POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Arquitecturas orientadas a la escalabilidad funcional

Autor: Jesús Martínez-Barquero Herrada

Director: Francisco Javier Soriano Camino

MADRID, JULIO 2016

Agradecimientos

Muchas gracias a Javier Soriano por su labor no solo como tutor de este trabajo, sino también como mentor durante estos dos últimos años de carrera brindándome oportunidades maravillosas.

Muchas gracias también al equipo de plataforma de Gennion Solutions: Mariano, Ignacio y Rodrigo. Por todas esas discusiones que han servido no solo para aprender, sino también para inspirar este trabajo.

Muchas gracias también a mis padres y mi hermana por apoyarme durante todos estos años.

Finalmente, quiero dedicar este trabajo a mi pareja Alba, que ha sido un apoyo constante, sobre todo en los momentos más difíciles. Gracias por animarme a continuar y no dejar nunca que abandone mis sueños.

Índice general

1. Introducción **1**

1.1. Motivaciones 2

1.2. Objetivos 3

2. El manifiesto de Sistemas Reactivos **5**

2.1. Reponsividad 6

2.2. Resiliencia 6

2.3. Elasticidad 7

2.4. Orientado a mensajes 7

3. Arquitectura hexagonal **9**

3.1. Ports 10

3.1.1. Ports primarios 11

3.1.2. Ports secundarios 11

3.2. Adapaters 11

3.2.1. Adapters primarios 12

3.2.2. Adapters secundarios 12

4. Concurrencia **13**

4.1. Modelos para la programación con concurrencia 14

4.2. Sincronización y coordinación como mecanismos de control 15

4.3. Tareas, procesos e hilos de ejecución 16

5. Aplicaciones de la concurrencia **17**

5.1. Threads y cerrojos 17
5.1.1. El estado compartido y mutable en la concurrencia 17
5.2. Sistemas de Actores 20
5.2.1. El modelo de actores 20
5.3. Corrutinas 21
5.4. Flujos 22
5.5. Futuros y promesas 23

6. Caso de estudio: *Nutshell API Rest* 25
6.1. La plataforma 25
6.2. Requisitos 26
6.3. Arquitectura software 27
6.3.1. Arquitectura de *Ports* y *Adapters* 29
6.4. Implementación 32
6.4.1. Actores 33
6.4.2. Fallos en actores: *Let it crash!* 34
6.4.3. Comunicación entre actores: *Ask Pattern* 34
6.4.4. Diseño final 37

7. Recomendaciones 39

8. Conclusiones 41

Abstract

Nowadays we are living in an era where ubiquitous computing is popular. Applications and services, that were running in the in private servers, are now transitioning to servers in the cloud, where infrastructure is offered as a service. This applications receive millions of request or process millions of bytes in data, so they are transitioning from an one big application to new software architectures and designs that can support the new requirements.

As a result of this changes, the Reactive Manifesto appeared offering a summary of ideas and concepts that help and guide developers to build scalable and resilient applications. Along with these ideas, new software architectures appeared to create modular applications. These designs allow scalability in functionality in a context where requirements are very dynamic.

Concurrent programming will be used to implement these architectures, providing to these designs the scalability they need. Actors and futures will be used to give these application the concurrency and parallelism they require.

Using all these concepts and technology together we will offered a design to build applications that can satisfy the requirements and needs of modern systems.

Resumen

Nos encontramos en un contexto en el que la informática hoy en día se centra en la computación ubicua. Las aplicaciones y servicios están migrando de estar alojadas en servidores propios de una empresa a ejecutar en la nube, utilizando infraestructuras que se ofrecen como servicios. Estas aplicaciones tienen que atender millones de peticiones o procesar millones de datos. Como consecuencia, se está produciendo una transición de las aplicaciones monolíticas del pasado a nuevas arquitecturas y diseños que permitan satisfacer estos requisitos.

Ante estos cambios, aparece el manifiesto de los Sistemas Reactivos, que recopila una serie de ideas y conceptos que para construir aplicaciones escalables, robustas y tolerantes a fallos. Conjuntamente con estas ideas aparecen nuevas arquitecturas *software* que permiten crear aplicaciones modulares y que a su vez son escalables en funcionalidad, en un contexto donde las necesidades y requisitos son muy dinámicos.

Para dar solución tecnológica a estos diseños se utilizará la programación concurrente, que proporcionará a estos diseños la escalabilidad que necesitan. Concretamente se emplearán actores y futuros, que en conjunto añadirán la concurrencia y paralelismo que requieren estas aplicaciones.

De esta forma, todas estas tecnologías y conceptos, empleados de forma conjunta ofrecerían un diseño final que permite construir aplicaciones que pueden dar soporte a los nuevos requisitos de los sistemas actuales.

Capítulo 1

Introducción

Los ordenadores han aumentado su impacto en nuestras vidas durante las últimas décadas. Han transcurrido distintas eras en la historia de la informática, buscando el progreso tecnológico y cambiando la forman en la que usamos los ordenadores. Este tránsito ha sido consecuencia principalmente de los avances en las tecnologías relacionadas con el *hardware* y el *software*, pero también se ha visto influenciado por el modo en el que el ser humano interactúa con los ordenadores.

Los primeros ordenadores eran máquinas de gran tamaño construidas solamente para aplicaciones muy concretas y específicas como por ejemplo cálculos numéricos. No existía una distinción real entre el *software* y el *hardware*, pues las instrucciones de la aplicación se ejecutaban directamente sobre la máquina, accediendo a los recursos disponibles sin necesidad de un sistema operativo.

Estos primeros ordenadores tuvieron un existo muy alto y una gran demanda a pesar de su alto coste. Con el paso del tiempo, aparecieron más innovaciones inspiradas por conceptos como permitir a varios usuarios utilizar estas máquinas o que estos usuarios pudieran ejecutar cada uno sus programas y aplicaciones de forma simultánea. Estos conceptos dieron lugar a la aparición de los *mainframes*, los sistemas operativos y el tiempo compartido entre usuarios. Unidos a estos avances, aparecieron las primeras líneas de comando que permitían por primera vez interactuar con estos sistemas.

La llegada de las redes a los ordenadores supuso un antes y un después en la informática. Las aplicaciones distribuidas no solo aprovecharon la ventaja de poder usar múltiples máquinas, sino que también permitieron la existencia de sistemas interconectados que consistían a su vez en múltiples máquinas localizadas en diferentes sitios remotos. Esto derivó también en la necesidad de crear nuevo *software* sobre las redes para dar soporte a estos sistemas distribuidos, dando lugar a los primeros *middlewares*.

La aparición de Internet se produjo gracias a la interconexión de todas las redes existentes en una red global. A su vez se produjo la transición de grandes *mainframes* a pequeños ordenadores, que dieron lugar a estaciones de trabajo y ordenadores personales, que precipitaron todavía más la evolución de la red. Además sugieron nuevas formas de interactuar con el ordenador con la aparición de interfaces gráficas que permitían una

interacción mucho más directa. La era del ordenador personal vino acompañada de una mejora sustancial en los microprocesadores y las interfaces gráficas para interacción con el usuario, así como la explosión de aplicaciones de escritorio.

Con la aparición de los móviles, las redes móviles y posteriormente los teléfonos inteligentes, el contexto tecnológico dejaba ver que se avecinaba una nueva era, las era de la computación ubicua. Los ordenadores no iban a hacerse más y más pequeños, pero estaba claro que formaban parte del día a día del ser humano en todos sus tamaños y formas. La computación ubicua vuelve a cambiar de forma drástica la forma de interactuar con la tecnología, en busca de ofrecer una interacción mucho más natural para el usuario. Los dispositivos como portátiles, tabletas, teléfonos móviles y teléfonos inteligentes han difuminado la frontera entre los diferentes tipos de dispositivos. Todo esto combinado con una conectividad inalámbrica en todo momento ha originado nuevas formas de uso y una gran libertad de movilidad con estos dispositivos.

Aunque actualmente nos encontramos alcanzando la computación ubicua, ya existen nuevas tendencias que están influenciando en el modo pensar y en como utilizamos el ordenador hoy en día. El progreso en microprocesadores está alcanzando lo límites físicos, por lo que las CPUs actuales vienen equipadas con varios núcleos. Esto unido a los nuevo requisitos de los que precisa esta nueva era de computación ubicua ha forzado que los desarrolladores, diseñadores de arquitecturas *software* y de lenguajes tengan que adaptarse a este nuevo paradigma con procesadores de varios núcleos.

Antes de estas necesidades nos encontrábamos aplicaciones de gran tamaño, monolíticas y de poca escalabilidad desplegadas en grandes servidores propios de la empresa, que ofrecían unos tiempos de respuesta de varios segundos, varias horas de baja por mantenimiento y con una cantidad de datos en el orden de giga-bytes.

Sin embargo, los requisitos actuales muestran necesidades muy distintas. Existe una gran disparidad y posibilidades de entornos de ejecución, desde un dispositivo móvil o tableta con un sistema operativo, hasta clústers en la nube con miles de procesadores con varios núcleos con otro sistema operativo. Además los usuarios esperan unos tiempos de respuesta inferiores a milisegundos, con una disponibilidad del 100 % y con datos del orden de peta-bytes.

Conociendo estos nuevos requisitos necesarios para los nuevos servicios y aplicaciones que se desarrollan y necesitan hoy en día, se observa que las antiguas arquitecturas con constaban de grandes aplicaciones muy acopladas y con poca escalabilidad no son válidas hoy en día. Y a raíz de estas necesidad surge la idea de las aplicaciones y diseños reactivos.

1.1. Motivaciones

En un contexto en el que la computación ubicua y la computación en la nube están ganando cada vez más importancia, es necesario replantear los diseños y arquitecturas que hasta ahora existen, y ofrecer nuevos diseños que ofrezcan una mejor solución a los nuevos requisitos que presenta el contexto actual.

El manifiesto de Sistemas Reactivos recoge una serie de conceptos e ideas que debe tener una aplicación o sistema que quiera considerarse reactivo. Todas esas ideas y conceptos que conciernen a una aplicación reactiva nos puede ayudar a diseñar estas nuevas aplicaciones.

Para dar escalabilidad a las aplicaciones, existen diversas tecnologías y conceptos que permiten dotar a un sistema o aplicación de concurrencia.

Además existen diversas arquitecturas *software* que buscan adaptarse a un contexto en el que las necesidades y requisitos son muy dinámicos, guiando el desarrollo a diseño que sean modulares.

Pero todavía no existe una propuesta que ponga en uso todos estos conceptos para ofrecer un nuevo diseño que de solución a los nuevos requisitos y retos que supone desarrollar aplicaciones que cumplan con las expectativas del contexto actual.

1.2. Objetivos

El objetivo de este trabajo es explorar las tecnologías actuales y las arquitecturas *software* hasta ahora conocidas para proponer un diseño que permita a los desarrolladores crear aplicaciones que sean reactivas y con escalabilidad funcional, para así dar soporte a los nuevos requisitos que han surgido a raíz de los nuevos escenarios en los que localizamos la informática.

Explorará las tecnologías que permiten desarrollar una aplicación con concurrencia, a la vez que se buscará una arquitectura que permite diseñar aplicaciones modulares y reactivas.

Capítulo 2

El manifiesto de Sistemas Reactivos

El manifiesto de Sistemas Reactivos [1] resume y plantea cuáles son las bases para abordar el desarrollo y diseño de aplicaciones y servicios que cumplan con las expectativas y necesidades de los usuarios actuales conjuntamente con los nuevos requisitos y tecnologías emergentes.

