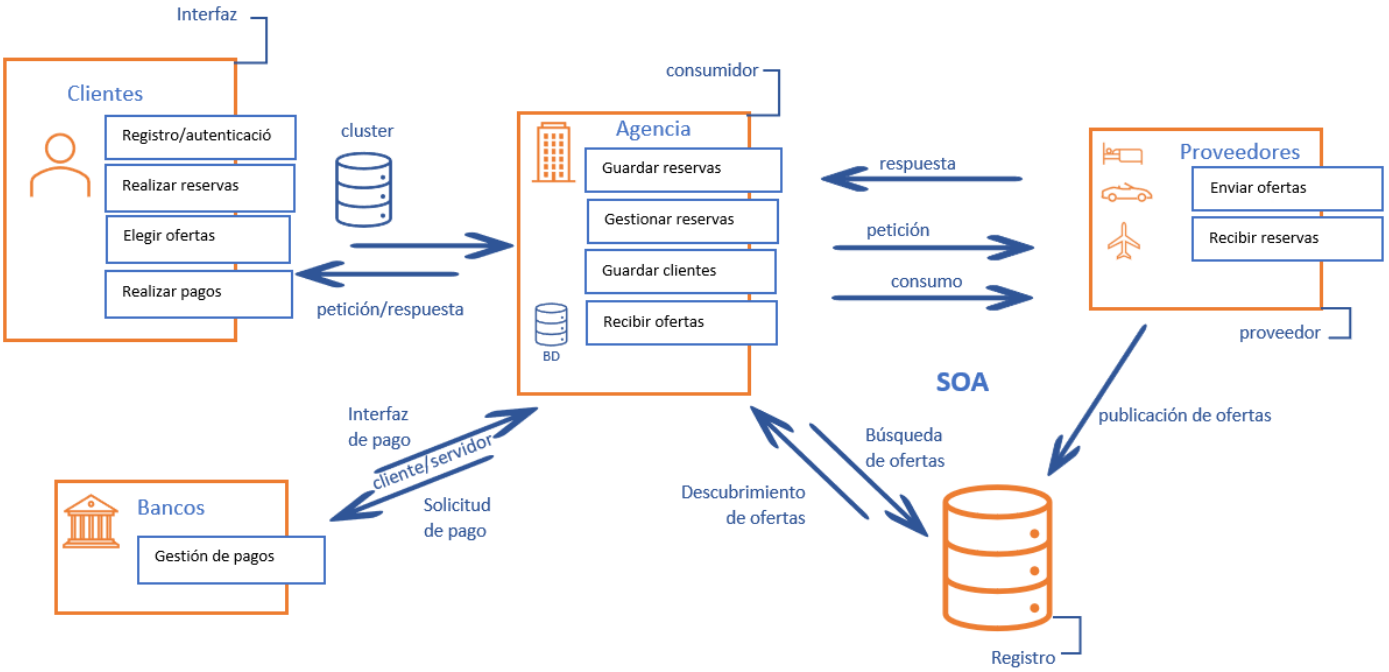
Francisco Joaquín Murcia Gómez
48734281H
Universidad de Alicante

Índice

Introducción

Arquitectura conceptual

He diseñado un esquema donde se puede observar de manera sencilla los actores, su función y como se comunica con el resto, no obstante, se indagará más en los siguientes apartados.



