UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Grado en Ingeniería Informática

Implementación de un Framework de Programación para el Desarrollo de Agentes Recolectores de Datos sobre Contenedores

Implementation of Software Framework for Developing Data Collector Agents Running in Containers

Realizado por
Juan José Trujillo Bueno

Tutorizado por
Daniel García Morán
Eduardo Guzmán de los Riscos

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

Málaga, septiembre de 2021

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

Implementación de un Framework de Programación para el Desarrollo de Agentes Recolectores de Datos sobre Contenedores

Implementation of Software Framework for Developing Data Collector Agents Running in Containers

Realizado por
Juan José Trujillo Bueno

Tutorizado por
Daniel García Morán
Eduardo Guzmán de los Riscos

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2021

Fecha defensa: octubre de 2021

Resumen

Hoy en día, las empresas con millones de clientes interactuando con sus servicios en todo el mundo a través de diferentes dispositivos, tienen la necesidad de explotar la enorme cantidad de datos que estas interacciones generan. Para ello acuden a empresas como Devo, que ofrece un servicio de almacenamiento y análisis de datos en tiempo real nativo en la nube.

Esta inmensa producción de datos se convierte en un reto desde el punto de vista del desarrollo software, obligando a construir soluciones que escalen sosteniblemente. Este trabajo presenta Boson, un framework de desarrollo cuyo objetivo es facilitar a Devo la creación, de forma escalable, de agentes de transmisión de datos para múltiples protocolos y formatos. Se describe el proceso de diseño y desarrollo del framework en Java, el cual permitirá que los desarrolladores de agentes puedan centrarse en la lógica de procesamiento sin necesidad de preocuparse de cómo se implementa la capa de transporte de datos. Durante el proceso se analiza el uso de ciertos patrones y principios de programación orientada a objetos que hacen que el framework sea extensible y fácil de usar.

Finalmente, se mostrará cómo usar el framework para implementar agentes. Desarrollaremos dos agentes que implementarán el protocolo Time y veremos cómo facilita la reutilización de código a través de los filtros de procesamiento.

Este TFG se ha realizado en colaboración con la empresa Devo, en el marco del programa Impulso TFE de la UMA.

Palabras clave: Framework, Java, Transmisión de Datos

Abstract

Nowadays, companies have millions of clients that use their services in different devices around the world and have the necessity of getting insights from the huge amount of data generated by their interactions. This is the reason that led them to require companies as Devo, that offers a cloud-native storage and real-time analytics service.

This immense data production is a challenge from the software development perspective, forcing to build solutions that scale sustainably. This project introduces Boson, a development framework whose objective is to assist Devo in the creation of data transmission agents, in scalable way, for multiple protocols and formats. Along this text, we describe the framework design and development process in Java language. Using this tool will allow developers to be focused on the data processing logic by forgetting about how the transport layer is implemented. Along the process we have analyzed the use of patterns and object oriented programming principles that make the framework extensible and easy to use.

Finally, it will be shown how to use the framework to implement agents. We will develop two agents that implement the Time protocol. Lastly, we will examine how this tool facilitates reusing code through processing filters.

This final degree project has been a collaboration between Devo and UMA, under the Impulso TFE program.

Keywords: Framework, Java, Data Streaming

Índice

Índice de Figuras

Índice de Tablas

1

Introducción

1.1 Motivación

Hoy en día tenemos una gran variedad de dispositivos digitales que generan datos de diversa índole, ya sea por tipo, formato, cadencia de generación, protocolo, etc. Los servicios SaaS (Software as a Service) de almacenamiento y análisis de datos en la nube como Devo [1] necesitan herramientas para poder procesar esa variedad de fuentes y poder reutilizar sus piezas comunes para evitar la repetición de código en el desarrollo, además de facilitar el despliegue a demanda de software en la nube.

La herramienta que usa Devo para la ingesta de datos es el Relay [2]. Esta es una aplicación de transmisión de datos de alto rendimiento que se instala en la infraestructura interna de los clientes y recibe eventos de entrada desde las fuentes, aplica unas reglas de procesamiento, los etiqueta y los envía a la plataforma de datos de Devo de forma segura mediante TLS (Transport Layer Security), como se observa en la Figura 1.

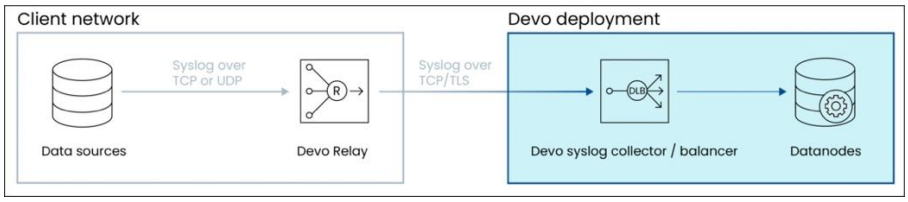

Figura 1. Arquitectura de ingesta de datos de Devo a través del Relay [3].

El Relay durante años ha cumplido con las exigencias de clientes que necesitaban enviar sus datos y podían hacerlo mediante el protocolo Syslog (System Logging Protocol) [4]. Para permitir el soporte de otros protocolos se crearon agentes específicos que se instalan junto al Relay y se encargan de transformar los datos a Syslog y reenviarlos al Relay, pero el desarrollo de estos agentes no permite la reutilización de código, ni provee capacidades de gestión de despliegue automático.

En la actualidad los clientes demandan nuevas funcionalidades para soportar diferentes protocolos y formatos de datos y no basta con los agentes actuales, lo que obliga al estudio de soluciones para hacer frente a las nuevas exigencias.

Además, la tendencia de desplazamiento del software al despliegue nativo en la nube, crea nuevos retos y obliga a abstraer la implementación de aplicaciones teniendo en cuenta que podrían desplegarse de forma tradicional, sobre una máquina servidor, o sobre un contenedor en una infraestructura de orquestación de contenedores.

1.2 Objetivos

El objetivo de este Trabajo de Fin de Grado (TFG) es presentar el proceso de desarrollo llevado a cabo en Devo para crear Boson[1], un framework para el

[1] Partícula elemental de la materia. El nombre tiene su origen en la física de partículas.

desarrollo de agentes de transmisión de datos activos (*pull*) y pasivos (*push*)[2] ejecutados sobre contenedores, cuyo objetivo es solucionar los problemas comunes de la implementación y gestión de agentes, evitando así reinventar la rueda constantemente.

La primera parte se centra en el estudio de la competencia y otros frameworks similares en el mercado para poder completar una base de requisitos para Boson.

A continuación, se detallará el proceso de investigación, diseño y desarrollo del framework, con la finalidad de aportar al lector una idea sobre los pasos seguidos de adquisición de conocimiento y habilidades antes de comenzar a desarrollar código de propósito específico que pueda solventar las necesidades de los agentes, así como la composición de sus piezas usando patrones de diseño de programación orientada a objetos que permita crear código reusable y escalable.

Como objetivo final, se desarrollarán dos agentes para demostrar como podemos usar Boson y se analizará su rendimiento con JDK Mission Control[3].

1.3 Estructura del documento

La estructura del documento se alinea con los objetivos de este trabajo. El Capítulo 2 contiene el estudio del arte donde se investigará cómo soluciona el mismo problema la competencia de Devo y veremos frameworks de desarrollo de aplicaciones similares.

[2] Terminología comúnmente usada en la arquitectura de servicios. En la transmisión de datos, activo significa que va a la fuente a por el dato y pasivo, que espera que el dato le llegue [5].
[3] Herramienta de código abierto que permite monitorizar y optimizar aplicaciones Java [6].

En el Capítulo 3 se presenta el estudio y análisis de funcionalidades que el framework contendrá.

Definidos los requisitos, el Capítulo 4 se centra en el diseño del framework. Primero se explicarán los conocimientos básicos sobre arquitectura de sistemas operativos y Java que serán fundamentales para entender, posteriormente, el análisis de la arquitectura de componentes que permiten la implementación de agentes de transmisión de datos.

Por último, en el Capítulo 5, se usará el framework para la creación de dos agentes, uno activo y otro pasivo, y analizaremos su rendimiento.

1.4 Tecnologías usadas

En este trabajo se ha utilizado Java Open JDK 16 para el desarrollo del framework y los agentes.

Como herramienta de automatización de código para todo el proceso de compilación, ejecución y empaquetado automático, se ha usado Gradle [7].

Para la realización de pruebas y monitorización se ha empleado JUnit 5 y JDK Mission Control. También se ha usado SonarQube [8] para detectar problemas de calidad y seguridad en el código.

2

Estudio del Arte

2.1 Análisis de la Competencia

Para obtener una mejor visión del problema, se ha hecho un análisis en las empresas de la competencia de Devo para ver qué tipo de soluciones ofrecen para la recolección de datos de diferentes fuentes. Entre todas las empresas de la competencia nos centraremos en las dos más importantes Splunk [9] y Elastic [10].

2.1.1 Splunk

Splunk, fundada en 2003, es una empresa con un producto software de almacenamiento y análisis de datos en tiempo real y actual líder del sector.

Al igual que Devo, Splunk ofrece una aplicación de recolección de datos, llamada *Universal Forwarded* [11], que se instala en la infraestructura del cliente. Normalmente se configura para leer los datos de ficheros en las fuentes, pero puede personalizarse a través de *add-ons* [12] que permiten añadir nuevas formas de recolección de datos.

2.1.2 Elastic

Elastic, fundada en 2012, es una compañía holandesa que nació con el objetivo de ofrecer servicios comerciales sobre Elasticsearch [13], un motor de búsqueda distribuido basado en texto. A parte de Elasticsearch, es propietaria de dos productos centrados en la recolección de datos, estos son Logstash [14] y Beats [15].