La idea central del manifiesto se encuentra alrededor de cuatro características que toda arquitectura reactiva necesita cumplir para conseguir el objetivo. Todas y cada una de estás características no son elementos nuevos y novedosos, sino un conjunto de ideas y conceptos que ya eran conocidos y que juntos permiten alcanzar el objetivo. Estas características son:

- Responsividad
- Resiliencia
- Elasticidad
- Orientados a mensajes

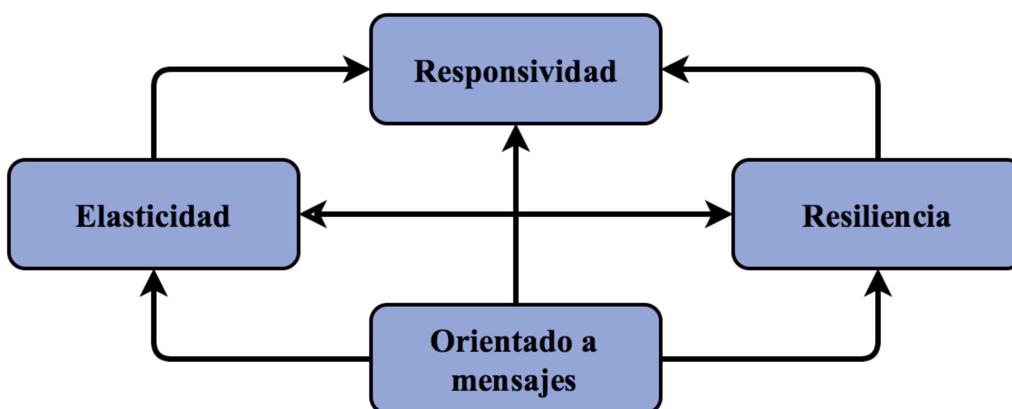


Figura 2.1: Características de un sistema reactivo.

Los sistemas que cumplan estas cuatro características serán más flexibles, estarán poco acoplados y serán escalables. Esto los hará más fáciles de mantener a largo plazo, así como ampliar su funcionalidad si es necesario.

Además, estos sistemas serán más tolerantes a fallos, y en caso de fallo, serán capaces de asumirlo y tratarlo sin causar un desastre total del sistema. Por último, tendrán una gran capacidad de responsividad que dará un servicio efectivo y adecuado a los usuarios.

2.1. Reponsividad

Cuando hablamos de un sistema nos referimos a un conjunto de componentes que trabajan de forma colaborativa para ofrecer unos servicios a los usuarios. Estos sistemas pueden ser desde sistemas pequeños con muy pocos servicios, hasta sistemas muy grandes y complejos compuestos por ciertos o miles de servicios que trabajan de forma colaborativa.

Un sistema se considera *responsive* si es capaz de responder en tiempos adecuados en la medida de lo posible. Estos sistemas *responsive* se centran en proveer tiempos de respuesta rápidos y consistentes, dentro de unos límites superiores establecidos de manera que provean una calidad de servicio consistente. Este comportamiento a cambio simplifica el manejo de errores y aumenta la confianza del usuario final y su fidelización.

2.2. Resiliencia

Un sistema se considera resiliente si es capaz de permanecer *responsive* ante situaciones de fallo. Se entiende por fallo un evento inesperado en un servicio que provoca un mal funcionamiento del mismo, evitando que continué ejecutando de forma normal, y por lo tanto, que deje de ofrecer su funcionalidad.

El concepto de resiliencia no se aplica solo a sistemas de misión crítica o de alta disponibilidad. Cualquier sistema que no es resiliente no podrá ser *responsive* después de un fallo.

Para alcanzar la resiliencia, se requerirá replicación, contención, aislamiento y delegación. De esta manera, los fallos estarán contenidos dentro de cada componente del sistema, aislando los componentes entre ellos y asegurando que las partes pueden fallar y recuperarse sin afectar al resto del sistema. La recuperación de cada componente se delega en otro componente externo, y la alta disponibilidad puede asegurarse mediante replicación cuando sea necesaria.

2.3. Elasticidad

Un sistema se considera elástico cuando es capaz de mantenerse *responsive* bajo todas las posibles variaciones de carga de trabajo a las que puede verse sometido. Además no solo tiene que mantenerse *responsive* sino que también tiene que ser capaz de reaccionar adaptándose a los cambios de carga de trabajo, aumentando o disminuyendo los recursos encargados de tratar dicha carga según sea necesario.

Para alcanzar esta propiedad se han de utilizar diseños que no tengan puntos de contención o cuellos de botella centralizados, permitiendo así que el sistema tenga la habilidad de fragmentar o replicar los componentes y distribuir las peticiones entre ellos.

Los sistemas reactivos soportan algoritmos de escalado predictivos y reactivos, proveyéndolos de medidas de rendimiento en tiempo real. Esto permite a los sistemas alcanzar la elasticidad de forma efectiva a nivel coste.

2.4. Orientado a mensajes

Los sistemas reactivos se basan en el intercambio de mensajes de forma asíncrona para establecer el límite entre componentes, lo que asegura desacoplamiento, aislamiento y transparencia de ubicación. Esta definición de los límites y comunicación mediante mensajes también proporciona los medios para delegar los fallos como mensajes.

Empleando un intercambio de mensajes explícito se puede conseguir controlar y manejar la carga y proporcionar elasticidad y control de flujo, monitorizando las colas de mensajes en el sistema y aplicando *back pressure* si es necesario.

Utilizando transparencia de ubicación en el intercambio de mensajes permite que sea posible el manejo de los fallos de forma idéntica en un clúster o en un único nodo.

Además, si utilizamos un intercambio de mensajes no bloqueante permite que los receptores solo consuman recursos si están activos, evitando así una sobrecarga del sistema.

Capítulo 3

Arquitectura hexagonal

La idea principal de esta arquitectura busca desacoplar la lógica principal que representa el núcleo del negocio de los diferentes servicios que dicha lógica puede hacer uso. Esto permite utilizar y conectar diferentes servicios a esta lógica de negocio, y a su vez poder definir y hacer funcionar dicha lógica sin necesidad de tener conectada ningún servicio.

La lógica principal, o lógica de negocio, de una aplicación consiste en los algoritmos que definen su propósito. Estos algoritmos implementan los casos de uso que componen la definición del propósito del núcleo de la aplicación. Si se producen cambios en ellos, la esencia de la propia aplicación cambia.

Sin embargo los servicios que conectamos a esta lógica no son necesarios. Deben poder cambiarse y reemplazarse sin afectar al propósito general de la aplicación. Algunos ejemplos de servicios pueden ser: acceso a la base de datos u otro tipo de almacenamientos, interfaces gráficas, correo electrónico y otros elementos de comunicación, dispositivos *hardware*...

En 3.1 se muestra un ejemplo de arquitectura hexagonal, también conocida como arquitectura de *ports* y *adapters*. El ejemplo muestra un caso en el que la arquitectura tiene seis *ports*, para mostrar el caso modelo del que deriva el nombre de la arquitectura. Pero la idea importante que se quiere reflejar en el nombre no es el número de *ports*, sino que en el centro de la arquitectura localizamos el núcleo de la aplicación, que representa la lógica principal de la misma. Un número realista de *ports* suele ser entre dos y cuatro.

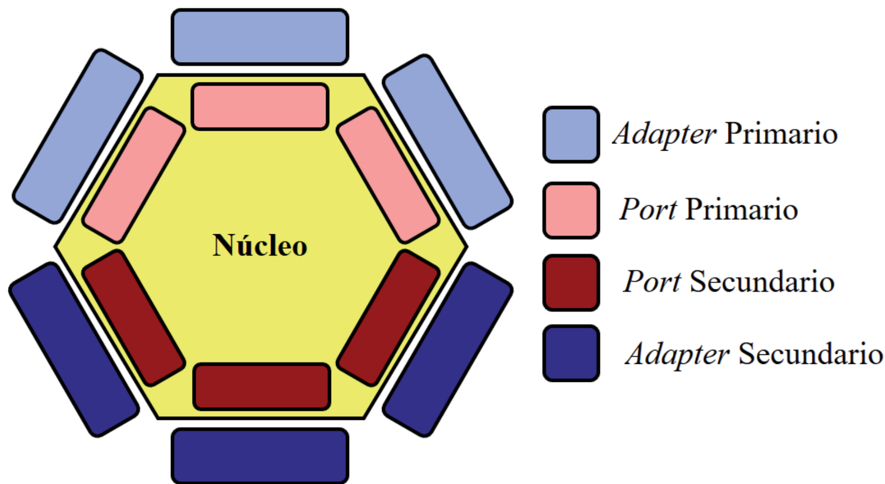


Figura 3.1: Arquitectura hexagonal o de *ports* y *adapters*.

Las ventajas que esta arquitectura presenta son:

- La lógica principal de negocio puede ser testeada y comprobada independientemente de los servicios que tenga conectados.
- Es más fácil cambiar unos servicios por otros más adecuados en caso de que los requisitos de la aplicación cambien.

3.1. Ports

Un ***port*** es un punto de entrada de información a la lógica del núcleo. Para ello el núcleo define una interfaz, que el *port* utiliza para realizar la entrada de información. Dentro de los *ports* podemos distinguir dos tipos:

- *Ports* primarios.
- *Ports* secundarios.

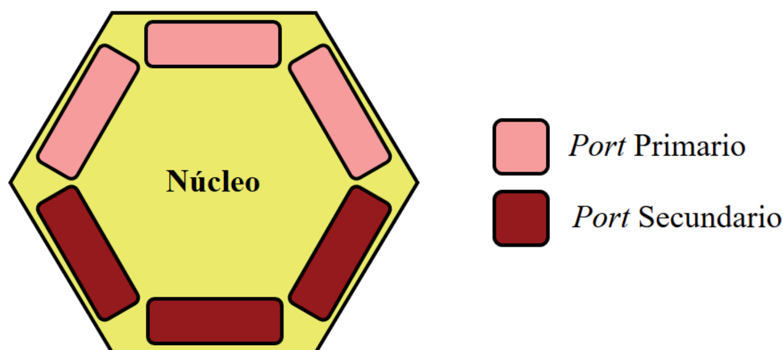


Figura 3.2: *Ports* en la arquitectura hexagonal.

3.1.1. Ports primarios

Se define como *port* primario a la API principal de la aplicación, es decir, la interfaz que que ofrece la lógica del núcleo para la entrada de información. Estos *ports* son invocados y utilizados por los *adapters* primarios, que son encargados de la interacción con el usuario de la aplicación. Algunos ejemplos de *ports* primarios son las funciones que permiten cambiar objetos, atributos y relaciones en la lógica del núcleo.

3.1.2. Ports secundarios

Se define como *port* secundario a la interfaz definida en la lógica del núcleo para los *adapters* secundarios. Estos *adapters* son invocados por la lógica del núcleo utilizando las interfaces definidas. Un ejemplo de *port* secundario es la interfaz para guardar objetos. Esta interfaz no ofrece ninguna información a cerca del objeto, simplemente ofrece una interfaz para crear, conseguirlo, actualizarlo y borrarlo.

3.2. Adapaters

Un ***adapter*** es el puente de unión entre la aplicación y los servicios que requiere. Para ello, estos *adapters* usarán las interfaces definidas por la lógica del núcleo, es decir, los *ports*. Al igual que pasa con los *ports*, los *adapters* se clasifican en dos tipos:

- *Adapters* primarios.
- *Adapters* secundarios.

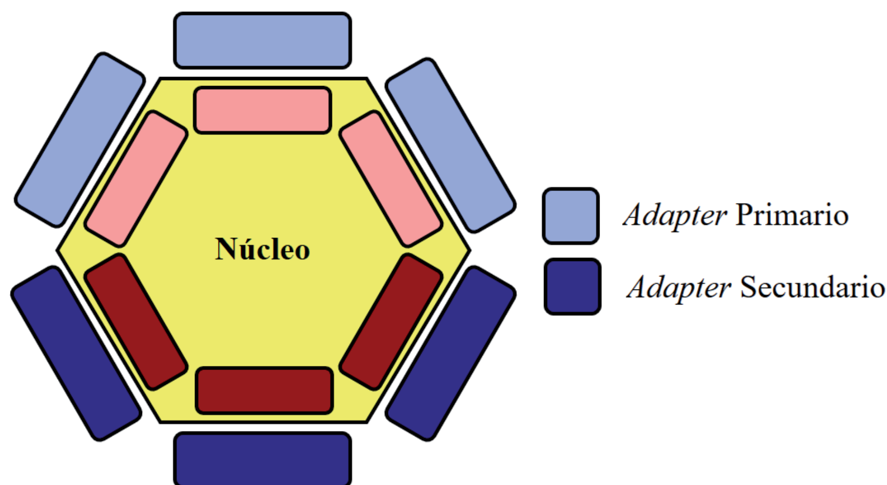


Figura 3.3: *Adapters* en la arquitectura hexagonal.

3.2.1. Adapters primarios

Se define *adapter* primario como la pieza de código entre el usuario y la lógica del núcleo. Un ejemplo de adapter es un test unitario de la lógica del núcleo. Otro ejemplo es el componente que hace las funciones de controlador para la interacción entre la interfaz gráfica para el usuario y la lógica del núcleo. Los *adapters* primarios llaman a la API definida en los *ports* primarios para interactuar con la lógica de núcleo.