Actores

Encontraremos cuatro actores:

- **Clientes**
  Los clientes podrán registrarse, realizar ofertas y lo que conlleva consigo ver los productos cancelarlas…, elegir ofertas y realizar pagos
- **Agencia**
  Es la encargada de gestionar y guardar las reservas que hagan los clientes en su base de datos, en esta base de datos guardarán los clientes que se vayan registrando, también recibirán ofertes de los proveedores
- **Proveedores**
  Los proveedores se encargan de recibir las reservas que realizan los clientes que son gestionadas por la reserva y de enviara a esta las ofertas
- **Bancos**
  Los bancos son los encargados de gestionar los pagos

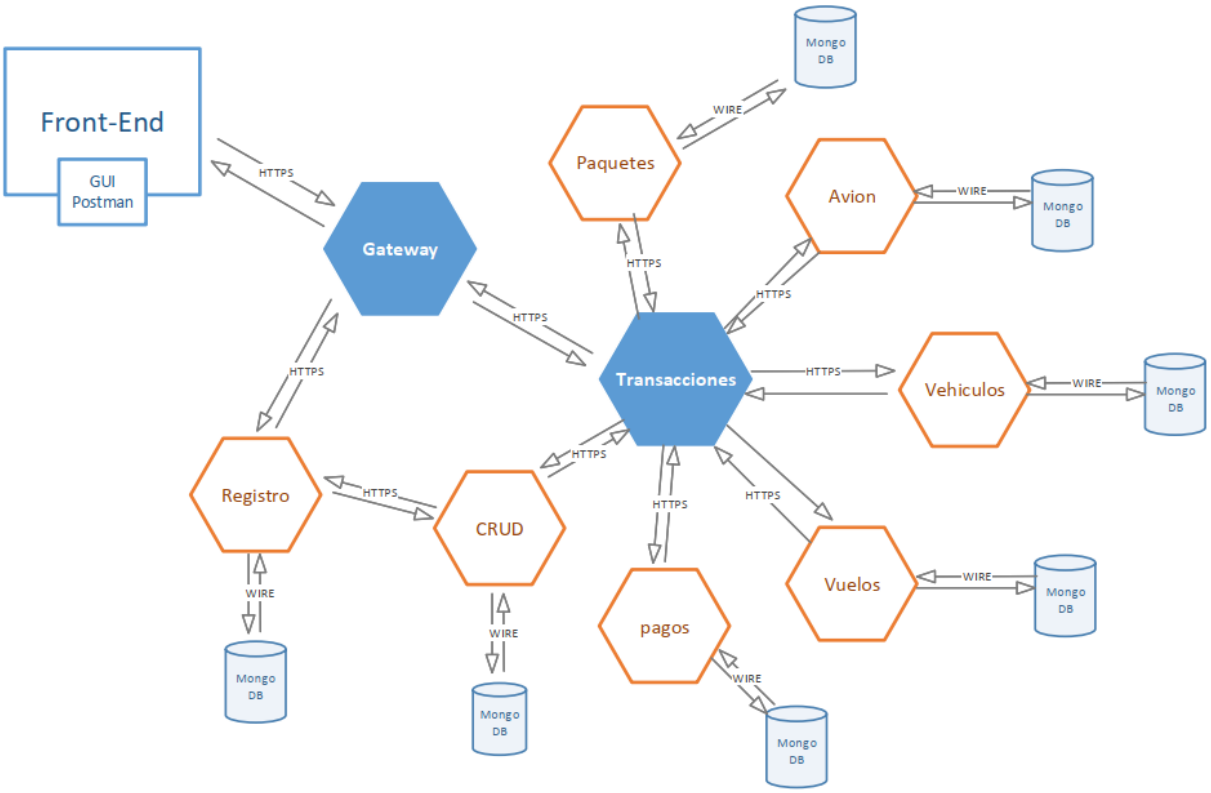
Arquitecturas empleadas

Para explicar el funcionamiento de la agencia de viajes hemos de dividir el esquema en tres partes:

- **Clientes-Agencia**
  Para la conexión de los clientes con el sistema de agencia estos se conectarán por media de cliente/servidor ya que clientes sería el front-end y la agencia sería el back-end, a aparte hay un fuerte acoplamiento debido a que la tecnología es la misma y es la misma empresa, a pate es la arquitectura más utilizada en entornos web.
- **Agencia-Proveedores**
  Para este caso he usado una arquitectura tipo SOA debido a que cumple todos los requisitos de este, necesitamos interoperabilidad, tenemos un contrato que facilita el desacoplamiento y la independencia con ellos, se va a hacer una composición de varios servicios (packs), necesitamos localizar como acceder a esos proveedores y es síncrono. También con SOA proporcionamos libertad al proveedor que dependiendo de su contrato ya sea REST o SOAP se hará un servicio punto a punto con su contrato.
- **Agencia-Bancos**
  Para la agencia con los bancos se utilizará cliente/servidor debido a que normalmente los bancos son los que nos indican como conectaros a ellos, y normalmente con un protocolo propiedad del banco que suele estar basado en sockets, por eso he decidido emplear C/S con sockets en el que el banco es el servidor y la agencia el cliente.

Arquitectura conceptual-técnica

Teniendo en cuenta los anteriores apartados obtenemos el siguiente esquema de la aplicación que se puede ver las diferentes apis (hexágonos) y como interactúan entre ellos.



Sincronización y Coordinación

En la agencia, necesitamos sincronizar diversos web services como los proveedores, tanto vuelos vehículos como hoteles; por lo tanto, pueden surgir problemas de sincronización, debido a que hay que evitar que varios clientes reserven un servicio a la vez usaremos cerrojos para bloquear estos servicios cuando un usuario este reservando un servicio y así evitar inconsistencias, MongoDB permite que se pueden implementar transacciones de manera sencilla con un nivel de bloqueo global o de colección, y con extensiones hasta de objetos gracias a sus últimas versiones como así indica su web oficial.

También usaríamos FIFO (first in, first out) así atenderemos primero a la primera petición del cliente que nos llegue, cada petición tendrán registros de tiempo para determinar el orden, como es evidente seria innecesario el uso de protocolos de tiempo (NTP,STP o PTP) ya que con ajustarnos a las milésimas nos sobraría

Transacciones Distribuidas

Para empezar, necesitamos recordar la diferencia principal entre transacciones distribuidas y concurrentes, siendo la primera que puede realizar varias operaciones al mismo tiempo y concurrente que mientras se realiza una operación, el servicio que se esta operando no se puede manipular.

Transacciones concurrentes

Encontramos transacciones concurrentes en la ya mencionada en el apartado anterior, la comunicación de Cliente-Agencia, mencionamos que hay que evitar que dos clientes interactúen con la misma reserva a la vez, para eso haremos uso de cerrojos que seria una solución sencilla y fácil de implementar, este cerrojo se cerrara cuando un usuario interactúe y se abrirá cuando deje de interactuar o surja algún error, de esta manera tomaremos el control en la sección critica.

Transacciones distribuidas

Por otro lado, tenemos las transacciones distribuidas, en nuestra aplicación la parte de reservar servicios serian transacciones distribuidas porque implican mínimo dos WS el de pagos y un proveedor tendremos un gestor de transacciones como se puede observar en el diagrama, en este están conectadas todos los servicios que requieren de esta sincronización, este gestor hará uso del protocolo 2PC de esta manera varios servicios podrán interactuar entre sí a la hora de realizar reservas.

2PC (2-phase commit)