Logstash es una aplicación que ingesta datos en el lado del cliente y los envía al destino que se configure, típicamente Elasticsearch. Soporta múltiples fuentes e integraciones a través de *plugins*, que al igual que el Universal Forwarded, hacen que sea modular y personalizable. Proporcionan un framework de desarrollo de plugins que son ejecutados por el motor de Logstash [16], es decir, no pueden ser ejecutados independientemente ni en solitario ni en la nube.

Por otro lado, Beats es una plataforma compuesta por agentes ligeros de recolección de datos en el origen. Todos estos agentes se basan en el framework libbeat [17], que es una librería escrita en Go con el objetivo de resolver los problemas comunes en la implementación de estos agentes, similar al framework desarrollado en este trabajo.

2.2 Frameworks Similares Open Source

En el mundo del código libre podemos encontrar algunas librerías de propósito general que permiten crear aplicaciones de transmisión de datos. El diseño de Boson comparte ideas en las que se basan estos frameworks, pero la principal diferencia es la gran comunidad que poseen, lo que hace que el código sea mucho más robusto y depurado. Entre ellos podemos encontrar:

- **Netty**: es un framework que permite crear aplicaciones de red altamente concurrentes [18]. Usa un API unificado para el uso de sockets bloqueantes y no bloqueantes (ver apartado 4.3.2), además de optimizar el uso de memoria. La idea de su creación fue mejorar la librería estándar de Java NIO (Non-blocking I/O) [19].
- **Tomcat**: es un conocido contenedor de aplicaciones web en Java que implementa la especificación de *servlets* de Oracle [20]. Solo permite crear aplicaciones en el lado del servidor y aunque ofrece un gran rendimiento no está optimizado para crear aplicaciones altamente concurrentes como Netty. La implementación de la capa de transporte se basa en el patrón Reactor [21].
- **Grizzly**: es otro framework que permite crear aplicaciones servidor que sean escalables basándose en Java NIO. Como Netty, intenta hacer fácil la creación de este tipo de aplicaciones ofreciendo abstracciones sobre conceptos de memoria, redes y concurrencia de bajo nivel [22].

2.3 Soluciones de Orquestación de Contenedores

El motivo principal por el que ejecutar los agentes en contenedores[4] es ser capaces de desplegarlos a demanda con tan solo un clic a través de un interfaz web. Para conseguirlo es necesario una herramienta de orquestación que pueda gestionar el ciclo de vida del contenedor y su integración con los recursos hardware sobre los que ejecute. Las tecnologías existentes debían cumplir los siguientes requisitos:

[4] Hace referencia a la tecnología de aislamiento de recursos del núcleo de Linux a través del uso de *cgroups* y *namespaces* [23].

1. La solución debe ser válida para instalaciones en infraestructura de cliente o infraestructura gestionada por Devo.
2. La herramienta tiene que ser conocida y aprobada por el equipo de operaciones que podría encargarse de la gestión.
3. Debe permitir el despliegue de agentes de forma automática a través de un API.
4. Debe tener un mantenimiento constante de errores y seguridad.
5. Debe poder configurarse en modo clúster.
6. Debe tener un fácil mantenimiento para las instalaciones en cliente.
7. Debe ser soportado por proveedores en la nube como Amazon y Google.

Una vez listados los requisitos pasamos a realizar una comparativa entre dos herramientas: Swarm y Kubernetes.

Cuando hablamos de Swarm nos referimos a Docker [24] en modo *swarm,* la tecnología que ofrece Docker para crear un clúster de contenedores a través de su motor [25]. Docker es un conjunto de herramientas de construcción de imágenes y ejecución de contenedores. Swarm no es más que el funcionamiento distribuido de múltiples motores de Docker en diferentes máquinas. Se ha elegido por ser simple y tener una curva de aprendizaje corta.

Por otro lado tenemos Kubernetes o k8s [26], un sistema creado por Google [27] para la automatización del despliegue y escalado de contenedores. A diferencia de Docker, permite el uso de múltiples motores de ejecución de contenedores, es mucho más completo y, a la vez, complejo de mantener. En Devo ya se usa Kubernetes, esto lo hace una herramienta muy factible de antemano.

A continuación, se añade una tabla comparativa de las funcionalidades necesitadas que cumple cada herramienta:

REQUISITO	SWARM	KUBERNETES
1	✔	✔
2	✘ (Por 4)	✔
3	✔	✔
4	✘ (Está siendo sustituido por k8s y el mantenimiento es cada vez menor) [28]	✔
5	✔	✔
6	✔	✘ (k3s como alternativa)
7	✔	✔

Tabla 1. Evaluación comparativa entre Swarm y Kubernetes.

Debido a los puntos negativos de Swarm, que Kubernetes ya era conocido y que además ofrece una alternativa mucho más simple, como k3s [29], se decidió elegir Kubernetes como herramienta de orquestación.

3

Análisis y Especificación

3.1 Análisis y Especificación de Requisitos

Después de haber analizado el estado del arte de frameworks de desarrollo de aplicaciones similares, nos enfocaremos en el análisis de funcionalidades que el framework requerirá. El proceso de recogida de requisitos es un proceso fundamental en todo desarrollo, puesto que define y acota la funcionalidad de los componentes que lo componen. La planificación del desarrollo tiene relación directa con los requisitos y no sería posible realizarla sin este paso.

Como se define en la ingeniería de requisitos [30], podemos tener requisitos funcionales y no funcionales. Los funcionales expresan la función de los componentes, es decir, su comportamiento. Los no funcionales expresan atributos de calidad, es decir, aquellos que no definen la funcionalidad del sistema.

A continuación detallamos el conjunto de requisitos funcionales y no funcionales de Boson:

Funcionales

IDENTIFICADOR	NOMBRE	DESCRIPCIÓN
RF1	Entrada de datos	Los agentes deben poder ser configurados para leer los datos remotamente o recibir los datos en un puerto
RF1.1	Creación de agente activo	El framework deberá poder crear agentes que se configuren para obtener los datos de las fuentes
RF1.2	Creación de agente pasivo	El framework deberá poder crear agentes que se configuren en modo servidor
RF3	Reusabilidad de filtros	Los filtros de procesamiento deben poder ser reutilizables entre múltiples agentes

RF4	Salida de datos	Los agentes deben poder ser configurados para enviar los datos a Devo
RF5	Configuración de agente: componentes	La configuración de un agente debe hacerse con un solo canal de entrada, múltiples filtros y un canal de salida
RF6	Configuración de agente: interfaz	La configuración de un agente debe hacerse usando el patrón de API fluida
RF8	Soporte de protocolos	El framework deberá permitir crear agentes que implementen protocolos sobre TCP o UDP
RF9	Configuración de hilos de procesamiento por canal	Debe poder ser modificable la cantidad de hilos dedicados al procesamiento de un canal
RF10	Uso de pool de hilos	El framework debe usar pool de hilos para gestionar eficientemente la creación de ellos

RF11	Filtros de procesamiento	La lógica del agente debe poder expresarse mediante filtros de procesamiento, que se ejecutarán en orden de adición.

Tabla 2. Listado de requisitos funcionales del framework.

No Funcionales

IDENTIFICADOR	NOMBRE	DESCRIPCIÓN
RNF1	Lenguaje de programación	El lenguaje de programación utilizado debe ser Java, el lenguaje de uso común en Devo
RNF2	Ejecución en contenedores	Los agentes ejecutarán sobre contenedores
RNF3	Escalabilidad	Los agentes deberán poder escalar con múltiples conexiones concurrentes

Tabla 3. Listado de requisitos no funcionales del framework.

4

Diseño del Framework

4.1 Introducción

Como se introdujo en el capítulo inicial, Boson es un framework en Java diseñado con el objetivo de crear agentes de transmisión de datos que sean capaces de soportar altas cargas de desplazamiento de datos desde las fuentes hasta la plataforma de Devo. En los siguientes capítulos analizaremos el diseño en detalle.

En primer lugar, comenzaremos explicando la metodología y planificación seguida en el desarrollo. Después explicaremos los conocimientos teóricos básicos de redes y sistemas operativos en los que se basan los componentes del framework. Una vez asentada la base teórica, presentaremos la arquitectura general y diseccionaremos cada componente en profundidad.

Por último, veremos cómo podemos juntar las piezas de Boson para construir agentes.

4.2 Metodología y Planificación

Durante el desarrollo se ha seguido la metodología SCRUM [31]. Esta es una metodología ágil[5] que concentra un conjunto de buenas prácticas para optimizar y acelerar el desarrollo software hoy en día. A continuación se detallan algunos de los aspectos fundamentales que se han empleado en este trabajo.

- Planificación de *sprints*[6] de dos semanas con ocho días de trabajo efectivo. Los dos días restantes se reservan para posibles reuniones y/o bloqueos que surjan.
- Reunión semanal, en vez de diaria, para el seguimiento de los objetivos marcados.

La planificación de los sprints contenía las siguientes fases:

- **Investigación:** inicialmente se dedicó un sprint completo para toda la parte de formación e investigación sobre la materia, ya que este trabajo contiene una alta carga de conocimientos técnicos sobre redes y sistemas operativos.

- **Desarrollo:** después de la fase de investigación se emplearon varios sprints para dar forma al desarrollo del framework.

- **Validación:** en esta fase se validó el funcionamiento del framework y se hicieron pruebas de rendimiento.

Una vez terminada la primera versión hubo un trabajo de iteración en el desarrollo que hizo volver a repasar algunas de las fases anteriores para mejorarlo.