3.2.2. Adapters secundarios

Se define *adapter* secundario como la implementación de los *ports* secundarios, los cuales solo definen la interfaz que el núcleo de la lógica va a utilizar. Un ejemplo es la implementación de la interfaz para guardar objetos, que convertiría las peticiones definidas en la interfaz del *port* en llamadas a una base de datos que lleven a cabo la acción, y devolvería los resultados según se hayan definido en el *port*. Otro ejemplo es sería el *mock* del objeto que simula la interacción con la base de datos para realizar los test unitarios de ciertas partes de la lógica del núcleo.

Capítulo 4

Concurrencia

La concurrencia es una propiedad de todo sistema que permite representar el hecho de que se pueden ejecutar varias actividades de forma simultánea. Además, dichas actividades pueden incluso interactuar entre ellas. Estas ejecuciones pueden tener lugar en diferentes entornos, como procesadores de un solo núcleo, procesadores de varios núcleos, multi-procesadores o incluso múltiples máquinas que forman parte de una sistema distribuido. A pesar de las diferencias, todos estos entornos tienen una característica común: proveer de mecanismos para permitir el control de los diferentes flujos de ejecución mediante la coordinación y la sincronización, y a la vez asegurar la consistencia.

El uso de la concurrencia viene motivado no solo por los nuevos procesadores con varios núcleos o los sistemas multi-procesador, sino también por la búsqueda de *software* que ofrezca un mayor rendimiento y aprovechamiento de la plataforma sobre la que ejecuta. Cantrill [2] además establece que concurrencia puede mejorar el rendimiento en tiempo de ejecución de una aplicación de tres maneras:

- Reducir la latencia: una tarea puede reducir su tiempo de ejecución si se consigue dividir en tareas más pequeñas que pueden ejecutar de forma concurrente.

- Ocultar la latencia: permite al sistema continuar ejecutando otras tareas si alguna de ellas está bloqueada a la espera de un operación de gran latencia como por ejemplo una operación de entrada/salida de disco duro.

- Aumentar el rendimiento: al ejecutar tareas de forma concurrente, el sistema puede soportar más carga y sacar mayor aprovechamiento de los recursos disponibles.

Además la presencia de la concurrencia es una propiedad intrínseca para cualquier tipo de sistema distribuido. Los procesos corriendo en diferentes máquinas forman un sistema común que ejecuta código en diferentes máquinas al mismo tiempo.

4.1. Modelos para la programación con concurrencia

En el libro de Von Roy [3] se introducen cuatro modelos de concurrencia a la hora de programar empleando este concepto. Estos son:

- Secuencial.
- Declarativa.
- Paso de mensajes.
- Estado compartido.

Concurrencia secuencial

En un modelo de programación determinista no se aplica concurrencia. En su forma más estricta, existe un orden total de todas las operaciones que se va a ejecutar. Pero en un modelo menos estricto, se puede conservar que el comportamiento sea determinista. Sin embargo, o bien no se puede garantizar el orden de ejecución del programa o se tiene que proveer de un mecanismo para declarar de forma explícita los flujos y transiciones entre tareas, como es el caso de las corrutinas, por ejemplo.

Concurrencia declarativa

La programación declarativa es un modelo de programación que favorece que el control de los flujos de ejecución sean implícitos. El control no se describe directamente, sino más bien es un resultado de la lógica y declaración del programa.

La concurrencia declarativa extiende el modelo de programación declarativa permitiendo que existan varios flujos de ejecución. Esto añade concurrencia de forma implícita, dirigida por los datos que viajan por le programa o por la demanda que se le exige al mismo. Aunque esta aproximación introduce cierto indeterminismo en tiempo de ejecución, este indeterminismo no se aprecia generalmente desde fuera.

Concurrencia con paso de mensajes

Este modelo permite que las tareas que están ejecutando concurrentemente se comuniquen por mensajes. Normalmente es el único mecanismo que permite a las tareas comunicarse e interactuar entre ellas, pues de otro modo se encuentran ejecutando totalmente aisladas de las demás. El paso de mensajes puede ser síncrono o asíncrono, dando lugar diferentes tipos de mecanismos y patrones para las sincronización y coordinación de las tareas.

Concurrencia con estado compartido

La concurrencia con estado compartido es un modelo de programación muy extendido en el que varias tareas puede acceder a recursos y estados. Compartir exactamente los mismos recursos y estados entre las diferentes tareas requiere mecanismos para sincronizar el acceso y coordinar las tareas. Pero este modelo introduce indeterminismo que puede derivar en problemas de consistencia en los recursos o estados.

4.2. Sincronización y coordinación como mecanismos de control

Independientemente del modelo de programación concurrente que se escoja, debe existir algún mecanismo de control sobre la concurrencia, ya sea de forma implícita o explícita. La ejecución de varias flujos de ejecución de forma simultanea sobre el mismo espacio de direcciones puede ser muy peligroso y un descontrol si no existe ningún mecanismos que garantice los accesos de forma ordenada. Dos o más flujos podrían intentar acceder a la misma información de forma simultánea e intentar modificarla introduciendo información que podría dejar el sistema en un estado inconsistente o no válido. Además, si varias tareas trabajan de forma conjunta para resolver un problema necesitan estar de acuerdo en los datos en los que acceden según va progresando la ejecución del problema. Todo esto es la base de los retos que hay que afrontar a la hora de utilizar la concurrencia y la programación concurrente.

La sincronización y la coordinación son dos mecanismos que permiten lidiar con estos problemas. La sincronización, o mejor expresado sincronización por competición, es el mecanismo que controla el acceso a los recursos compartidos entre varias tareas. Es especialmente importante cuando varias actividades necesitan acceder a un recurso que no puede ser accedido de forma simultánea. Un mecanismos de sincronización adecuado obligará a tener la exclusividad sobre el recurso para acceder y ordenará los accesos al recurso por las diferentes tareas. La coordinación, también conocida como sincronización cooperativa, realiza una orquestación de forma colaborativa de las tareas sobre recurso requerido.

En la práctica, la sincronización y la coordinación a veces se utilizan conjuntamente. Ambos mecanismos pueden ser explícitos o implícitos. La sincronización de forma implícita oculta la sincronización como parte de la semántica del lenguaje, por lo que no se muestra en el código visible del programa. Por el contrario, la sincronización de forma explícita requiere que el programador añada de forma explícita las operaciones en el código para indicar la sincronización.

4.3. Tareas, procesos e hilos de ejecución

A continuación utilizaremos el término *tarea* para referirnos a la abstracción de una unidad de ejecución.

La habilidad de ejecutar múltiples tareas de forma concurrente, es decir, la multitarea, ha sido un requisito fundamental para los sistemas operativos. Con este mecanismo se consigue intercalar y alternar la ejecución de tareas. En el caso de tener procesadores con múltiples núcleos o varios procesadores, el caso de la multitarea se complementa con el multiprocesador, que permite distribuir las tareas entre los diferentes núcleos/procesadores disponibles. El concepto clave para ambos mecanismos es la programación o *scheduling*, que se encarga de organizar y asignar los tiempos de procesador a cada tarea siguiendo según estrategias definidas. Cada estrategia puede tener diferentes objetivos, como una distribución equitativa entre las tareas o tiempos máximos de ejecución de tareas. Otro tipo de estrategia es el modelo de asignación. En este modelo, el *scheduler* asigna un tiempo de ejecución a la tarea y pasado el tiempo se lo revoca. La tarea no tiene ningún control sobre la asignación de tiempo. En un modelo cooperativo, la propia tarea tiene la responsabilidad de ceder el recursos después de un tiempo para permitir a otras tareas ejecutarse. Es por esto que la programación o *scheduling* es un labor muy importante para cualquier sistema operativo. Sin embargo, también hay que tener en cuenta que las propias aplicaciones también pueden tener su propio *scheluder* para gestionar sus propias tareas.

A nivel de sistema operativo podemos encontrar dos tipos de tareas: los procesos y los hilos. Esencialmente, cada uno representa tareas de diferente granularidad. Un proceso es un tarea grande y pesada, con sus propios recursos como memoria y descriptores de ficheros, que gestiona y aprovisiona el sistema operativo. Los hilos de ejecución, por su parte, son más ligeros y pertenecen a un proceso en concreto. Todos los hilos de un mismo proceso comparten la memoria, los descriptores de ficheros y todos los recursos asociados a dicho proceso. Crear un hilo es una operación menos costosa operacionalmente comparada con crear un nuevo proceso.

La mayoría de las aplicaciones concurrentes hace mucho uso de múltiples hilos. Sin embargo, estoy no implica que el propio lenguaje permita y contemple los hilos como entidades propias. En su lugar, el entorno de ejecución puede transformar las entidades concurrentes de un lenguaje a hilos en tiempo de ejecución.

Capítulo 5

Aplicaciones de la concurrencia

5.1. Threads y cerrojos

La programación imperativa es el modo de programar más popular y extendido a la hora de estructurar el desarrollo de un programa. Este tipo de programación se encuentra basada en la idea de la ejecución de una secuencia de acciones y en la idea de la modificación de un estado, en principio mutable. Estás ideas y conceptos vienen heredadas del diseño de un procesador desde la arquitectura de una máquina de *Von Neumann*.

Los hilos de ejecución son una consecución natural de esos conceptos ante la necesidad de tener varios flujos de control y ejecución simultáneamente. Después de los procesos pesados, los hilos de ejecución son la principal forma de conseguir paralelismo en una aplicación. Normalmente encontramos soporte para hilos de ejecución o bien ofrecido por el propio sistema operativo o bien por la propia arquitectura *hardware* (por ejemplo, *hyperthreading*). Es por esta razón que los hilos de ejecución son el mecanismo básico para construir y ofrecer concurrencia en la mayoría de lenguajes. Sin embargo, si no se hace uso de un mecanismo de sincronización adecuado a la hora de acceder al estado compartido, la programación concurrente puede llegar a ser muy tediosa y propensa a error.

5.1.1. El estado compartido y mutable en la concurrencia

Conceptualmente, un hilo de ejecución se describe como un flujo de control secuencial, totalmente aislado de los demás a simple vista. Pero a diferencia de los procesos, los hilos de ejecución comparten el mismo espacio de direcciones de memoria, entre otros recursos. Esto implica que múltiples hilos de ejecución podrían acceder a un mismo dato de forma concurrente. Además, si añadimos mutabilidad a estos datos, significa que podríamos encontrar varios hilos de ejecución compitiendo por realizar operaciones de escritura sobre dichos datos, pudiendo crear inconsistencias en los datos del programada. Esto representa un gran grado de indeterminación en la ejecución. Si no se tiene cuidado, la mutabilidad del estado y la indeterminación de las operaciones introduce un gran riesgo

de que se produzcan condiciones de carrera.

Una condición de carrera ocurre cuando dos o más hilos de ejecución compiten por el acceso a una sección crítica, la cual contiene un estado compartido entre los hilos de ejecución. Dada la gran variedad de posibilidades de acceso, las condiciones de carrera pueden derivar en varias inconsistencias en el estado. Por ejemplo, un hilos de ejecución puede leer un estado obsoleto mientras otro está actualizándolo. Cuando múltiples hilos de ejecución modifican un estado al mismo tiempo, solo uno de ellos será el que quede reflejado, y los demás se perderán, llegando incluso a guardarse un estado inconsistente. Es por esto que aparece la necesidad de mecanismos para proteger las secciones críticas y forzar el acceso de forma sincronizada.

Cerrojos

La forma más intuitiva para conseguir el acceso de forma sincronizada a las secciones críticas son los cerrojos. Existen diferentes tipos de cerrojos con diferentes comportamientos y semántica. Los semáforos son un tipo de cerrojos muy sencillos que ofrecen dos métodos: *wait* y *signal*. Cuando un semáforo va a acceder una sección crítica o un recurso compartido, ejecuta la función *wait*. Una vez haya terminado de utilizar el recurso o vaya a salir de la sección crítica, sale usando la función *signal*. El semáforo impide que múltiples hilos de ejecución consigan el semáforo al mismo tiempo impidiendo que otro consigan pasar en la función *wait*.

Además de los semáforos podemos encontrar implementaciones más complejas como los monitores o cerrojos de lectores/escritores entre otros.