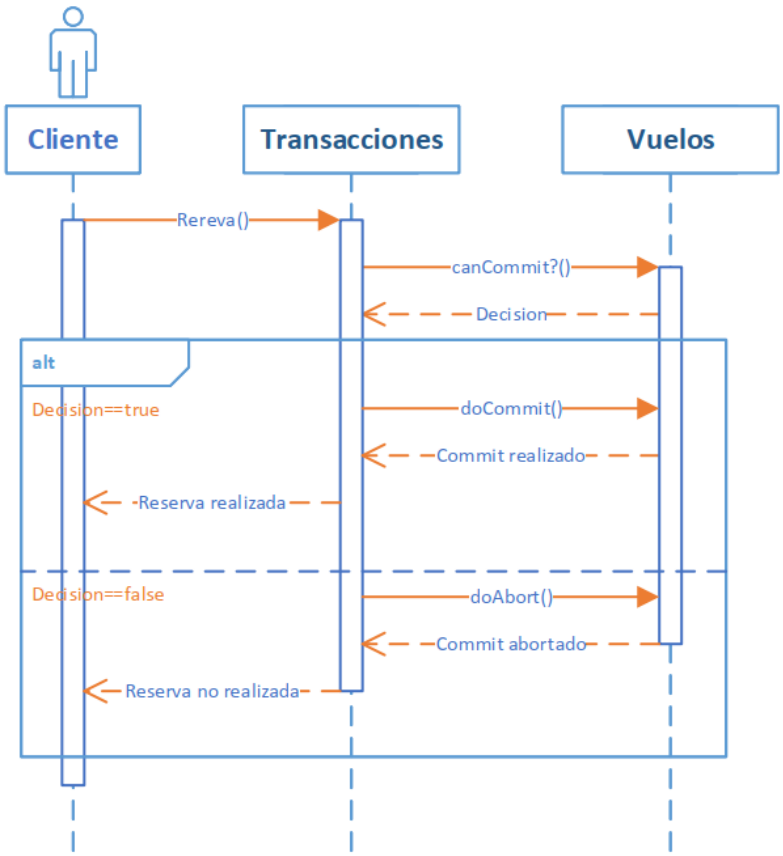
El funcionamiento de este protocolo es simple, tenemos dos partes la de votación y la de compromiso:

1. **Votación**
   En la fase de votación se pregunta a todos los servicios implicados en una acción si esta se puede realizar, los servicios envían su respuesta.
2. **Compromiso**
   En esta parte se analiza las respuestas de los servicios, es imperativo que todos los servicios estén de acuerdo con la realizar la acción ya que, si solo uno dice que no la acción se abortara, si todos están de acuerdo se realizara.

Este algoritmo hace uso de varias funciones las más importantes y que forman parte del grueso del algoritmo son:

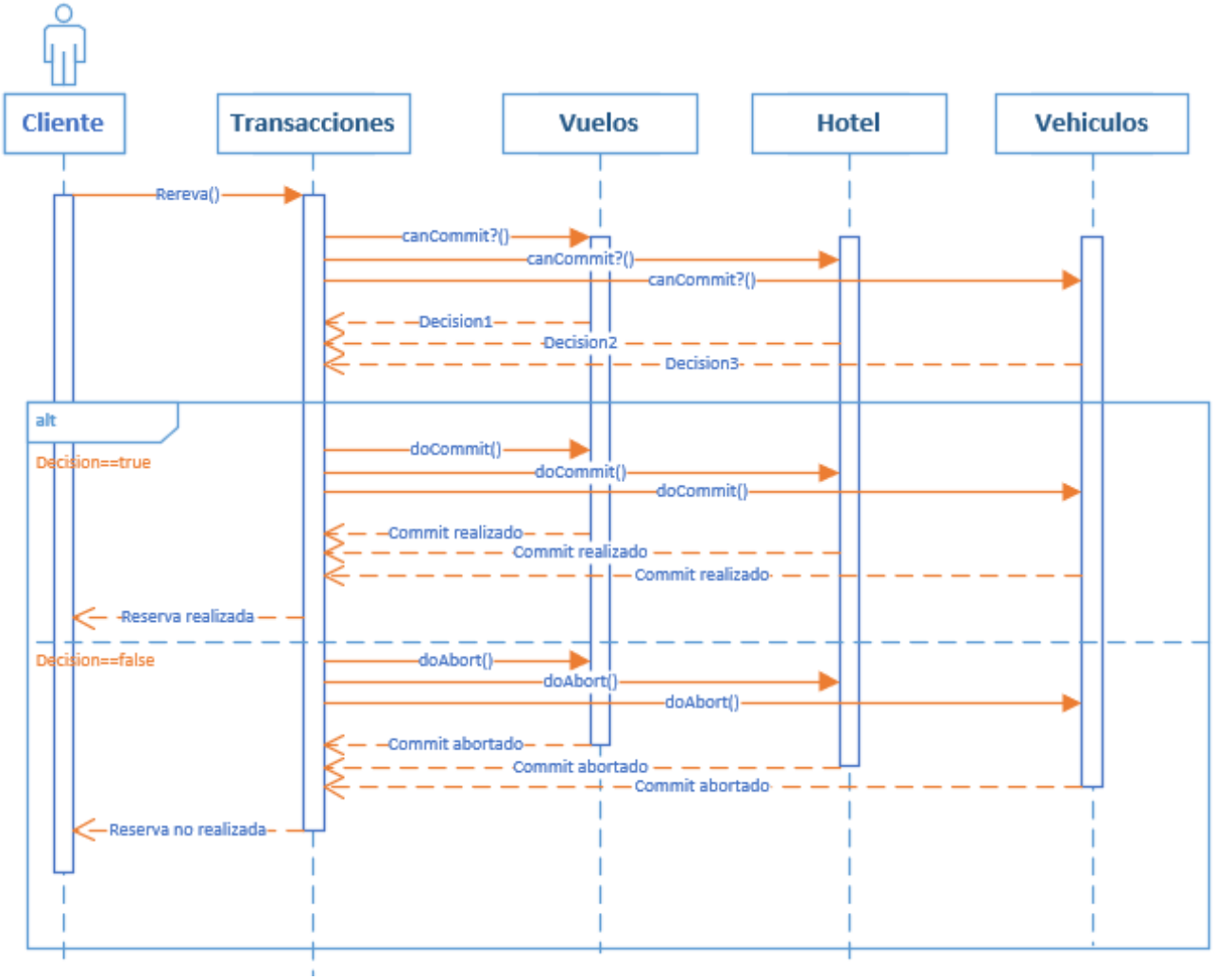
- **canCommit?():** pregunta si se puede realizar la acción.
- **doCommit():** realiza la acción.
- **doAbort():** aborta la acción.

Una forma visual de ver el funcionamiento de este es empleando un diagrama de secuencia, en este caso es cuando interactúa solo un servicio, en este caso un cliente esta reservando un vuelo.



Como se puede ver en el caso de que la decisión sea afirmativa se realizara la reserva y se almacenará en la base de datos mediante el CRUD, en caso contrario se abortara y no se realizara la reserva, si se produce algún error como "tarjeta invalida" se realizara un "rollback" descartando los cambios.

Si un cliente quisiera reservar un pack que envuelve a los tres proveedores seria de la misma manera, solo habrá que tener en cuenta los servicios implicados.



Si la decisión global es afirmativa se procederá a la realización de la reserva, sin embargo, si alguno no puede realizarla la decisión global será negativa entonces se abortará la acción en todos los servicios.