[5] Se consideran metodologías ágiles aquellas que siguen el manifiesto ágil [32].

[6] En SCRUM un sprint es una ventana de ejecución de tareas muy corta, típicamente de cuatro semanas o menos.

4.3 Conceptos Básicos

Una parte fundamental del desarrollo de este trabajo ha sido la investigación y el aprendizaje de los conceptos teóricos que lo sustentan. En el desarrollo de Boson se han usado librerías de Java como NIO2 (Non-blocking I/O versión 2) o la API (Application Programming Interface) de acceso a memoria directa, *off-heap*. El uso de ellas no es trivial, son una abstracción de áreas avanzadas como redes y procesamiento concurrente en sistemas operativos y, por lo tanto, requieren de un aprendizaje de conceptos de bajo nivel.

A continuación, se intenta explicar de forma sencilla algunos de los conceptos clave que el lector podría necesitar para entender las siguientes partes de este trabajo. De la misma forma, se invita a seguir las referencias si se desea saber más en profundidad sobre la materia.

4.3.1 Modelo OSI y Protocolos de la Capa de Transporte

Dos aplicaciones que se comuniquen a través de la red, de forma local o distribuida, necesitan llegar a una serie de acuerdos antes de comenzar a hablar entre ellas. Por ejemplo, decidir quién inicia la conexión, lo que definiría el modelo básico Cliente-Servidor, es decir, qué punto actúa en modo demonio a la espera de conexiones y quién actúa de forma activa. Para establecer este tipo de acuerdos lo que se usa son los protocolos de la pila de protocolos de red como OSI o TCP/IP[7].

[7] Ambos modelos definen un estándar para interconectar sistemas de distinta procedencia a través de protocolos de red, pero con sutiles diferencias entre capas [33].

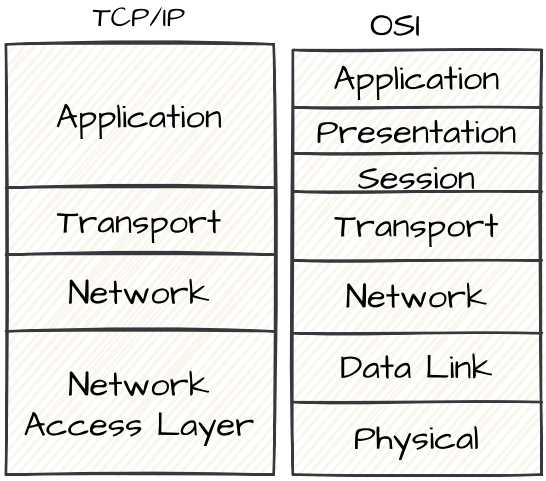

Figura 2. Comparativa de capas entre TCP/IP y OSI.

Entre todas las capas de estos modelos, clientes y servidores se construyen a partir de la tercera capa (cuarta en OSI), la capa de transporte. Esta capa define los dos protocolos principales en los que se basa el framework, TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). Ambos protocolos dan soporte a otros protocolos de las capas superiores (ver Figura 2), pero tienen bastantes diferencias entre ellos:

TCP

- Orientado a la conexión.
- Transmisión de flujo de datos binario.
- Control de flujo y errores, lo que lo hace fiable frente a UDP.

UDP

- Transmisión de datos en datagramas.
- No orientado a la conexión.
- Más rápido, puesto que elimina la lógica de control de TCP a cambio de la pérdida de fiabilidad.

Como veremos más adelante en la arquitectura del framework, para soportar la creación de agentes que funcionen sobre ambos protocolos, necesitamos abstraer el modelo para implementar diferente lógica de procesamiento de datos, puesto que uno es binario y el otro no, en TCP existe el concepto de conexión y en UDP no.

4.3.2 Sockets de Red

La primitiva que implementan los sistemas operativos para representar una un punto de comunicación se denomina *socket*, que no es más que una abstracción sobre la pila de protocolos que son los encargados de establecer cómo los datos que fluyen por el socket deberían ser empaquetados y dirigidos por la red.

Cada sistema operativo tiene su propia librería de sockets y los lenguajes de programación crean adaptadores para usarlas. Por ejemplo, cuando creamos un socket en Java para crear una conexión TCP, la máquina virtual hará una llamada a código nativo de la librería de sockets que ofrezca el sistema. Los sistemas de la familia Unix[8] implementan el interfaz de sockets del estándar POSIX (Portable Operating System Interface) [34]. En cambio, Windows tiene su propia implementación Winsock, pero ambas están basadas en el interfaz de sockets de Berkeley que fue creado en 1983 para el sistema operativo BSD (Berkeley Software Distribution) y que más tarde se convirtió en un componente mas de la especificación POSIX [35].

Existen cinco modelos diferentes de E/S[9] que podemos usar con sockets y que se relacionan estrechamente con la ejecución de la aplicación pudiendo bloquearla

[8] Término usado comúnmente para hacer referencia a la familia de sistemas operativos (Linux, BSD, MacOS, etc.) que derivaron de la primera versión del sistema operativo Unix.
[9] Hace referencia a la Entrada/Salida del sistema operativo, del inglés I/O o Input/Output.

a la espera de eventos de E/S. Los modelos son bloqueante, no bloqueante, con multiplexación, dirigido por señal y asíncrono.

- **Bloqueante:** en este modo la aplicación hace una llamada de sistema y el hilo (del inglés *thread*) bloquea hasta que el núcleo (del inglés *kernel*) recibe datos y son copiados desde su espacio al espacio de usuario en el *buffer* de la aplicación. Por defecto, todos los sockets son bloqueantes en la creación. En la Figura 3, podemos ver cómo sería la secuencia de una operación de lectura de datos del socket.

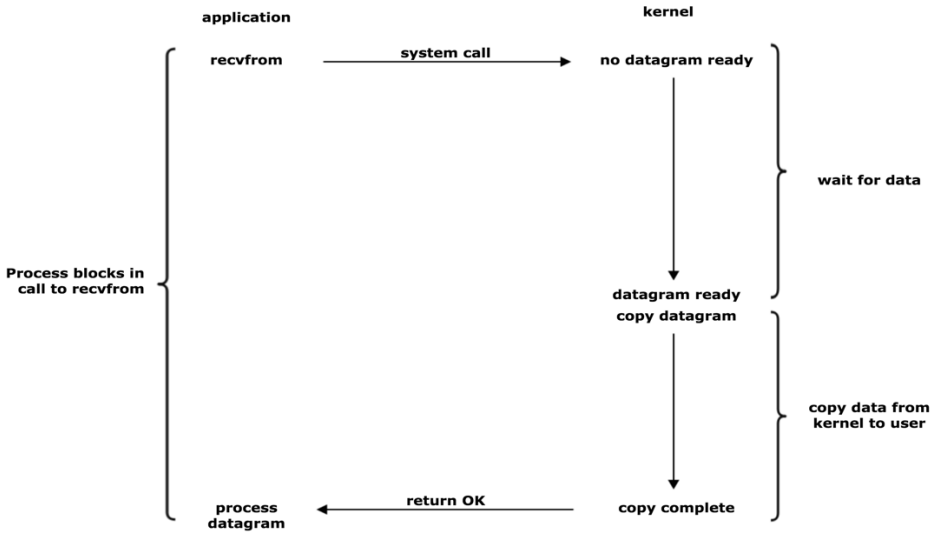

Figura 3. Modelo de E/S bloqueante [36].

Este ha sido el modelo tradicional en las versiones iniciales de Java con la API IO y fue usado en servidores de aplicaciones como Tomcat hasta que se comenzara a probar con NIO [37]. Debido a la naturaleza del modelo, los servidores que lo usaban tenían que crear un hilo por conexión lo cuál no escalaba en entornos que necesitaban ser capaces de mantener concurrentemente miles de conexiones [38], ya que la creación y mantenimiento de hilos es costoso en memoria.

- **Modo no bloqueante:** a diferencia del bloqueante, cuando se realiza la llamada al sistema pueden ocurrir dos cosas:

    - si la operación puede ser completada inmediatamente, entonces los datos se copian del espacio del núcleo al espacio de la aplicación;
    - si la operación no puede ser completada de manera inmediata, entonces se devuelve un código de error y la aplicación debe seguir haciendo llamadas al sistema repetidamente hasta que se complete o cancele.

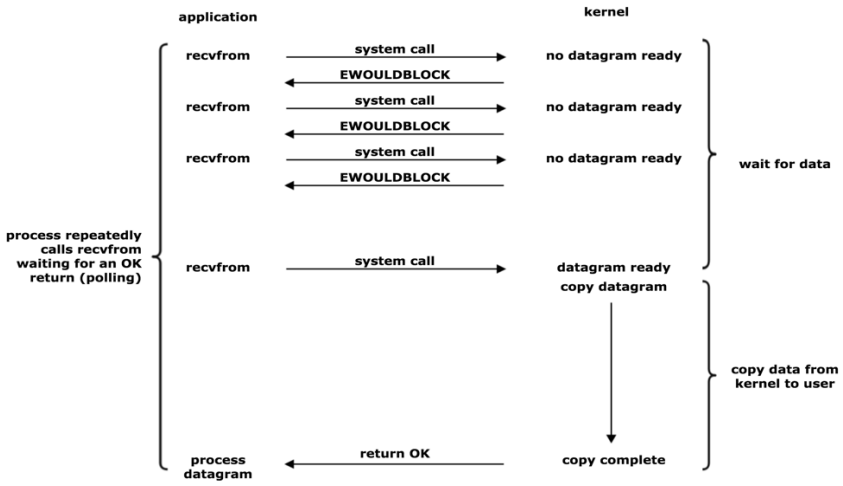

Figura 4. Modelo de E/S no bloqueante [36].