Las consecuencias de los cerrojos

Los cerrojos nos permiten secuenciar el acceso a las secciones críticas. Identificando las secciones de código vulnerables a condiciones de carrera y colocando cuidadosamente cerrojos alrededor podemos conseguir reducir la indeterminación y forzar el acceso de forma secuencia a dichas regiones.

Sin embargo, el concepto de los cerrojos ha introducido otro tipo de problemas para el código multi-hilo. Si se usan los cerrojos inapropiadamente, la aplicación puede alcanzar un estado inconsistente en tiempo de ejecución debido a que los cerrojos no se liberan nunca, o los hilos de ejecución se quedan en espera a que un cerrojo se libere pero eso nunca sucede. Este tipo de errores siempre son susceptibles de ocurrir si el programador tiene que usar explícitamente las funciones como *wait* o *signal* para proteger las secciones críticas. Abstracciones de más nivel como los monitores ofrece mecanismos para definir y marcar una sección crítica y que de forma implícita y transparente para el programador, gestionando de forma automática los cerrojos.

Sin embargo, todavía pueden aparecer problemas ocasionados por los cerrojos. El más notorio es el conocido como interbloqueo. Esto ocurre cuando dos o más hilos de ejecución compiten por obtener unos cerrojos que tiene dependencias entre ellos. El escenario

más simple son dos hilos de ejecución, ambos con su cerrojo propio, pero además necesitan adquirir el cerrojo del otro para poder avanzar. Como ninguno de los dos puede avanzar pues el otro cerrojo está bloqueado, ninguno de los dos puede avanzar, produciéndose el interbloqueo.

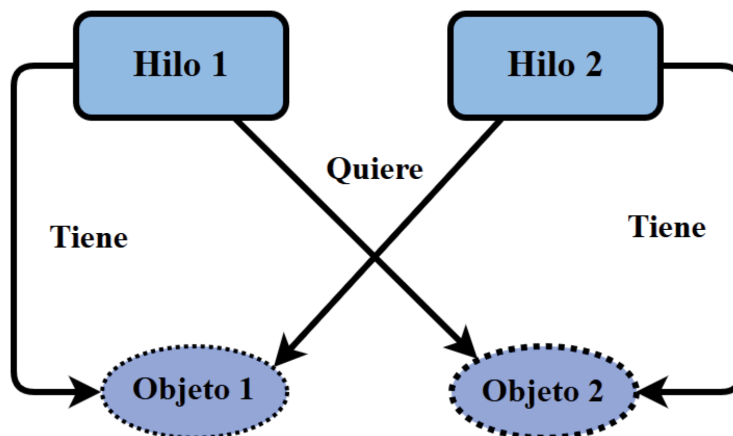


Figura 5.1: Ejemplo básico de interbloqueo.

Otro problema son los *livelocks* y la inanición de cerrojos. Parecido a los interbloqueos, un *livelock* impide que los hilos de ejecución continúen. Sin embargo, no están bloqueados, sino que se encuentran cambiando constantemente de estado en respuesta a los cambios de estado de los otros hilos de ejecución involucrados, que a su vez también están cambiando de estado. Los *livelocks* se pueden considerar un tipo especial de inanición. La inanición se considera el caso en el que un hilo de ejecución intenta de forma repetida y constante acceder y conseguir un cerrojo para un recurso compartido pero no lo puede conseguir porque existen otros hilos de ejecución que también los quieren y lo consiguen siempre.

Mientras que los casos de inanición como los *livelocks* pueden solucionarse en tiempo de ejecución usando gestores que detecten y apliquen una política en esos casos, los casos potenciales de interbloqueos son generalmente muy difíciles de detectar debido a su indeterminismo. El riesgo de interbloqueos aumenta cuando existen múltiples cerrojos de diferentes secciones críticas en uso. Para evitar esto en gran parte se recomienda el uso de pocos cerrojos que cubran regiones más amplias de código en lugar de varios cerrojos distintos que cubran diferentes regiones. Utilizando regiones de código críticas más grandes y protegiéndolas con cerrojos podemos asegurar una ejecución secuencial y en orden sobre ellas. Sin embargo, usando cerrojos en regiones así de grandes termina produciendo una ejecución secuencial de los propios hilos de ejecución. Esto es efecto totalmente contrario al propósito principal de aumentar el paralelismo de la aplicación.

A parte de estos problemas, existe otra dificultad cuando los cerrojos están al uso. Si tenemos múltiples regiones de código con secciones críticas protegidas por cerrojos, no podemos garantizar que la composición de dichas regiones de código protegidas por cerrojos no termine en un interbloqueo. Básicamente, no podemos componer implementaciones que aseguren la consistencia y el acceso ordenado sin arriesgarnos a nuevos problemas de cerrojos.

A pesar de todos los problemas visto, la programación concurrente basada en hilos de ejecución, usando cerrojos como mecanismos de sincronización y un estado compartido está al uso hoy en día y disponible en casi todos los lenguajes. Es importante tener en cuenta que esta forma de afrontar la programación concurrente es de bajo nivel. Está más cerca de la base de la concurrencia, y cada uno de los conceptos que veremos a continuación lo usarán como base para su implementación interna

5.2. Sistemas de Actores

Los principales modelos de concurrencia considerados hasta ahora tienen como noción un estado compartido. Este estado compartido puede ser accedido por múltiples hilos de ejecución al mismo tiempo y por lo tanto hay que utilizar mecanismos de protección como los cerrojos o monitores. El tener un estado compartido y mutable no solo es algo inherente en los modelos, también es algo inherente en la complejidad del problema.

Un sistema de actores nos ofrece un punto de vista distinto, en el que se deshecha la idea de un estado compartido por todos. El estado sigue siendo mutable, pero sin embargo su acceso está restringido a una entidad única en concreto, que son las que pueden modificarlo, los actores.

5.2.1. El modelo de actores

El modelo de actores se base en dos conceptos, el modelo de concurrencia y el paso de mensajes. La idea fundamental del modelo de actores es el uso de los actores como unidad mínima de concurrencia que puede actuar de diferentes maneras según los mensajes que recibe:

- Mandar un número finito de mensajes a otros actores.
- Crear un número finito de nuevos actores.
- Cambiar su comportamiento interno, que entrará en vigor para los siguientes mensajes que atienda.

Para la comunicación, el modelo de actores usa paso de mensajes de manera asíncrona. Además, no usa ningún mecanismo intermedio como canales. En su lugar, cada actor posee un buzón (*mailbox*) al que todos pueden direccionar. Hay que tener en cuenta que esa dirección para el buzón no debe confundirse con la identidad del actor, pues un actor puede tener ninguna, una o varias direcciones. Cuando un actor manda un mensaje, este debe conocer la dirección del actor destino. Además los actores pueden mandarse mensajes a sí mismos, y ellos lo recibirán y tratarán en el futuro. Otro elemento a tener en cuenta es que el mapeo de las direcciones y actores no forma parte del modelo conceptual, aunque es una característica de la implementación.

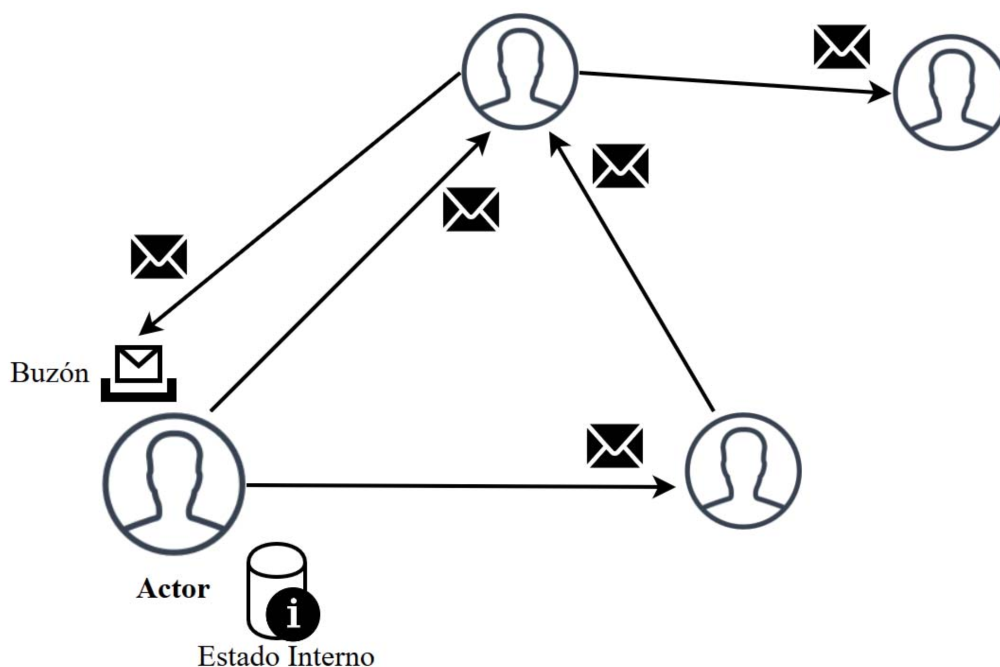


Figura 5.2: Ejemplo sistema de actores.

Los mensajes son enviados de forma asíncrona y puede llegar al buzón del destinatario en un tiempo arbitrario. Además, el modelo de actores no garantiza el orden de llegada de los mensajes. El encolado y desencolado de los mensajes en el buzón son operaciones atómicas, por lo que no se puede producir una condición de carrera en el tratamiento de los mensajes. Además, un actor procesa los mensajes del buzón de forma secuencial reaccionando de las tres posibilidades descritas antes. La tercera posibilidad, cambiar su comportamiento interno, permite finalmente tratar con los casos en los que se precisa un estado mutable. Sin embargo, este nuevo comportamiento solo se aplicará cuando se haya realizado el procesamiento de dicho mensaje. Además, todos las operaciones de procesamiento de mensajes representan una operación sin efectos secundarios desde un punto de vista conceptual.

El modelo de actores puede utilizarse para modelar sistemas inherentemente concurrentes, ya que cada actor es completamente independiente de cualquier otra instancia. No existe un estado compartido entre diferentes actores y las interacciones entre los actores están totalmente basadas en paso de mensajes de forma asíncrona, como se muestra en 5.2.

5.3. Corrutinas

Las corrutinas son un generalización de las subrutinas. Mientras que una subrutina se ejecuta secuencialmente y toda de una vez, una corrutina puede suspenderse y continuar su ejecución en distintos lugares del código. Además, las corrutinas son una buena unidad mínima en el contexto de la concurrencia ya que permite la ejecución de tareas de

forma colaborativa, permitiendo que se pasen entre ellas el control del contexto según se requiera.

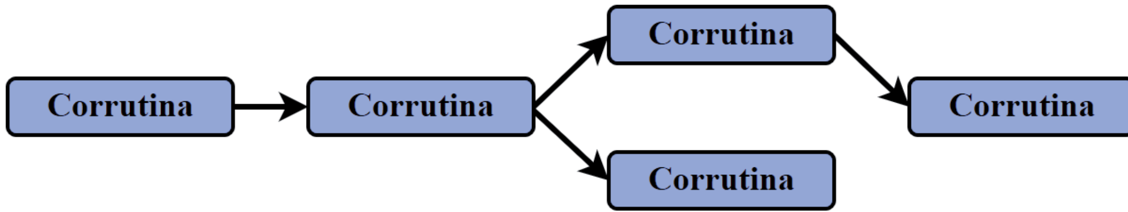


Figura 5.3: Ejemplo de corruinas.

Las corrutinas se consideran en muchos casos una alternativa a los hilos de ejecución ya que implementan conceptos de más alto nivel en lo referente a concurrencia. Algunos sistemas de actores se encuentran desarrollados sobre corrutinas.

5.4. Flujos

La programación mediante flujos es una forma muy elegante pero poco común de afrontar el desarrollo de aplicaciones que requieren concurrencia. La programación imperativa se basa en la idea de describir una secuencia explícita de operaciones a realizar. Sin embargo, la programación mediante flujos define la relación entre las operaciones, creando un grafo de dependencias que representa los flujos de ejecución. Esto permite a los sistemas de ejecución identificar de forma automática pasos independientes en las operaciones y paralelizarlos en tiempo de ejecución. La coordinación y sincronización quedan totalmente ocultas en el sistema de ejecución, normalmente usando canales que esperan las múltiples entradas de información y luego inician la ejecución de dicha entrada.