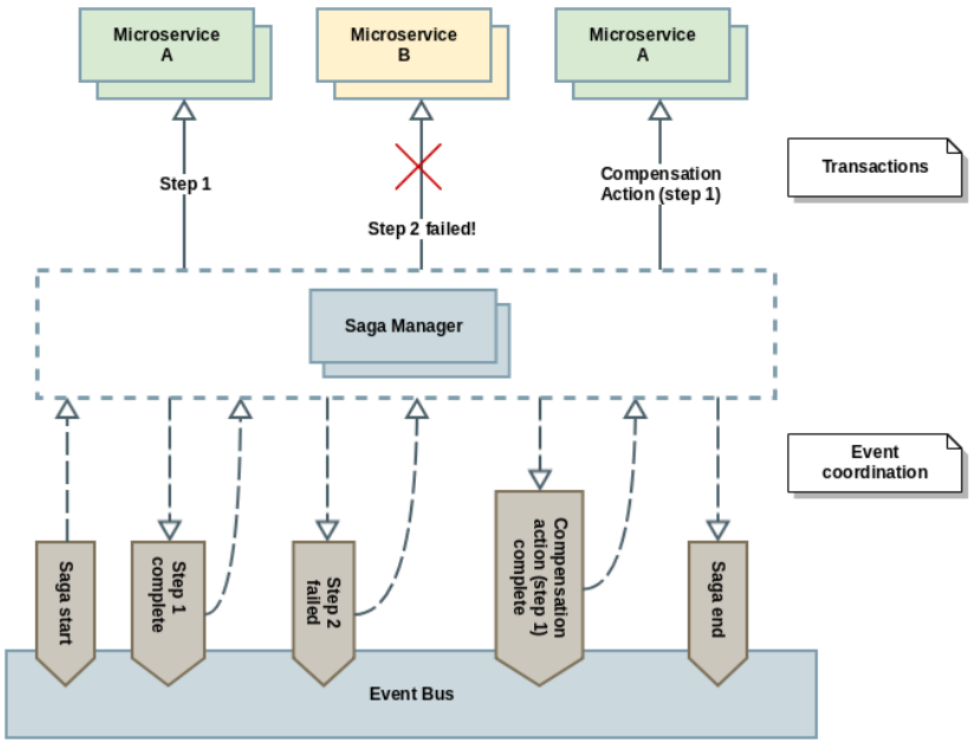
Este algoritmo posee una buena tolerancia a fallos ya que se puede recuperar de caídas o imprevistos como hemos observado. No obstante, no aporta mucha consistencia de datos

2PC con patrón SAGA

El patrón SAGA es aporta mas consistencia de datos ya que tratamos con transacciones distribuidas ya que SAGA cuando hace una transacción también hace una acción para deshacela en caso de que esta falle.
Existen dos tipos de SAGA:

- **Mediante Coreografía**
  Cuando no hay una coordinación central, cada servicio produce y escucha los eventos de otros servicios y decide si se debe tomar una acción o no.
- **Mediante Orquestación**
  Cuando un servicio de coordinador es responsable de centralizar la toma de decisiones del SAGA