Como se aprecia en la Figura 4 en ningún caso bloquearía el proceso, en este modo, lo que nos permitiría decidir en la lógica de la aplicación si se quiere dejar un intervalo de tiempo haciendo otro trabajo antes de volver a hacer una llamada para comprobar si el evento se ha completado. Este modelo puede soportar una mayor concurrencia, pero la espera activa provoca muchos cambios de contexto entre modo usuario y modo privilegiado [39].

- **Multiplexación:** este modo es una variante del modelo no bloqueante, pero con notificaciones bloqueantes. Se hace uso de la llamada bloqueante

`select()` a la espera de notificación de los eventos en los que se haya registrado interés. A simple vista, parece no ofrecer ninguna ventaja sobre el modelo no bloqueante, pero la ventaja es importante realmente ya que este modo hace uso de un selector que permite multiplexar o esperar notificaciones de eventos de múltiples sockets de forma concurrente.

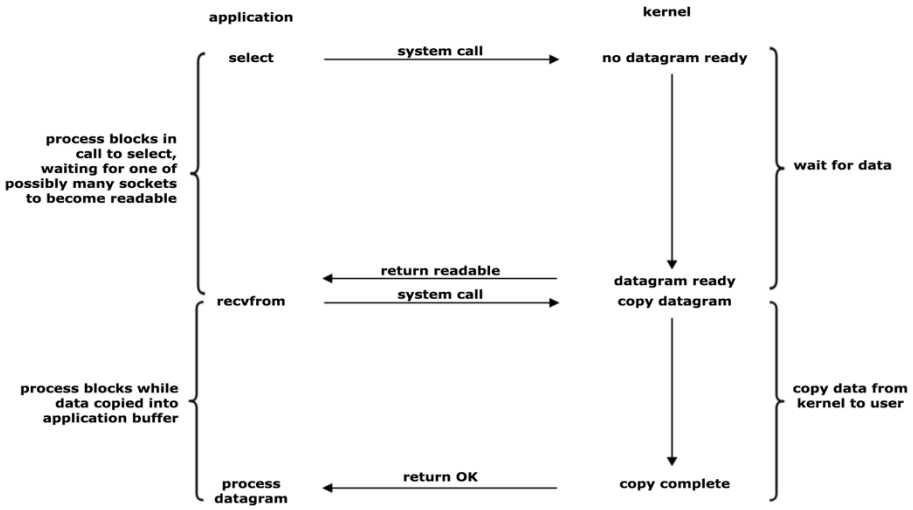

Figura 5. Modelo de E/S con multiplexación [36].

Como veremos más adelante, el modo con multiplexación ha sido implementado en Boson para permitir que los agentes tengan la capacidad de escalar en entornos concurrentes. Netty implementa también este modelo debido a las capacidades de concurrencia [19].

- **Dirigido por señal:** en este modelo la aplicación realiza una llamada no bloqueante al sistema y registra un manejador de señal. Cuando el evento se ha completado el núcleo envía una señal a la aplicación para invocar el manejador y copiar los datos.

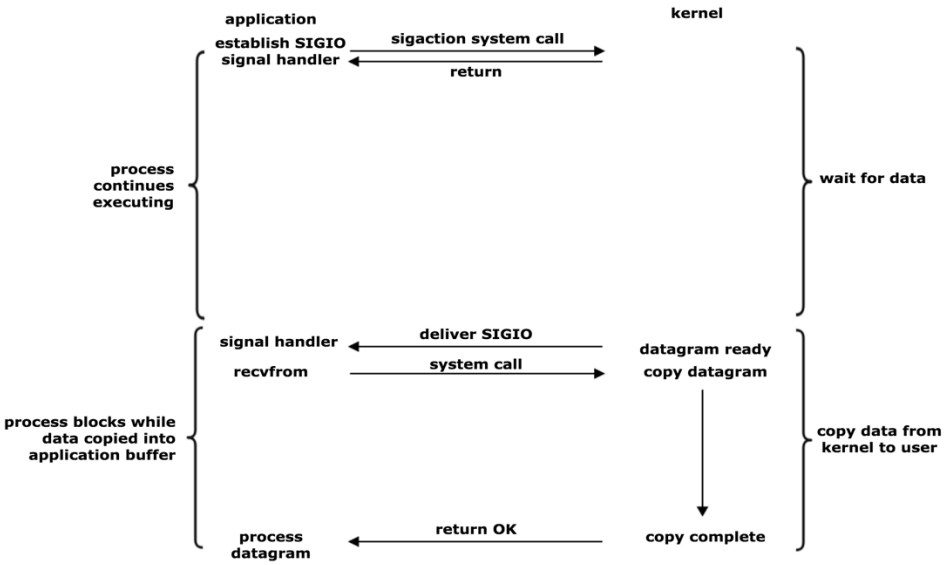

Figura 6. Modelo de E/S dirigido por señal [36].

La ventaja de este modelo es que la aplicación no bloquea hasta que llega la señal de notificación para terminar la operación. Es muy parecido al modelo asíncrono, pero en este caso la aplicación ejecuta la llamada al sistema cuando recibe la señal y hace el mismo procedimiento que hemos visto en los modelos anteriores, bloquea mientras se copian los datos de un espacio a otro.

- **Asíncrono:** a diferencia del modelo dirigido por señal, este es verdaderamente asíncrono, como su nombre indica, porque inicia la operación haciendo una llamada al sistema no bloqueante y el núcleo se encarga de todo el trabajo, es decir, espera y copia los datos. Una vez ha completado la operación se notifica a la aplicación invocando un manejador de procesamiento de datos.

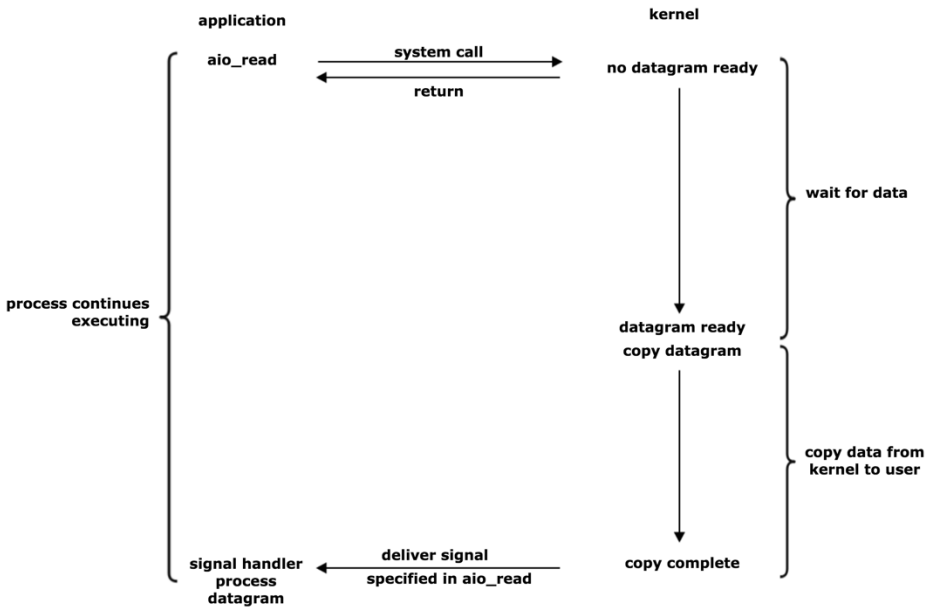

Figura 7. Modelo E/S asíncrono [36].

Como hemos visto, este modelo delega todo el trabajo de E/S en el sistema operativo, por lo que permite que la lógica de la aplicación se centre exclusivamente en el procesamiento de datos. Es el modelo que, en la teoría, ofrecería una mayor concurrencia, pero en la práctica no todos los sistemas operativos lo implementan de forma eficiente [40].

4.3.3 Espacio de Usuario y Espacio Privilegiado

Estos dos conceptos hacen referencia a las dos regiones en las que los sistemas operativos Linux dividen la memoria [41]:

- **Espacio de usuario:** del inglés *user space*, es la zona de memoria donde los procesos del usuario ejecutan. El núcleo se encarga de gestionar las aplicaciones ejecutando en este espacio para que no entren en conflicto entre ellas.
- **Espacio privilegiado:** del inglés *kernel space*, es la zona de memoria donde reside y se ejecuta el código del núcleo.

Los procesos que ejecutan en el espacio de usuario no tienen acceso al espacio privilegiado, en cambio el núcleo sí tiene acceso a todas las regiones de memoria. Cuando un proceso necesita acceder al espacio privilegiado, lo hace a través de llamadas del sistema, típicamente para realizar acciones privilegiadas como el uso de periféricos, por ejemplo E/S.

4.3.4 Java NIO2 y Memoria Directa

Java ha ido evolucionado sus funcionalidades a la vez que lo hacían los sistemas operativos. En sus primeros años solo existía la interfaz IO que usaba el modelo bloqueante de E/S. Unos años después en la versión 1.4, introdujeron NIO que permitía hacer uso de llamadas no bloqueantes y por último, evolucionaron a la versión NIO2 en Java 1.7 donde se mejoró el soporte no bloqueante y se añadió soporte asíncrono. A partir de ahora, se usará NIO para hacer referencia a la última versión de Java, NIO2.

En el paquete java.nio se definen las interfaces y clases que Java ofrece para representar conexiones entre entidades usando `Channel`, transferencia de datos mediante *Buffers* y multiplexación a través de `Selector` [42]. En el apartado 4.4 veremos cómo se usa la API para construir la base de transmisión de datos de un agente.