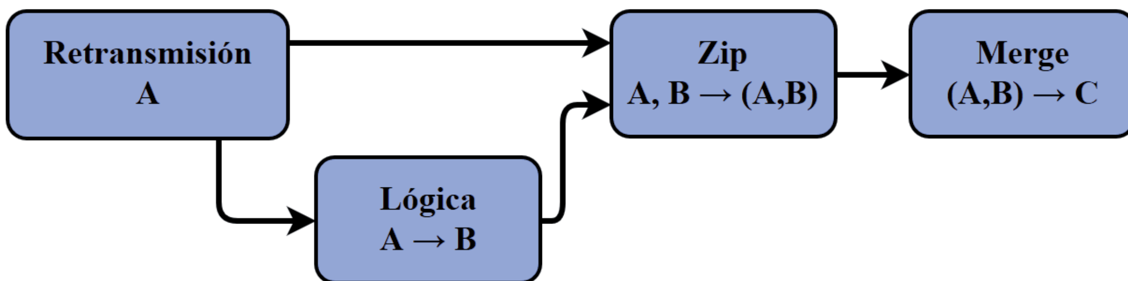


Figura 5.4: Ejemplo de flujos.

5.5. Futuros y promesas

Los futuros y promesas se basan el la idea de desacoplar la ejecución y la devolución del resultado, utilizando una entidad intermedia que devuelve el resultado una vez está disponible. En la programación concurrente, los futuros y las promesas se utilizan para proporcionar asincronía para tareas que pueden ejecutar en segundo plano. Una vez que la tarea se ha lanzado, el flujo de la entidad que ha hecho la llamada a ese futuro o promesa puede continuar ejecutando independientemente de la ejecución del futuro o promesa. El resultado de la ejecución del futuro o promesa puede ser solicitado posteriormente. En caso de no encontrarse disponible todavía, se bloqueará o mandará una notificación en el caso de ser no bloqueante.

Los futuros y promesas también introducen un mecanismo de sincronización, pues aunque permite lanzar ejecuciones independientes, permite sincronizarlas con el flujo inicial de control cuando este pregunta por su resultado y espera que retorne. Muchas de las implementaciones disponibles en múltiples lenguajes se encuentra hechas con hilos de ejecución para ofrecer la concurrencia. De esta manera la ejecución del futuro o promesa se realiza en otro hilo de ejecución, permitiendo al hilo de ejecución inicial continuar su ejecución.

En un sistema implementado con actores, los actores pueden verse e interpretarse como futuros. Enviar un mensaje a otro actor y esperar a que eventualmente responda se puede abstraer muchas veces con futuros.

Capítulo 6

Caso de estudio: *Nutshell API Rest*

Todos los conceptos y tecnologías descritos anteriormente nos sirven para analizar y dar solución a un caso concreto de estudio. En nuestro caso este caso de estudio será una API RESTful encargada encargada de gestionar y administrar la información sobre parámetros de configuración y datos de gestión de la plataforma BigData / FastData de Gennion Solutions.

6.1. La plataforma

La plataforma de Gennion Solutions es una plataforma de recolección y procesamiento de datos procedentes de dispositivos con conectividad inalámbrica. La recolección de datos de estos dispositivos como teléfonos móviles o tabletas, entre otros, se realiza de manera masiva utilizando varias nodos, *harvesters* , que monitorizan y escuchan la información generada por estos dispositivos. Toda esta información se transmite a la plataforma a través de una serie de servicios encargados de retransmitirla a la plataforma para su posterior procesamiento. Para recolectar todos estos datos, los *harvesters* necesitan funcionar muy rápido y bajo grandes cargas de datos, procesando la la información de manera inmediata y sin retrasos para evitar así la perdida de datos.

Una vez el dato entra en la plataforma, pasará por una serie de servicios que lo procesarán y transformarán para extraer más información. Estos servicios encargados de procesar el dato están sometidos a una gran carga de trabajo, derivada de la carga de recolección de datos, por lo que estos servicios deben soportarla. Además, el procesamiento de estos datos no se realiza de manera puntual, sino que el dato se procesa según se va recibiendo, es decir, en tiempo real.

Tras procesar estos datos, toda la información que se extraiga en cada una de las etapas del procesamiento a de guardarse de forma temporal con algún mecanismo que permita almacenar esta información de forma persistente. Además, esta información almacenada se expondrá para consulta a través de diferentes servicios RESTful para los clientes.

Figura 6.1: Plataforma de Gennion Solutions.

Para gestionar y poder extraer más información acerca de los datos que van entrando en la plataforma, debe existir una configuración que establezca los parámetros a utilizar según a dónde pertenezca el dato. Para ello la plataforma contiene información sobre la definición de los proyectos que están en funcionamiento. Los proyectos contienen información acerca de los diferentes nodos y su localización, para saber a qué clientes pertenecen y los parámetros a aplicar según su localización.

Todos estos servicios que se encargan de recolectar y procesar la información para posteriormente almacenarla de forma persistente necesitarán poder aplicar la configuración adecuada a cada dato recibido. Para evitar tener que definir en cada uno de ellos esta configuración acerca de proyectos, existe un servicio encargado de gestionar y almacenar dicha configuración. Este servicio servirá de fuente de información para el resto de servicios que necesiten preguntar por ella. El servicio encargado de gestionar esta configuración y servirla al resto de servicios de la plataforma es **Nutshell API Rest**.

6.2. Requisitos

Analizando la plataforma podemos ver que presenta la necesidad de funcionar en tiempo real y bajo tiempos de respuesta muy bajos. Para gestionar la configuración de esta plataforma encontramos a Nutshell API Rest, que tendrá que ser capaz de cumplir y satisfacer una serie de necesidades y requisitos que vendrán impuestos no solo por la información que guarda sino también por la plataforma a la que pertenece y su funcionamiento en conjunto con el resto de los servicios.

Los requisitos de esta aplicación son:

- Debe ser capaz de proveer la información necesaria de configuración que le sea requerida por el resto de servicios de la plataforma, de forma rápida y consistente, evitando así entorpecer al resto de servicios de la plataforma cuyo funcionamiento es en tiempo real.
- Debe exponer su funcionalidad para crear y gestionar los datos de configuración a través de una API RESTful. Esta API permitirá gestionar la configuración de manera programática y a su vez crear una aplicación web encargada de utilizar esta API para permitir la configuración mediante una interfaz gráfica para usuarios.

- Al ofrecer una API RESTful, también se ha de tener en cuenta que debe cumplir con los estándares de calidad de los servicios y API RESTful que podemos encontrar hoy en día.

- Los datos de configuración deben almacenarse de forma persistente para no perderlos y poder reutilizarlos. Se requerirá de una base de datos para ello.

- Al utilizar una base de datos para almacenar los datos, la aplicación debe ser capaz de responder y no fallar en caso de que exista algún problema al utilizar esta base de datos, adaptando su respuesta a los diferentes problemas que puedan surgir de forma consistente y siempre sin dejar de ofrecer servicio.

- Tendrá que ser capaz de soportar no solo la carga que de solicitudes de los servicios, sino también ofrecer soporte para poder realizar consultas y configuraciones nuevas de manera simultanea sin generar inconsistencias y sin dejar de dar servicio en caso de que existan varias solicitudes de forma paralela.

- Hasta el momento, necesita exponer al menos dos interfaces para acceso de los datos persistidos, una interfaz HTTP y una interfaz para intercambio de mensajes, por lo que se ha de tener en cuenta un diseño modular que permite adaptarse a nuevos posibles mecanismos de comunicación.

Para solventar la problemática que presenta esta aplicación, vamos a poner en juego las tecnologías y conocimientos que hemos tratado en los capítulos anteriores, viendo como en conjunto nos permiten concluir en un diseño y conjunto de buenas prácticas que nos permitirán solventar toda la problemática que presentan aplicaciones como Nutshell API Rest.

6.3. Arquitectura software

La primera problemática que vamos a abordar para diseñar Nutshell API Rest va a ser qué arquitectura software es la más adecuada para implementarla. Analizando cada uno de los requisitos iremos iterando sobre la arquitectura software hasta extraer un diseño que se acomode y cumpla con los requisitos de Nutshell API Rest.

Diseño básico

Analizando los requisitos de Nutshell API Rest vemos que a simple vista la principal necesidad de esta aplicación es funcionar como una API RESTful para la gestión y configuración de la plataforma. Por una lado necesitará exponer una interfaz HTTP a través de la cuál se realizará toda la gestión de las configuraciones. Toda esta información de configuración que se le transmita, debe ser almacenada. En este caso se ha escogido una base de datos relacional (MySQL) que permite crear un modelo relacional que represente adecuadamente los datos de configuración.

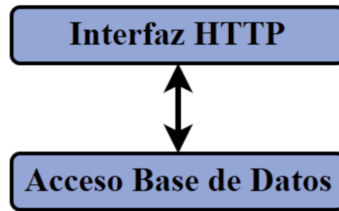


Figura 6.2: Diseño arquitectónico básico.

Diseño con núcleo de lógica propia

Además de ofrecer una interfaz con HTTP para la entrada / salida de datos, y poder persistir estos datos en una base de datos relacional, existe la necesidad de aplicar cierta lógica sobre los datos que se reciben para aplicar las restricciones y definiciones propias del modelo de la aplicación. Estas restricciones no están relacionadas con la entrada de datos por HTTP, pues son restricciones y definiciones independientes de la tecnología que se utilice. Por otro lado, también es buena práctica no acoplar estas restricciones a la tecnología que se ha escogido para almacenar los datos dado que esta podría cambiar en el futuro.

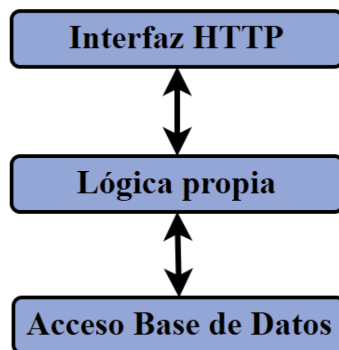


Figura 6.3: Diseño arquitectónico con lógica propia de negocio.

Diseño con múltiples interfaces entrada / salida

Un factor que todavía no se ha tenido en cuenta en el diseño es la necesidad de exponer este servicio no solo mediante una interfaz HTTP para gestión de las configuraciones, sino que también es necesario que este servicio sea capaz de proveer la información de configuración a los servicios que así lo requieran en la plataforma de la que forma parte. Dicha plataforma contiene un mecanismo para comunicación de servicios mediante mensajes, por lo que además de ofrecer una interfaz HTTP, Nutshell API Rest necesitará disponer también de una interfaz que sea capaz de recibir y mandar mensajes a través de la plataforma.

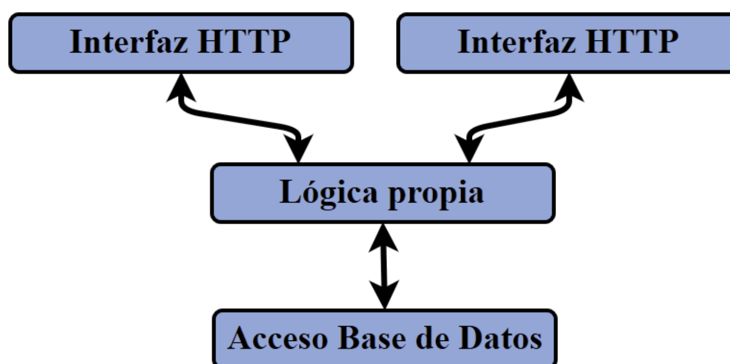


Figura 6.4: Diseño arquitectónico múltiples interfaces de entrada / salida.

6.3.1. Arquitectura de *Ports* y *Adapters*

Si observamos el patrón arquitectónico que nos resulta de hacer un análisis de los requisitos que necesita cubrir Nutshell API Rest, podemos concluir en términos generales que:

- Necesita varias interfaces de entrada / salida de datos para ofrecer los datos sobre configuración de la plataforma en diferentes formatos según el mecanismo de comunicación que se utilice.

- Se requiere un almacenamiento de forma persistente de los datos de configuración, en este caso una base de datos relacional (MySQL).

- Existen una serie de reglas y restricciones que hay que aplicar sobre los datos de configuración que se quieren almacenar. Estas reglas forman parte de la definición base de la aplicación y del modelo propio del negocio, independientes de las tecnologías que se utilicen tanto para almacenar dicha información como para comunicarla.