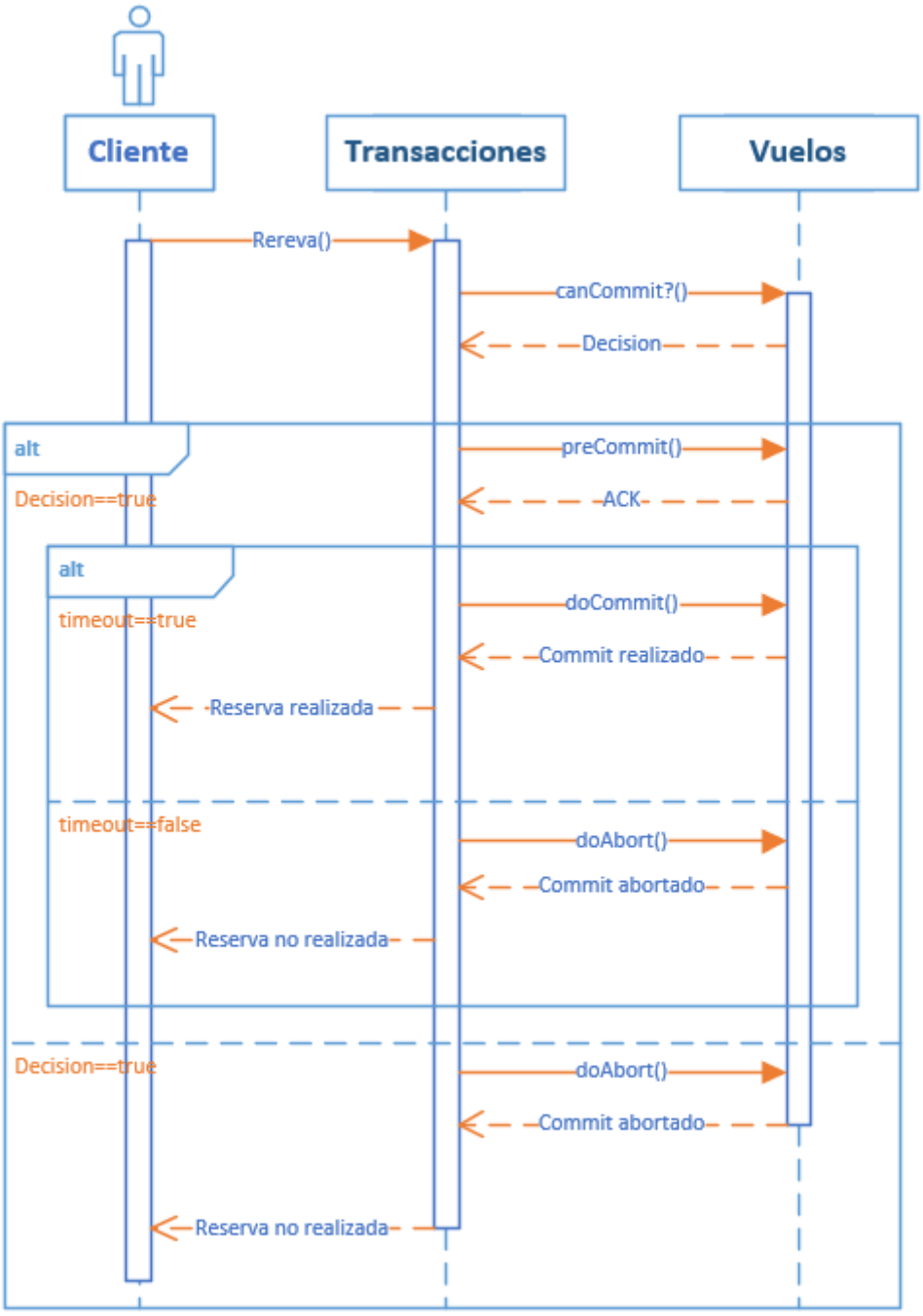
Para nuestra aplicación lo idóneo seria usar orquestación ya que tenemos un WS de transacciones que es el que organiza las transacciones, a parte este evita las redundancias cíclicas; el funcionario de 2PC con SAGA mediante orquestación seria simple:



Sin embargo, los algoritmos basados en 2pc no son perfectos ya que es posible que algún elemento se quede bloqueado, es este caso el protocolo estría esperando eternamente, para solucionar esto existe una variante de este llamada 3PC.

3PC (3-phase commit)

3PC aparte de seguir todos los procesos que 2PC, este también usa "timeouts" y un mensaje intermedio si todos aceptan llamado "preCommit()", al enviar esto, se espera una respuesta por parte de los servicios y envase al tiempo de respuesta se continua o no, el proceso seria el siguiente:



Si cunado enviar preCommit() hay mucho tiempo de espera se aborta el proceso y si esta por debajo del máximo se continuara con el proceso y se realizara la reserva en este caso, de esta forma si por ejemplo en este caso vuelos esta bloqueado no se colgará el proceso.

Conclusiones

Teniendo en cuenta lo ya explicado, he decidido implementar en la práctica los siguientes recursos:

La utilización de registros de tiempo para atender las peticiones de los clientes por medio de FIFO y usar cerrojos para evitar incongruencias de datos

Y en la parte distribuida me he decantado por 2PC a parte que es el que más recursos hay en la web para implementarlo con nuestra tecnología (node.js), sería suficiente para el tipo de aplicación que estamos realizando ya que evitaremos sobredimensionar la solución para las transacciones distribuidas entre los diferentes servicios