En los modelos de E/S del apartado 4.3.2 hemos visto que para realizar una operación de lectura de datos de un socket era necesario realizar una llamada de sistema para ejecutar en modo privilegiado y copiar los datos de un espacio a otro [39]. Es importante resaltar este funcionamiento ya que la aplicación para una operación de lectura de socket necesita:

1. Hacer una llamada de sistema, lo que implica un cambio de contexto de modo usuario a modo privilegiado.
2. Usar un buffer intermediario para copiar datos del buffer de lectura en el espacio de memoria privilegiado al espacio de usuario.
3. Hacer un cambio de contexto para volver del modo privilegiado al modo usuario.

En aplicaciones que requieren una baja latencia, no es eficiente esta forma de trabajar, por lo que Java introdujo en la clase `ByteBuffer` de NIO una forma de acceder a memoria directa, fuera del *heap* [43], también llamada *off-heap*, a través de llamadas nativas[10]. En la documentación de la clase se detalla perfectamente el propósito de los buffers con memoria directa:

> "The contents of direct buffers may reside outside of the normal garbage-collected heap [...] It is therefore recommended that direct buffers be allocated primarily for large, long-lived buffers that are subject to the underlying system's native I/O" [45].

La ventaja de la memoria directa reside en que se elimina la necesidad que teníamos en la operación anterior de lectura de socket de copiar los datos del espacio privilegiado al de usuario, ya que el controlador de E/S podría copiar los datos directamente en el buffer de memoria directa de la aplicación gracias a una característica fundamental de la arquitectura en los sistemas operativos, DMA (Direct Memory Access) [46].

No todo son ventajas. El uso de memoria directa tiene sus consecuencias si no se utiliza correctamente, puesto que queda fuera de la gestión de memoria del

[10] Anteriormente existía la API *sun.misc.Unsafe* como una forma de acceder a espacios de memoria fuera del heap, pero nunca se documentó ni recomendó su uso [44].

recolector de basura de la JVM, es necesario realizar la liberación manual para no incurrir en agotamiento de memoria sin darnos cuenta.

Para solventar este tipo de problemas, Java 14 introdujo la primera versión de la API de acceso a memoria foránea [47] que facilita trabajar con memoria del heap y nativa de una forma más segura y eficiente. La interfaz principal es `MemorySegment` y representa una región de memoria contigua que tiene dos límites, uno espacial y otro temporal. El espacial evita que se acceda a direcciones de memoria fuera de su límite y el temporal prohíbe operaciones de acceso después de que el segmento se haya cerrado. Una de las características por las que se usa en Boson, en vez de `ByteBuffer`, es porque ofrece la posibilidad de crear vistas sobre un segmento y permite control de acceso por hilo. Esto permite poder tener un solo segmento con múltiples particiones donde cada una es dedicada de forma aislada para las operaciones de E/S de cada cliente en un agente que actúe como servidor.

4.4 Arquitectura

En la introducción se han explicado los hechos que han motivado la necesidad de desarrollar el framework que facilitara construir agentes de recolección de datos y se han analizado tecnologías existentes similares. Una vez explicados los fundamentos en los que se basa una aplicación de transmisión de datos, vamos a pisar el barro centrándonos en los detalles técnicos de la arquitectura de Boson donde se explicarán las decisiones de su arquitectura y cómo se han elegido ciertos patrones de diseño con el objetivo de crear código reusable, fácil de mantener, extensible y escalable [48].

En primer lugar presentaremos la arquitectura general que rodea a un agente y seguidamente diseccionaremos las partes más importantes que lo componen.

4.4.1 Arquitectura General

En la Figura 8 se muestra la arquitectura de componentes de un agente de izquierda a derecha y de arriba hacia abajo:

Fuentes de datos: formado por cualquier tipo de origen de datos. Puede ser desde un router que genera estadísticas de red a una base datos expuesta a través de una API REST (Representational State Transfer).

Código de agente: estaría compuesto por la lógica de procesamiento de datos provenientes de las fuentes a través de un canal de entrada y un canal de salida. Podría ser cualquier tipo de aplicación de red que se base en TCP o UDP y tendría Boson como motor de ejecución.

Boson: mantiene un motor de ejecución para la transmisión de datos desde las fuentes hasta Devo y la lógica de procesamiento definida por el agente.

Sistema operativo: ofrece la arquitectura necesaria para poder hacer uso de los componentes hardware subyacentes.

Relay y plataforma de datos de Devo: el destino final de los datos.

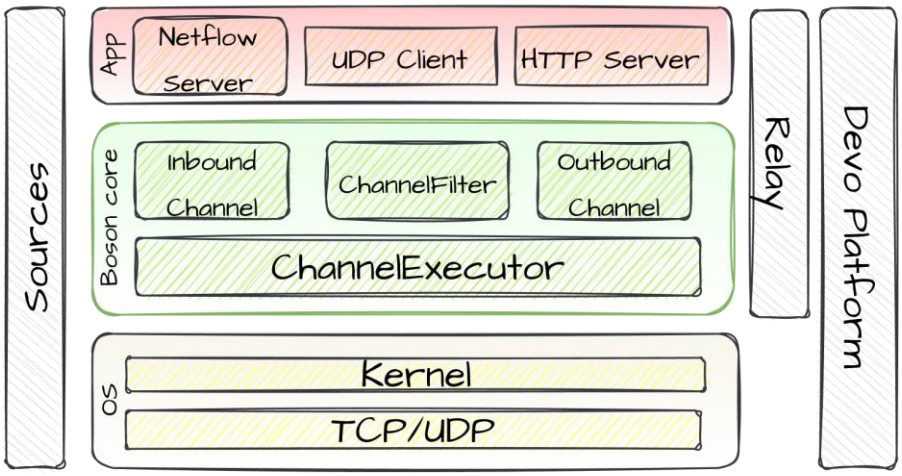

Figura 8. Arquitectura de un agente de transmisión de datos con Boson.

A través de la abstracción conseguimos que la capa de aplicación solo tenga que definir qué necesita para transmitir datos y el framework se encarga de forma transparente de implementar la capa de transporte. Esta división de responsabilidades hace posible que extender las funcionalidades del framework o cambiar el comportamiento de la aplicación pueda hacerse independientemente. Para conseguirlo Boson hace uso de las siguientes API:

- **Channel**: define un envoltorio sobre un `Channel` de Java NIO y añade el comportamiento necesario para el contexto de ejecución del agente.
- **ChannelExecutor**: define el comportamiento de la capa de transporte sobre la que comunica `Channel`.
- **ChannelFilter**: define cómo deben ser los filtros que implementan los agentes para procesar el flujo de datos.
- **Pipeline**: se encarga de la inicialización y mantenimiento del contexto de configuración del agente.
- **State**: establece cómo obtener y guardar el estado de un agente.

A continuación se analizan en detalle cada uno de ellos.

4.4.2 Channel API

Como se ha indicado antes, esta API es un envoltorio sobre un objeto de tipo `Channel` de Java al que se añade el comportamiento necesario para definir la capa de transporte y el contexto de ejecución del agente, que se compone principalmente de un canal de entrada, un canal de salida y los filtros de procesamiento.

El canal representa un punto de comunicación por el cual se transmiten datos. Tiene un punto de entrada y uno de salida por los que se realizan las operaciones de lectura y escritura de datos. La configuración mínima de un agente debe estar compuesta por dos canales: uno de entrada y otro de salida. Como se aprecia en la Figura 9, la salida del canal de entrada siempre es la entrada del de salida.

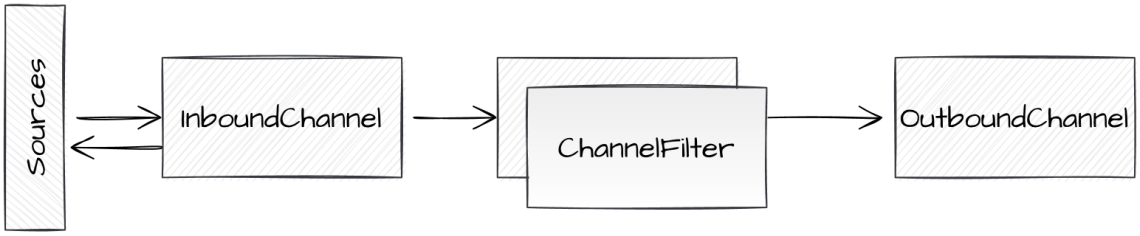

Figura 9. Flujo de comunicación de canales de un agente.

La implementación del canal es independiente de si este se ejecuta como canal de entrada o de salida. Boson implementa por defecto la jerarquía de clases que vemos en la Figura 10 para poder configurar un agente con sockets no bloqueantes o asíncronos.

La interfaz principal es `Channel` de la que heredan `ServerChannel` y `ConnectableChannel`; el primero, implementado por canales tipo servidor y el segundo, por canales de tipo cliente. `Channel` tiene una implementación directa, `AbstractChannel` que implementa el comportamiento común para todos los tipos de canal, tanto no bloqueantes (Nio), como asíncronos (Aio). `AioServerChannel`

y `AioChannel` extienden `AbstractChannel` con lógica asíncrona de cliente y servidor. `AbstractNioChannel` hace lo mismo, con la lógica para canales no bloqueantes. Esta última clase es la que extienden las restantes clases Nio que se diferencian por el prefijo `NioMessage-` para los canales UDP y `NioStream-` para los canales TCP, porque como hemos explicado anteriormente, tienen características muy diferenciadas que obligan a separar su lógica.