- Al disponer de lógica propia independiente los mecanismos empleados, está lógica se aplicará también sobre un modelo de datos que debe ser agnóstico de dichos mecanismos. Este modelo propio de la aplicación será transformado y adecuado por cada mecanismo según las necesidades que tenga, además de permitir que la definición de las reglas y restricciones pueda seguir aplicándose independientemente de las tecnologías que se utilicen para almacenar o compartir las configuraciones.

Todas estas necesidades se ven cubiertas en una arquitectura de *ports* y *adapters* (capítulo 3). Esta arquitectura *software* nos permite dar modularidad a Nutshell API Rest en una serie de componentes que a su vez dotarán a la aplicación de la flexibilidad de tecnologías e interfaces que necesita, así como la separación correcta entre los diferentes dominios de cada componente, acoplándolos solo por donde es correcto. A continuación veremos como quedaría un diseño final con está arquitectura en concreto.

Podemos identificar como núcleo de la aplicación el conjunto de restricciones y validaciones que hay que aplicar sobre los datos de configuración. Todas estas características pueden modelarse en un módulo que recibe los datos y peticiones y aplica sobre ellos las restricciones y validaciones. Además, como ya hemos visto, estas validaciones representan la lógica propia de la aplicación inherente en la definición del negocio de Gennion, independiente de los mecanismos de entrada / salida o de almacenamiento. Representa la esencia de la aplicación independientemente de los mecanismos que tenga que utilizar, y debe ser agnóstica de los cambios de tecnologías que existan. Por tanto este núcleo será el encargado de definir toda la lógica de comprobación y validación y definir las interfaces de interacción con los diferentes componentes, que se ajustarán a ellas.

Hemos visto también que se requiere una interfaz HTTP para realizar la entrada de datos y gestión de la configuración a través de una aplicación web que explotará esa API. Si queremos que nuestra lógica de negocio definida en el núcleo pueda recibir esa información necesitaremos dos piezas nuevas:

- Un *port* primario (sección 3.1.1) que define una interfaz de entrada de información al núcleo para gestionar las configuraciones.
- Un *adapter* primario (sección 3.2.1) que implementará la API RESTful, y se encargará de utilizar la interfaz que le ofrece el núcleo para mandarle la información e interactuar con él. Además de utilizar esta interfaz, también será el encargado de transformar la información que le llega en su propio dominio al dominio de la aplicación definido en el núcleo. Y viceversa a la hora de devolver la información a través de la API RESTful.

Además de la interfaz HTTP, también requiere de una interfaz para conectarse al sistema de paso de mensajes de la plataforma y poder comunicarse con el resto de los servicios. Esto podemos hacerlo de la misma manera que hemos hecho con la interfaz HTTP, dado que la arquitectura hexagonal provoca un diseño modular que permite anexionar al núcleo de nuestra aplicación tantas interfaces como sean necesarias. Para ellos, al igual que en el componente HTTP, necesitaremos definir dos piezas nuevas:

- Un *port* primario (sección 3.1.1) que define la interfaz que expondrá el núcleo de la aplicación al *adapter* que se conecte.
- Un *adapter* primario (sección 3.2.1) que utilizará la interfaz definida en el núcleo para, una vez conseguida la conexión al sistema de paso de mensajes, atender y responder a todas las peticiones que le lleguen de los servicios de plataforma acerca de la información que almacena. Además también será el encargado de transformar la información del dominio de la aplicación al dominio de los mensajes, y viceversa.

Con estas tres piezas conectadas como en la figura 6.5, la aplicación estaría preparada para interactuar tanto con la plataforma como con la aplicación web encargada de usar su API RESTful.

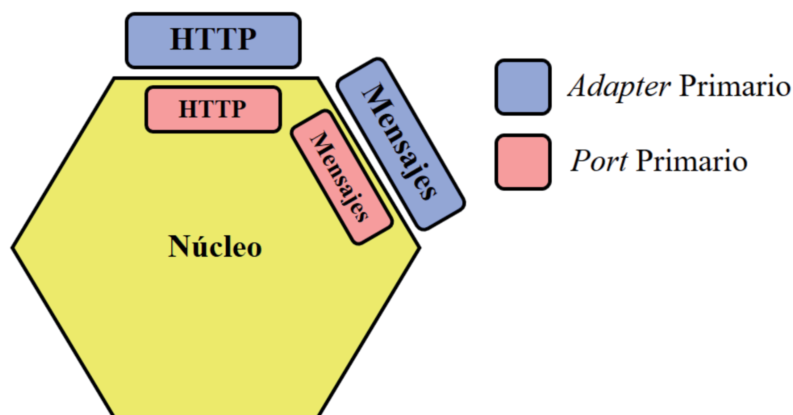


Figura 6.5: Arquitectura hexagonal con entrada entrada / salida.

Una vez tenemos preparada la entrada de datos a nuestra aplicación, el siguiente requisito a tratar es el almacenamiento de forma persistente de la información en una base de datos. La aplicación como necesidad propia solo plantea el requisito de un mecanismo que le permita almacenar de forma persistente la información de la que dispone. Esta parte de la definición irá dentro del propio núcleo, y será otro componente el encargado de implementar y dar soporte a ese mecanismo que ha definido el núcleo, almacenando en este caso en una base de datos relacional (MySQL). Necesitaremos para hacer esto dos piezas nuevas:

- Un *port* secundario (sección 3.1.2) encargado de definir la interfaz que utilizará el núcleo de la aplicación para realizar las operaciones de almacenamiento.
- Un *adapter* secundario (sección 3.2.2) encargado de implementar y dar soporte a esa interfaz que ha definido el núcleo. Para ello tendrá que implementar los métodos definidos en el *port* secundario, transformando los datos que recibe al modelo específico de la base de datos que se está utilizando.

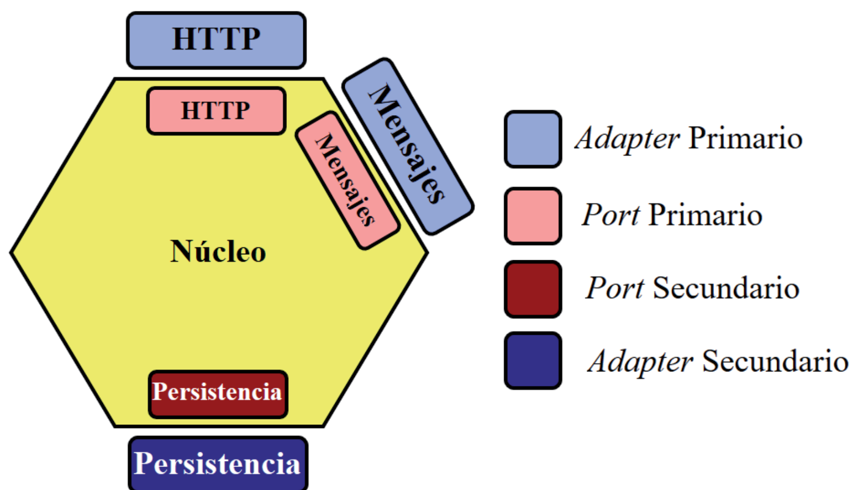


Figura 6.6: Arquitectura hexagonal de Nutshell API Rest.

El diseño final de Nutshell API Rest tendría entonces dos *adapters* primarios para la entrada / salida de información de la aplicación, y un único *adapter* secundario para implementar el soporte para almacenarla. La figura 6.6 muestra el resultado final de este diseño.

6.4. Implementación

En la sección anterior hemos definido la arquitectura software más adecuada para que Nutshell API Rest sea capaz de adaptarse y funcionar dentro de la plataforma en la que se encuentra. Pero además de establecer cómo debe estructurarse esta aplicación, hay que dar soporte a esta estructura e implementar con tecnologías que no sólo nos permitan respetar y seguir este diseño, sino también cubrir y satisfacer el resto de requisitos de Nutshell API Rest. Hasta ahora solo se han abarcado los requisitos referentes a las necesidades de como se han de poder servir y tratar los datos, quedando todavía por resolver:

- Proporcionar la información de configuración al resto de servicios de la plataforma en tiempos de respuesta rápidos.
- La interfaz HTTP debe tener unos tiempos de respuesta adecuados y dentro de los estándares actuales.
- Dada la importancia de los datos que gestiona la aplicación para otros servicios, debe ser capaz de funcionar bajo cualquier situación de carga de trabajo.
- Debe ser capaz de funcionar en caso de que se produzcan errores o fallos inesperados, pudiendo actuar adecuadamente sin dejar de ofrecer el servicio.

Observando los requisitos que quedan por abarcar y solucionar, se desprende que la aplicación necesita cumplir tres propiedades:

- La aplicación debe ser capaz de funcionar bajo casos de error y fallo sin dejar de ofrecer su servicio y responder de forma consistente, para evitar paralizar y entorpecer al resto de servicios de la plataforma, es decir, debe ser resiliente.
- La aplicación debe funcionar bajo diferentes niveles de carga, desde pocas peticiones HTTP por minuto a varios miles de mensajes por segundo. Esto quiere decir que la aplicación debe ser elástica.
- La aplicación debe ser capaz de ofrecer unos tiempos de respuesta adecuados y rápidos, dentro de unos límites establecidos por la plataforma, es decir, debe ser *responsive*.

Si estás tres propiedades consiguen satisfacerse en la implementación de Nutshell API Rest, la aplicación cumplirá con los requisitos. Estás tres propiedades las podemos localizar dentro del manifiesto de Sistemas Reactivos [1], que establece las características que

tiene que cumplir un sistema reactivo (capítulo 2). Añadidas a estas tres se recomienda hacer un sistema orientado a mensajes, para así favorecer el desacoplamiento, aislamiento y transparencia de ubicación entre los componentes internos de la aplicación. Es por esto que podemos considerar que Nutshell API Rest debe implementar siguiendo los estándares y recomendaciones de un sistema reactivo.

A continuación el uso de la concurrencia con actores, así como el uso de futuros para añadir asincronía nos permite dar soporte tecnológico a todas estás características y requisitos.

6.4.1. Actores

Como se trató en la sección 5.2, los actores representan una unidad de concurrencia en la que existe un estado compartido pero solo accesible y modificable por una única instancia del propio actor y no el resto de actores. Además, la metodología de comunicación que usan los actores es a través de mensajes. El uso de mensajes para la comunicación nos permite toda la comunicación entre los diferentes componentes de la aplicación, utilizando los actores como entidades que van a encapsular cada uno de los componentes y van a comunicarse entre ellas.

La primera aproximación del uso de actores que podemos hacer es encapsular cada componente que requiere la aplicación en actores, de manera que la comunicación entre componentes se traduzca en un mero paso de mensajes entre ellos, como en la figura 6.7.

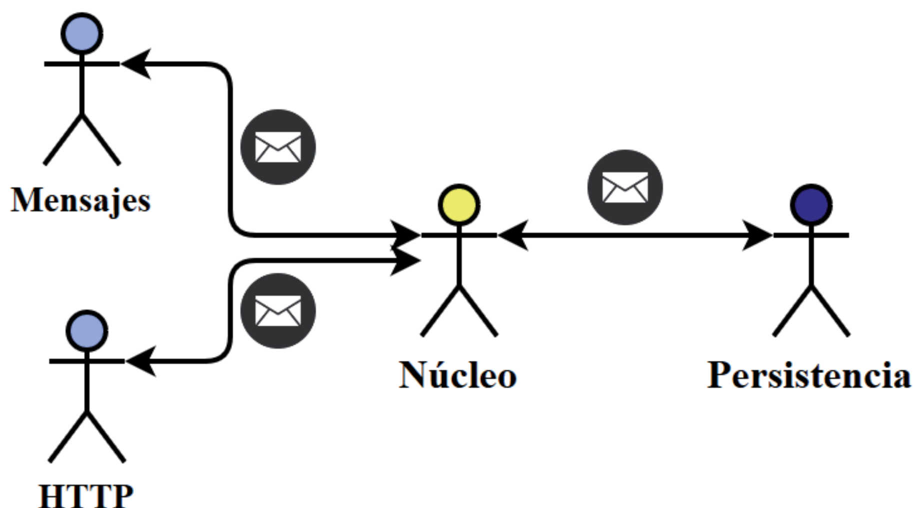


Figura 6.7: Actores en Nutshell API Rest.