Finalmente, los dos tipos de canales servidor no bloqueante implementan `AcceptableChannel` que define el comportamiento de aceptación de conexiones.

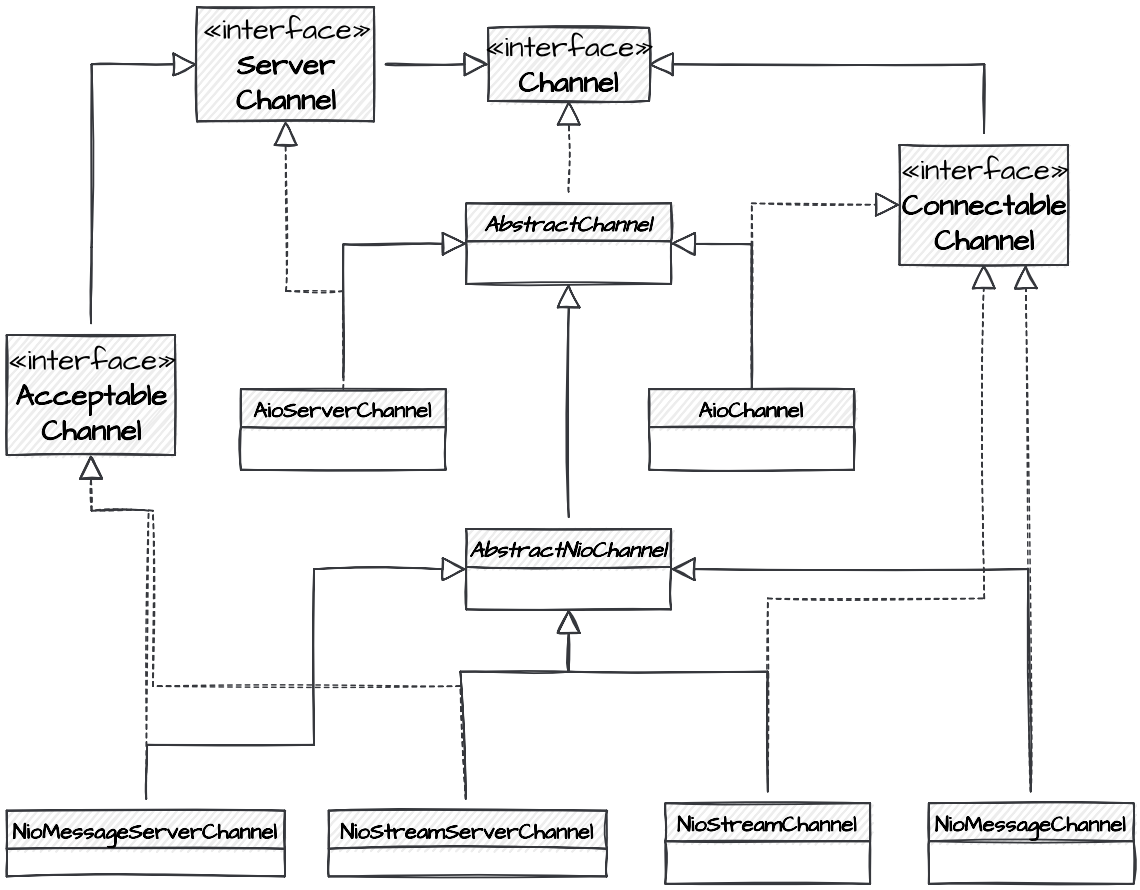

Figura 10. Diagrama de clases que componen la API de Channel.

Las clases no abstractas de la jerarquía pueden ser utilizadas para crear agentes activos o pasivos, dependiendo del canal de entrada que elijamos. Todas

implementan un constructor que acepta dos argumentos, uno de tipo `SocketAddress` [49] y otro de tipo `ChannelExecutor:`

```
public Channel(SocketAddress, ChannelExecutor)
```

para crear una instancia a partir de una dirección de red local o remota y el tipo de transporte que se desee.

4.4.3 ChannelExecutor API

Para poder transmitir datos desde las fuentes hasta su destino se necesita un motor de ejecución que modele la capa de transporte. Para ello se define `ChannelExecutor` que tiene dos implementaciones diferentes (ver Figura 11):

- **NioChannelExecutor**: implementación basada en el patrón *Reactor* [50] usando un modelo de E/S con multiplexación (no bloqueante) para tener capacidad de escalar en situaciones de alta carga. Existen múltiples variantes de este patrón, aunque en este framework se ha implementado la más sencilla con un único aceptador.
- **AioChannelExecutor**: es una implementación basada en Java NIO2 usando la clase `AsynchronousChannelGroup` [51] que se encarga de implementar el modelo de E/S asíncrona que explicamos anteriormente.

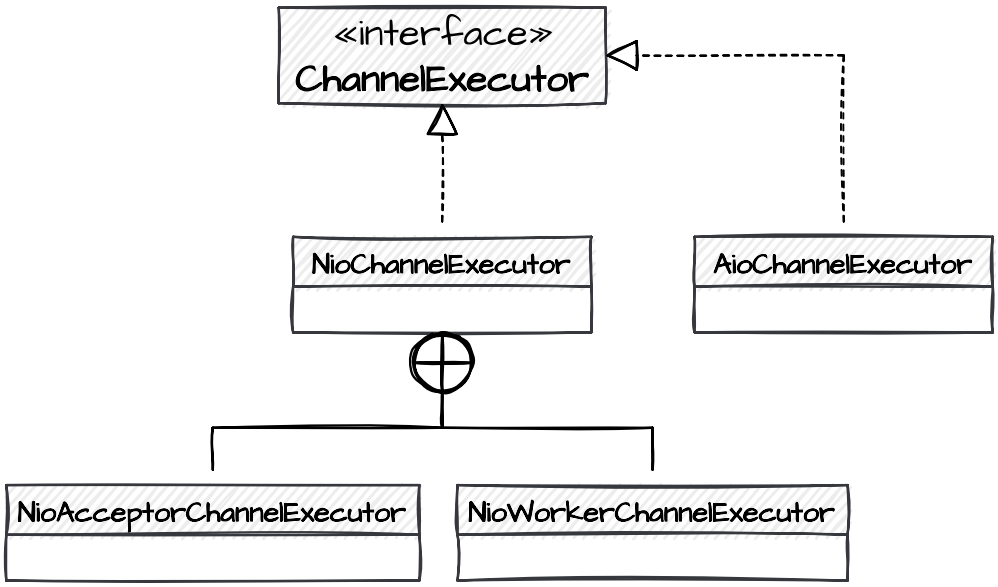

Figura 11. Jerarquía de clases de ChannelExecutor.

Como se ha explicado en los conceptos básicos los sockets no bloqueantes son ideales para aplicaciones con alta concurrencia. En Boson se implementa este modelo de la siguiente forma mediante el patrón Reactor:

1. `NioChannelExecutor` inicia un grupo de `NioWorkerChannelExecutor` igual al doble del número de núcleos de CPU disponibles y un `NioAcceptorExecutor` que contiene un solo `NioWorkerChannelExecutor`. Cada `NioWorkerChannelExecutor` tiene asignado un solo hilo a través de un `ExecutorService` [52] y un selector de E/S.
2. Cuando se inicializa un canal de tipo `AcceptableChannel`, automáticamente se registra con el `NioAcceptorExecutor` y empieza a ejecutar el bucle de eventos a la espera de recibir conexiones.
3. En cuanto llega una nueva conexión, `NioAcceptorExecutor` devuelve un `Channel` después de aceptarla y lo registra con uno de los `NioWorkerChannelExecutor` de `NioChannelExecutor`, la selección se

hace mediante *round-robin*[11]. El registro se hace con el selector del `NioWorkerChannelExecutor` y así poder multiplexar entre todos los canales (conexiones) registrados.

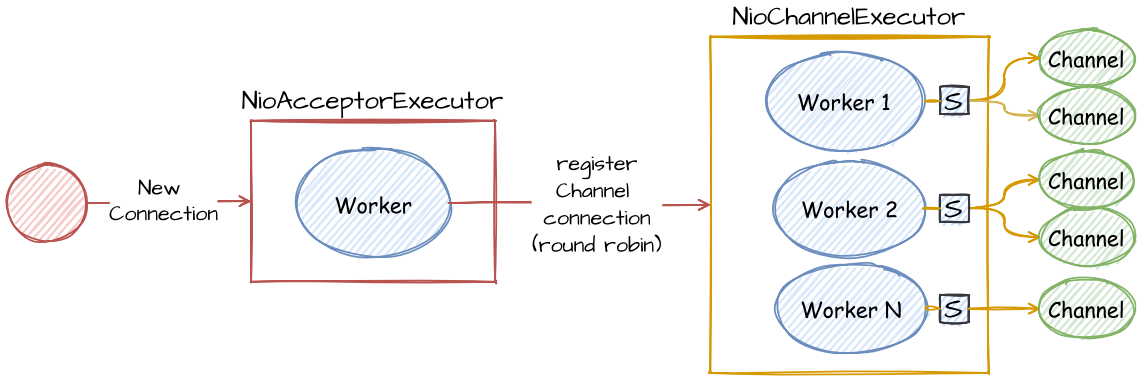

Figura 12. Patrón Reactor en NioChannelExecutor.

En un agente en activo (cliente) no se usaría una instancia de tipo `NioAcceptorExecutor,` ya que no es necesario aceptar conexiones y `NioChannelExecutor` tan solo tendría un `NioWorkerChannelExecutor` para procesar la E/S de un único canal.

Gracias a la división que se hace de un `NioWorkerChannelExecutor` para múltiples canales, evitamos tener que resolver problemas de concurrencia que se originarían con los accesos a los canales de forma compartida entre varios hilos de ejecución.

Por otro lado tendríamos el modelo asíncrono en el que se simplifica el trabajo de la aplicación a tan solo tener que definir una función manejadora que se ejecutará cuando la operación de E/S se haya completado (ver Figura 13).

[11] Algoritmo que asigna en orden circular una entidad a una lista de recursos de procesamiento [53].

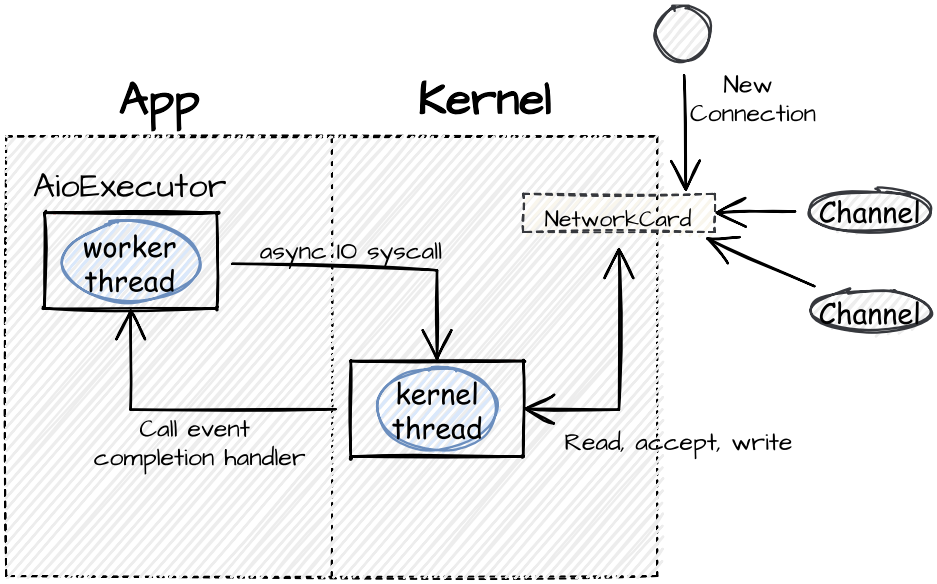

Figura 13. Modelo asíncrono en AioChannelExecutor.

4.4.4 ChannelFilter API

Esta API es fundamental en el desarrollo de agentes puesto que es la interfaz necesaria para implementar la lógica de procesamiento de datos. En la siguiente figura vemos cómo se relaciona con las demás interfaces del framework implementando el patrón filtro de intercepción [54].

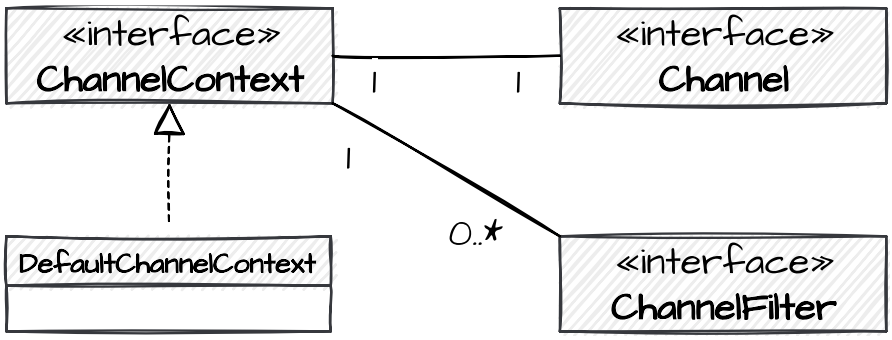

Figura 14. Relación de clases con ChannelFilter.

Un `Channel` está compuesto de un `ChannelContext`, que está formado por una lista de `ChannelFilter` para la entrada del canal y otra lista de `ChannelFilter` para la salida del canal. Dependiendo de si se escribe o se lee se aplicará una

lógica u otra, según los filtros que se hayan definido. `ChannelFilter` tan solo define dos métodos:

```
apply(PipelineContext context, MemorySegment data, DataContainer result)

void failed(PipelineContext context, Throwable cause)
```

El primero es obligatorio implementarlo para definir un filtro y es invocado automáticamente por el `ChannelContext` y en orden de aparición en la lista de filtros, formando así una cadena hasta que se ejecute el último o haya un fallo que invocase el método `failed()` con la excepción para capturarla si se quiere. Con este diseño conseguimos que el procesamiento de datos pueda ser reusable y modular porque podemos tener filtros comunes entre varios agentes, por ejemplo de TLS, reutilizarlos y desarrollarlos independientemente sin que afecte a la lógica central del agente en cuestión.

Como hemos comentado antes, se hace uso de la clase `MemorySegment` con memoria directa para evitar hacer copias de datos innecesarias.

4.4.5 StateManager API

Cuando un agente se ejecuta en contenedores sobre un orquestador, es posible que algún momento por algún motivo tenga un problema y se pare el contenedor. Esto provoca que el orquestador intente replanificar la ejecución del contenedor, pero en el arranque del agente probablemente se quiera continuar procesando datos en el punto en que se dejó. Por este motivo, esta API permite implementar un gestor de estado que puede ser usado por un agente durante el arranque para obtener su estado inicial o durante la ejecución para guardar su estado.

4.4.6 Pipeline API

Por último, tenemos la API que ensambla todas las piezas del framework para poder construir un agente, `Pipeline`. Esta clase a través del patrón de interfaz fluida [55] permite encadenar métodos sobre el mismo objeto para configurar las partes del agente, canal de entrada, filtros y canal de salida.

```
DefaultPipeline pipeline = new DefaultPipeline();

NioStreamServerChannel serverChannel =
    new NioStreamServerChannel(
    new InetSocketAddress(8085),
    new NioChannelExecutor());
pipeline.inbound(serverChannel);

NioStreamChannel channel = new NioStreamChannel(
    new InetSocketAddress(8082),
    new NioChannelExecutor());
pipeline.outbound(channel);

pipeline.addInboundFilter(new ProcessingFilter());
```

Figura 15. Ejemplo de uso de la API de configuración.

5

Implementación de Agentes

5.1 Objetivo

El objetivo de este apartado es mostrar cómo podemos implementar agentes de una forma sencilla, modificar la capa de transporte de un agente con tan solo cambiar una clase o ver cómo se puede reutilizar filtros entre agentes y el envío de datos a Devo.

El ejemplo que vamos a realizar está relacionado con un protocolo muy sencillo de implementar, el protocolo de tiempo [56]. Es un protocolo que se usaba para implementar servidores de hora antes de que llegase el protocolo NTP (Network Time Protocol) [57].

El entorno de prueba que se ha preparado está compuesto por dos agentes Java, un servidor Netty que implementa el protocolo de tiempo [58], una base de datos en Redis [59] para gestionar el estado y un Relay para el reenvío de datos

a Devo. Todo ejecutando sobre contenedores en Docker. El entorno quedaría como en la siguiente figura.

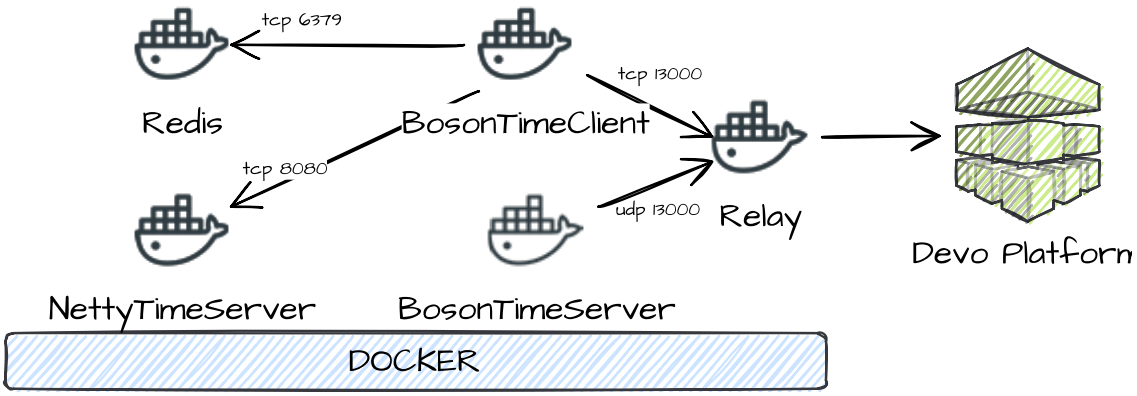

Figura 16. Arquitectura del entorno de pruebas.

5.2 Implementación de Agente Pasivo

El agente pasivo que se ha desarrollado es un simple agente que actúa como servidor de reenvío de hora en formato Syslog. Está formado por los siguientes componentes:

- `NioStreamServerChannel` que configura el puerto de escucha y un `NioChannelExecutor` con dos hilos.
- `NioMessageChannel` como canal de salida UDP para el envío de datos al Relay a través del puerto 13000 UDP.
- Dos filtros de procesamiento, uno para trazas de ejecución `LoggingFilter` y otro para la transformación del dato de tiempo a Syslog, `TimeToSyslogFilter`.

En la Figura 17, vemos cómo utilizar la API de configuración para componer las piezas de un agente. Como se ha comentado anteriormente, este API permite que configurar agentes sea un trabajo sencillo.

```
DefaultPipeline pipeline = new DefaultPipeline();
NioStreamServerChannel clientChannel =
  new NioStreamServerChannel(
    new InetSocketAddress(port),
    new NioChannelExecutor(2));
pipeline.inbound(clientChannel);

NioMessageChannel outboundChannel = new NioMessageChannel(
  new InetSocketAddress("relay",13000),
  new NioChannelExecutor(1));
pipeline.outbound(outboundChannel);

pipeline.addInboundFilter(new LoggingFilter());
pipeline.addInboundFilter(new TimeToSyslogFilter());

pipeline.start();
```

Figura 17. Configuración de agente pasivo.