Cada componente queda encapsulado en un sistema de actores. Este sistema de actores puede estar formado por un único actor si dicho componente no tiene una alta carga de trabajo. Hay que tener en cuenta que aunque exista un único actor, dicho actor puede tener varias instancias de él mismo, solo que todas sus instancias comparten un único buzón de mensajes que es tratado de forma atómica y de uno en uno. Esto permite dar

cierta elasticidad y escalabilidad a la aplicación si se usa con otros mecanismos como los futuros.

El sistema de actores también puede estar compuesto por varios actores distintos, que a su vez pueden tener varias instancias de ellos mismo compartiendo un único buzón. Utilizando un sistema de actores con varios actores distintos podemos conseguir que dentro de un componente existan diferentes piezas que realicen diferentes operaciones de forma paralela. Con un sistema así podemos conseguir una gran escalabilidad y elasticidad para la aplicación.

Utilizando así los actores y los sistemas de actores en la aplicación, podemos dotar a cada componente de la elasticidad y escalabilidad que necesita, y por lo tanto otorgar a Nutshell API Rest de la escalabilidad y elasticidad que requiere, ya que un sistema de actores con varios actores puede encargarse de dar soporte a la gran carga de mensajes que va a tener que tratar de las peticiones de la plataforma, mientras que con un sistema de actores de un único actor puede atender y dar soporte a la interfaz HTTP cuya carga es mucho menor.

6.4.2. Fallos en actores: *Let it crash!*

Los principales mecanismos para manejo de errores en un aplicación son la captura y tratamiento de los posibles errores (excepciones) que pueden ocurrir. El problema de este tipo de estrategias es que no permite construir un sistema totalmente tolerante a fallos.

En cada sistema de actores existe un actor que realiza la función de supervisor, que conoce y puede comunicarse con el resto de actores del sistema. Dado que cada actor ejecuta de forma independiente al resto de actores de su sistema, los errores que ocurran dentro de cada actor quedan aislados dentro del mismo. Este fallo puede comunicarse al actor supervisor que aplicará la política que tenga definida en caso de fallo de un actor.

La filosofía *Let it crash!* se basa en usar este aislamiento de los actores para establecer la siguiente política: si se produce un fallo, dejar que dicho actor falle y finalizar su ejecución. El supervisor se encargará de restaurarlo si así está definido en su política.

Existen también mecanismos para salvar el histórico de cada uno de los actores, de manera que cuando sea necesaria su restauración, es posible reconstruir su estado antes del fallo, y así reanudar su ejecución en momentos antes del fallo.

Vemos que utilizando los actores la aplicación adquiere la propiedad de ser resiliente, pues cada componente puede fallar de forma independiente, pero esto no provoca un fallo en cadena de cada uno de los componentes y por lo tanto evita que se produzca un fallo total de la aplicación.

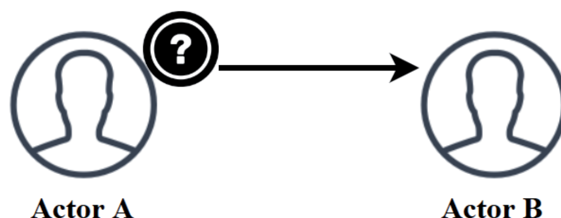
6.4.3. Comunicación entre actores: *Ask Pattern*

La comunicación entre actores es mediante mensajes de forma síncrona o asíncrona. El patrón más inmediato de uso es mensajes entre ellos, sin esperar. El problema de este

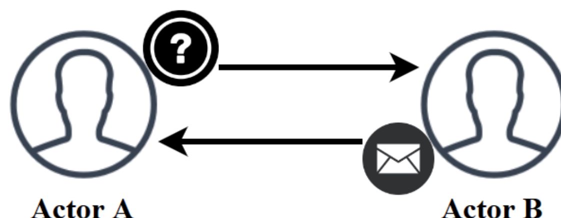
patrón es que un sistema de actores no asegura ni la recepción del mensaje ni el orden de llegada, por lo que no podemos saber si el mensaje se ha recibido correctamente.

Para solucionar este problema, existe un patrón de intercambio de mensajes que establece una comunicación entre dos actores bloqueante para el actor que lanza el mensaje. Este patrón es el *Ask Pattern*. Su funcionamiento es el siguiente:

- Un actor manda un mensaje a otro actor, bloqueando su ejecución a la espera de la respuesta.



- El actor que recibe el mensaje además recibe una referencia al actor que le mandó la ”pregunta”. Tras procesar el mensaje, manda la respuesta al actor que preguntó.



- El actor que mandó la pregunta recibe la respuesta, desbloquea su ejecución y procesa la respuesta.

Para evitar un bloqueo de forma indefinida, las preguntas que bloquean al actor tienen que tener configurado un parámetro que establezca un tiempo de espera máximo, y en caso de no recibir respuesta, lanzar un aviso que desbloquea el actor para que continúe ejecutando nuevos mensajes ya que el que ha mandado no se ha respondido.

Este patrón nos permite asegurar que la comunicación entre los actores se produce de forma normal y correcta. A su vez, ante un caso de fallo, la aplicación adquiere la capacidad de detectar el error de comunicación y comportarse de forma resiliente, de manera que puede volver a preguntar o responder de forma consistente ante estos fallos de comunicación entre actores. Esto puede aplicarse tanto a nivel de módulos de la aplicación como dentro de los propios sistemas de actores de cada componente.

Futuros

Hemos visto que con el patrón *Ask Pattern* podemos asegurar la comunicación entre actores, y por lo tanto entre módulos de la aplicación, y a su vez hacer más resiliente

la aplicación. Pero al usarlo provocamos un efecto indeseado en nuestra aplicación: la comunicación entre los actores se vuelve síncrona, dado que el actor se bloquea a la espera de una respuesta.

Esta sincronía puede no suponer un problema si la carga de la aplicación no es excesivamente alta, pero en situaciones de gran carga, será necesario eliminar esta sincronía a la hora de tratar las peticiones y realizar el trabajo de forma concurrente y asíncrona siempre que sea posible. Como hemos visto, los actores aportan concurrencia a la aplicación. El problema lo encontramos en que si las operaciones que se realizan cada vez que se recibe un mensaje se realizan de forma síncrona, cada vez que se use un *Ask Pattern* se producirá un bloqueo a la espera de que el actor que tiene que mandar la respuesta haga la ejecución de todas las operaciones. Esto además puede provocar que si esas operaciones tienen una duración un poco más larga de lo normal sobrepasen el tiempo de espera del actor, a pesar de que se está procesando correctamente la petición.

Para solventar esta problemática encontramos soluciones como los *futuros* (capítulo 5.5). Los *futuros* nos permiten realizar todas operaciones en un hilo totalmente separado al propio actor, y consultar su resultado cuando lo necesitemos. Veamos como podemos usar estos los *futuros* para solucionar la problemática del bloqueo en el *Ask Pattern*.

Un *futuro* se define como una serie de operaciones que se ejecutan en un hilo distinto al hilo principal. Para poder trabajar con ellos conjuntamente con los actores hay que cumplir los siguientes requisitos:

- Ser capaces de componer operaciones sobre los *futuros* para que se apliquen cuando se ejecuten.
- Los actores deben ser capaces de comunicarse los *futuros* como mensajes.

Veamos en un ejemplo de petición HTTP como los actores conjuntamente con los *futuros* sirven para solucionar esta problemática. Al recibir la petición HTTP, el actor del módulo HTTP lanza un mensaje al actor del núcleo preguntando por el elemento que indique la petición. A su vez, el actor del núcleo lanza la consulta en forma de pregunta a la base de datos. El actor de la base de datos lanzará una operación de lectura contra la base de datos, y en lugar de bloquearse a la espera del resultado, encapsula esta operación en un *futuro*, de manera que no tiene que esperar a su resultado. Este *futuro* se manda al actor del núcleo como respuesta a su pregunta.

El actor del núcleo añade operaciones a este *futuro* que se realizarán sobre el dato que devuelve la operación que le ha retornado el actor de la base de datos en un *futuro*. Como estas operaciones son simplemente una concatenación a la operación de la base de datos, no se ejecutan en ese momento, sino que se añaden al *futuro*. El actor del núcleo retornará este nuevo *futuro* al actor HTTP.

El actor HTTP será el encargado de esperar al valor de retorno del *futuro*. Para ello, al igual que ha hecho el actor del núcleo, toma el mensaje que ha recibido del núcleo, añade a ese las operaciones que transforman el resultado en una respuesta HTTP. Con toda esta composición, el *futuro* resultante no ha evaluado nada todavía pero tiene todas las operaciones necesarias para dar una respuesta HTTP.

Para finalizar, el actor HTTP pone el *futuro* en ejecución en un hilo separado, que generará la respuesta una vez haya terminado de ejecutar, y permitirá a su vez al actor seguir ejecutando y tratar nuevos mensajes sin bloquearse.

6.4.4. Diseño final

Hasta ahora hemos visto que se puede implementar una aplicación que siga los principio del manifiesto de sistemas reactivo utilizando una arquitectura hexagonal. Está arquitectura se puede implementar usando los actores de forma conjunta con los *futuros*. Utilizando todos estos conceptos se puede abarcar todos los requisitos que tiene que cumplir Nutshell API Rest. A continuación veremos el diseño final de todos estos elementos de forma conjunta.

Ya hemos establecido que cada componente estará encapsulado en un actor. Veamos el diseño con la interfaz HTTP de la aplicación. El actor HTTP se encargará en un principio del *adapter* primario de HTTP. Este *adapter*, al recibir las peticiones, traducirá la información y llamará a las funciones definidas en el *port* HTTP dentro del núcleo de la aplicación. Será esta pieza, el *port*, el encargado de traducir esas llamadas que recibe del *adapter* en mensajes para el actor encargado del núcleo de la aplicación, quedando el actor HTTP como en la figura 6.8

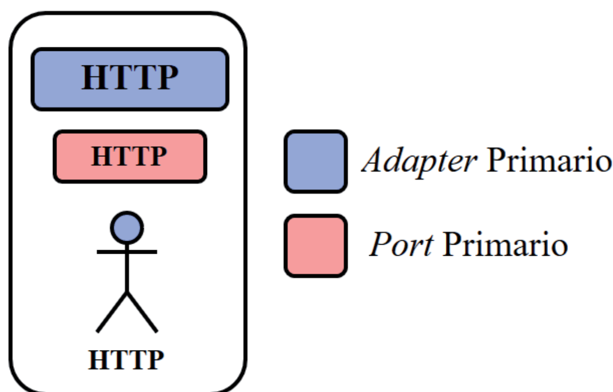


Figura 6.8: Actor encargado del componente HTTP.

El actor del núcleo será el receptos de estos mensajes y aplicar la lógica que tiene definida para cada caso a la información que le llega. El núcleo además definirá la interfaz de la que hará uso para realizar las operaciones de almacenado de la información. Esta interfaz se encuentra definida en el *port* secundario de almacenamiento.

El encargado de implementar esta interfaz o *port* será el componente encargado de almacenar de forma persistente la información. Este componente es el *adapter* secundario de almacenamiento. Este adapter será el encargado de traducir esas llamadas a la interfaz de almacenamiento, es decir, el port, a a mensajes al actor encargado de realizar las operaciones de almacenamiento contra una base de datos.

Estos mensajes serán recibidos por el actor del almacenamiento de forma persistente,

que realizará la consiguiente traducción de los mensajes a los diferentes métodos que interactúan contra la base de datos. Es por esto que podemos ver que en el *adapater* secundario de almacenamiento encontramos por un lado una sección que pertenece al actor del núcleo de nuestra aplicación y por otro lado el resto de la implementación que es la utilizada por el actor de almacenamiento.

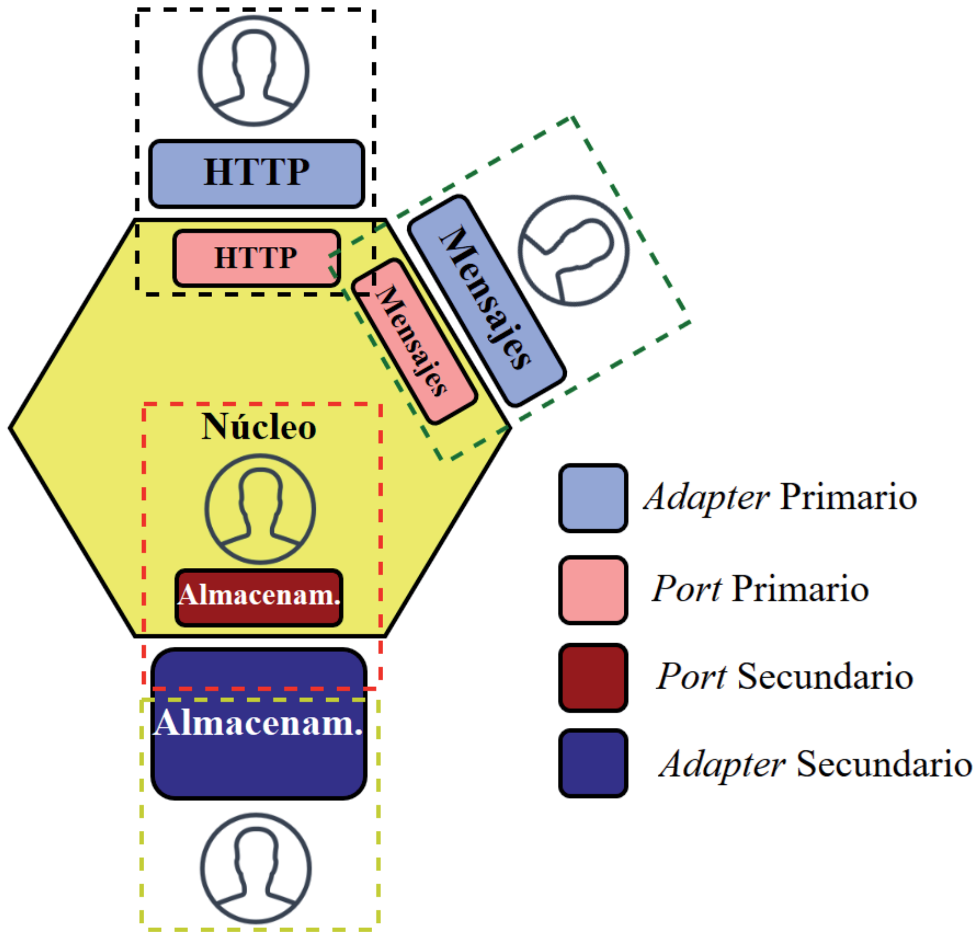


Figura 6.9: Diseño final con actores.

Capítulo 7

Recomendaciones

A continuación las recomendaciones y prácticas que podemos extraer de este desarrollo de Nutshell API Rest para el desarrollo de aplicaciones que requieren un diseño que ofrezcan escalabilidad funcional.

- Uno de los primero factores importantes a tener en cuenta es la arquitectura software que se ha de utilizar. La arquitectura hexagonal nos ofrece una serie de características como:

    - Definir un modelo propio de la aplicación así como una lógica que define el propósito y esencia de la aplicación. Todo esto lo encontramos definido en el núcleo de la aplicación.
    - Definir una interfaz para las posibles entradas de datos que va a tener nuestra aplicación, y conectar a esa interfaz un componente que puede poseer su propio dominio y sabe como utilizar ese mecanismo de entrada de información y usarlo para meter información al núcleo de la aplicación.
    - Definir una interfaz para la salida de información del núcleo de la aplicación, y al igual que con la entrada de información, poder conectar un componente que implemente esa interfaz y reciba esta información para hacer con ella lo que necesite, como almacenarla o trasmitirla a otro lado.

  Esta separación ofrece una estructura bien definida de los componentes de la aplicación, ofreciendo también una escalabilidad funcional de manera muy sencilla, ya que solo hace falta crear o reutilizar las interfaces que define el núcleo para añadir nuevos componentes.

  También permite que la aplicación pueda cambiar su comportamiento interno, y mientras no modifique sus interfaces, los demás componentes de entrada salida seguirán funcionando. Además, los componentes que se conectan a la interfaz puede sustituirse por otros ante necesidad sin que el propósito de la aplicación se vea afectado. Esto demuestra que la arquitectura promueve y facilita el diseño de la aplicación de forma modular.

- Seguir las recomendaciones de un sistema reactivo nos guía para crear una aplicación que no solo se adapta a los requisitos de las aplicaciones, sino que será más robusta y podrá soportar más carga.

- Utilizar los actores como mecanismo de concurrencia para la aplicación. Nos permiten cubrir las necesidades de una aplicación reactiva y a su vez implementar una arquitectura de *ports* y *adapters*. Cada componente de la arquitectura se encapsula dentro de un sistema de actores, que a su vez puede estar implementado por uno o varios actores.

  Este razonamiento aporta escalabilidad a la aplicación, pues cada parte de la aplicación puede escalar en el número de actores que necesita según su carga, incluso manejarlo de forma dinámica para adaptarse. De esta manera conseguimos concurrencia dentro de cada componente.

  Aislando cada componente en su propio sistema de actores, aislamos al resto de componentes de sus fallos, proporcionando resiliencia a la aplicación, haciéndola tolerante a fallos.

  Los actores, al comunicarse por mensajes, pueden hacerlo de forma síncrona o asíncrona. Utilizando patrón *Ask Pattern* podemos asegurar la comunicación entre los componentes y en caso de fallo, dar una respuesta consistente, haciendo que la aplicación más resiliente.

- Utilizar futuros para componer las operaciones que se quieren realizar para ponerlo en ejecución en un hilo paralelo. Esto permite no bloquear los actores en el *Ask Pattern*, ya que los actores componen las acciones y luego uno de ellos pone en ejecución el futuro. Además, al ejecutar en un hilo separado, todos los actores pueden seguir trabajando atendiendo otras peticiones de forma concurrente.

Capítulo 8

Conclusiones

El desarrollo nos demuestra que la concurrencia es un elemento muy importante cuando hablamos de aplicaciones que tienen que ser reactivas para las necesidades de los usuarios.

Utilizando la concurrencia conjuntamente con un diseño reactivo podemos conseguir aplicaciones que ofrecen una disponibilidad cercana al 100 %, tolerante a fallos y escalable ante mucha carga de trabajo. Pero como hemos visto, un diseño que cumpla estás característica termina necesitando otras propiedades como:

- Modularidad: la aplicación tiene que estar bien definida en componentes bien diferenciados que no solo tienen su propia lógica, sino que manejan un modelo propio que es capaz de traducir al modelo de la núcleo.
- Acoplamiento: a pesar de que la aplicación está dividida en módulos, tiene que existir un acoplamiento que conecte y comunique todos estos módulos. El acoplamiento en parte lo da el núcleo de la aplicación, el cual conocen todos los módulos su dominio. Además existe un acoplamiento por la interfaz que define el núcleo, ya que un cambio en dicha interfaz produce un cambio en todos los módulos que hagan uso de la misma.

La modularidad de la aplicación se ha conseguido no solo mediante un desarrollo siguiendo las guías de la arquitectura. El uso de técnicas de testeo, en concreto el desarrollo guiado por los tests, han sido cruciales para el desarrollo de este diseño. El uso de está técnica permitía realizar correctamente la definición de cada módulo, ya que cada uno debía poder testearse de forma independiente. Si el diseño era incorrecto, los test resaltan este error al no poderse realizar de los tests de forma unitaria para cada uno de los componentes. Esto ha provocado que el desarrollo y diseño de la aplicación no haya sido un proceso de una sola vez, sino todo lo contrario, ha sido un proceso continuo, iterando sobre el diseño para comprobar posible errores y aplicar las correcciones.

A pesar de todas la ventajas que presenta un diseño y desarrollo con las directrices de una aplicación reactiva con arquitectura hexagonal, esto añade una complejidad y un coste de desarrollo más elevado que si se utilizaran arquitecturas más convencionales y sencillas

como podría ser un monolito de aplicación que no fuera modular. Esta complejidad la encontramos al precisar de componentes bien diferenciados y modulares, la utilización de actores como tecnología con toda su complejidad inherente, así como todas las pruebas necesarias para probar cada módulo por su parte y las pruebas de integración para probar la aplicación.

Otra problema complejo a solucionar son las bases de datos. Todo sistema de almacenamiento tiene que enfrentarse al Teorema de Brewer [4]. Esto se traduce en que no se puede garantizar la consistencia y la disponibilidad al mismo tiempo si se quiere que exista tolerancia al particionado, es decir, que el sistema siga funcionando aunque haya un fallo de red. Como resultado de esto las aplicaciones tiene que o bien utilizar un sistema de almacenamiento que cumpla las propiedades ACID, aceptando que puede existir tiempos de no disponibilidad temporalmente, o bien elegir consistencia eventual y hacer que toda la lógica de la aplicación resiliente a posibles perdidas de datos.

Conclusiones personales

Este trabajo se ha realizado en un entorno de empresa real, Gennion Solutions. El desarrollo de este trabajo en un entorno profesional me ha permitido no solo enfrentarme a una gran cantidad de nuevos conceptos que hasta ahora me eran desconocidos, sino también ponerles en práctica en casos reales que tienen una aplicación inmediata en la empresa.

También quiero de destacar la labor realizada por mis compañeros de trabajo que han sido los que más han participado en mi formación, y han permitido que adquiera las bases para llevar a cabo este trabajo.

Aunque el trabajo se plantea como una sucesión de conceptos y aplicaciones de los mismos, el proceso de desarrollo de un diseño como el presentado ha sido un trabajo iterativo de forma constante. Se comenzó un un diseño mucho más trivial y sencillo, y con el desarrollo y definición de nuevos requisitos se fue perfeccionando hasta alcanzar el que se expone.

Bibliografía

[1] Reactive Manifesto. [Online] Available: http://www.reactivemanifesto.org/.

[2] B. Cantrill and J. Bonwick, "Real-world concurrency," *Queue*, vol. 6, pp. 16–25, Sept. 2008.

[3] peter Van Roy and S. Hafidi, *Concepts, Techniques and Models of Computer Programming*. PHI Learning Private Limited, 2009.

[4] J. Browne, "Brewer's cap theorem," *J. Browne blog*, 2009.

[5] Ports-And-Adapters / Hexagonal Architecture. [Online] Available: http://www.dossier-andreas.net/software_architecture/ports_and_adapters.html.

[6] Hexagonal Architecture. [Online] Available: http://alistair.cockburn.us/Hexagonal+architecture.

[7] Don't Ask, Tell. [Online] Available: http://bartoszsypytkowski.com/dont-ask-tell-2/.

[8] Akka Ask Pattern. [Online] Available: https://www.trivento.io/akka-ask-pattern/.

[9] The Clean Architecture. [Online] Available: https://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[10] J. Gray and A. Reuter, *Transaction processing: concepts and techniques*. Elsevier, 1992.

[11] J. Gray *et al.*, "The transaction concept: Virtues and limitations," in *VLDB*, vol. 81, pp. 144–154, 1981.

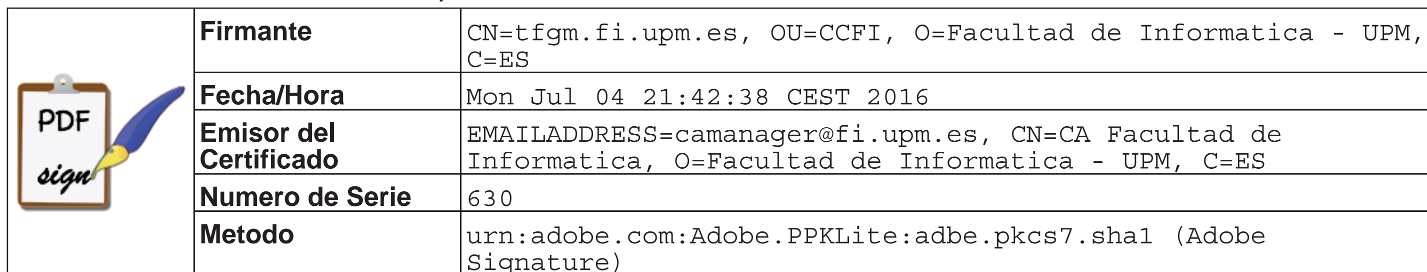
[12] P. M. Lewis, A. J. Bernstein, and M. Kifer, *Databases and transaction processing: an application-oriented approach*. Addison-Wesley, 2002.

[13] D. Wyatt, *Akka Concurrency*. USA: Artima Incorporation, 2013.

[14] S. Freeman and N. Pryce, *Growing object-oriented software, guided by tests*. Pearson Education, 2009.

[15] B. Erb, “Concurrent programming for scalable web architectures,” diploma thesis, Institute of Distributed Systems, Ulm University, April 2012.

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Mon Jul 04 21:42:38 CEST 2016
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)