Si quisiéramos cambiar el envío al Relay de UDP a TCP, lo único que habría que hacer es cambiar el tipo del canal de salida por `NioStreamChannel`, solo eso.

La lógica que implementa este agente es la que vemos en la Figura 18. Por un lado, `LoggingFilter` solo añade trazas y, por otro, `TimeToSyslogFilter` se encarga de obtener la fecha actual en bytes y transformarla a `long` para poder construir el mensaje con Syslog.

```
public class TimeToSyslogFilter implements ChannelFilter {

  @Override
  public void apply(PipelineContext context, MemorySegment data,
    DataContainer result) {
    ByteBuffer bb = data.asByteBuffer();
    long currentTimeMillis = (Integer.toUnsignedLong(bb.getInt()) -
2208988800L) * 1000L;
    MemorySegment ms = MemorySegment.allocateNative(2000);

ms.asByteBuffer().put(timeToSyslog(currentTimeMillis).getBytes(StandardCh
arsets.UTF_8));
    context.getOutboundChannelContext().channel().write(ms);
  }

  public String timeToSyslog(long time) {
    StringBuilder sb = new StringBuilder();
    sb.append("<14>");
    sb.append(Instant.now()).append(" ");
    sb.append("test.keep.free").append(": ");
    sb.append("Boson Time Client").append("|");
    sb.append(new Date(time));
    return sb.toString();
  }
}

public static class LoggingFilter implements ChannelFilter {

  @Override
  public void apply(PipelineContext context, MemorySegment data,
    DataContainer result) {
    log.info("Logging the request number: {}. Sending time to Devo.",
      count ++);
  }

  @Override
  public void failed(PipelineContext context, Throwable cause)  {
    log.error("Error in the logging filter: {}", cause.getMessage());
  }
}
```

Figura 18. Filtros de procesamiento del agente pasivo.

Finalmente una vez transformados los datos a Syslog, se mostrarían en Devo en la tabla `test.keep.free` en formato columna comenzando por la fecha de recepción del evento, como se aprecia en la siguiente figura.

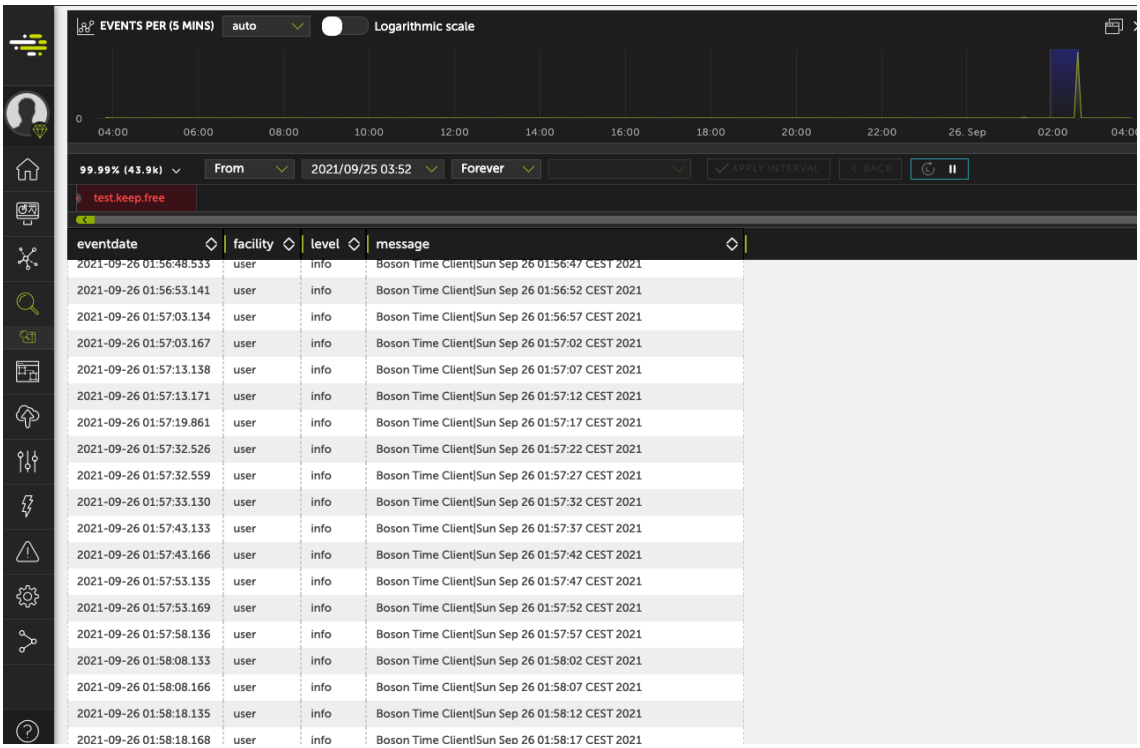

Figura 19. Visualización en Devo de los datos enviados por el agente.

Más adelante veremos, en el apartado de pruebas cómo se podría monitorizar un agente de este tipo.

5.3 Implementación de Agente Activo

Como se ha mostrado en la Figura 16 este agente se configura para que de manera activa pregunte al servidor de tiempo la hora. Después de obtenerla la procese y la convierta a Syslog para enviarla a Devo.

Empecemos viendo su configuración. Se compone de varios elementos muy similares al agente pasivo (ver Figura 20):

- `NioStreamChannel` como canal de entrada de datos. Este canal se conecta al servidor de Netty. A diferencia del agente pasivo, solo configura un hilo, puesto que al no ser un servidor siempre existirá una única conexión que procesar, no hay necesidad de multiplexación.

- `NioStreamChannel` como canal de salida de datos hacia Devo. La única diferencia con el otro agente es que este utiliza TCP para enviar al Relay.
- Dos filtros de procesamiento: `InitFilter` y `TimeToSyslogFilter`. El segundo lo hemos conocido antes, estamos aprovechando la capacidad de reutilización de filtros que nos aporta el framework. El primero se encarga del proceso de inicialización del agente y la comprobación del estado, si existe, en redis. Configura un hilo planificado para ejecutarse cada 5 segundos para pedir la hora al servidor. En caso de que se hubiese parado por algún motivo, mirará en redis cuál fue la última hora que no pudo terminar de enviar y la pedirá al servidor de nuevo, antes de seguir. Este sencillo ejemplo podría ser una analogía para un agente más complejo que obtenga datos de una API.

```
public void run() {
  DefaultPipeline pipeline = new DefaultPipeline();
  NioStreamChannel clientChannel =
    new NioStreamChannel(
      new InetSocketAddress(8080),
      new NioChannelExecutor(1));
  pipeline.inbound(clientChannel);

  NioStreamChannel outboundChannel = new NioStreamChannel(
    new InetSocketAddress("relay",13000),
    new NioChannelExecutor(1));
  pipeline.outbound(outboundChannel);

  pipeline.addInboundFilter(new InitFilter(1));
  pipeline.addInboundFilter(new TimeToSyslogFilter());

  pipeline.start();
}
```

Figura 20. Configuración de agente activo.

```java
@Override
public void apply(PipelineContext context, MemorySegment data,
  DataContainer result) {
  if (!started) {
    PipelineContext pcontext = context.getInboundChannelContext()
      .channel()
      .pipelineContext();
    StateManager sm = pcontext.getStateManager().isPresent()
      ? pcontext.getStateManager().get()
      : null;

    if (sm != null) {
      Object lastSaved = sm.get(clientId);
      if (lastSaved != null) lastTime = (long) lastSaved;
    }
    Executors.newSingleThreadScheduledExecutor()
      .scheduleAtFixedRate(() -> {
        if (lastTime != -1) {
          MemorySegment ms = MemorySegment.allocateNative(8);
          ms.asByteBuffer()
            .put("time".getBytes(StandardCharsets.UTF_8))
            .putLong(lastTime);
          context.getInboundChannelContext().channel().write(ms);
        } else {
          context.getInboundChannelContext().channel()
            .write(MemorySegment.allocateNative(1).fill((byte)0));
        }
      }, 5, 5,TimeUnit.SECONDS);
    started = true;
  }
}
```

Figura 21. Filtro de inicialización del agente activo.

5.4 Pruebas

Por último, en este apartado, vamos explicar algunas de las pruebas de funcionamiento y monitorización que se han realizado sobre los agentes. Estas han ayudado a ver de forma rápida el consumo de recursos durante ejecución. Usando la herramienta JDK Mission Control hemos visto la cantidad de memoria que consume un agente o los hilos que va creando, también nos ha ayudado a resolver un problema de consumo excesivo de CPU.

En la primera prueba de monitorización se vio que al arrancar el agente empezaba a consumir CPU y memoria de forma desproporcionada, pero no se sabía por qué. Si nos fijamos en la imagen de la Figura 22, se ve claramente que un hilo llamado `boson-worker-nio-executor-1` está usando demasiada CPU y memoria, un 8.31% y 8.39 GiB, tan solo en el arranque. A través de la pila de

llamadas del hilo y acotando, se pudo ver que el problema estaba en el bucle de ejecución de `NioWorkerChannelExecutor`, la multiplexación de canales no estaba bien implementada y el selector nunca bloqueaba a la espera de eventos de E/S, se quedaba ejecutando el bucle sin parar, eso explica el alto consumo de CPU.

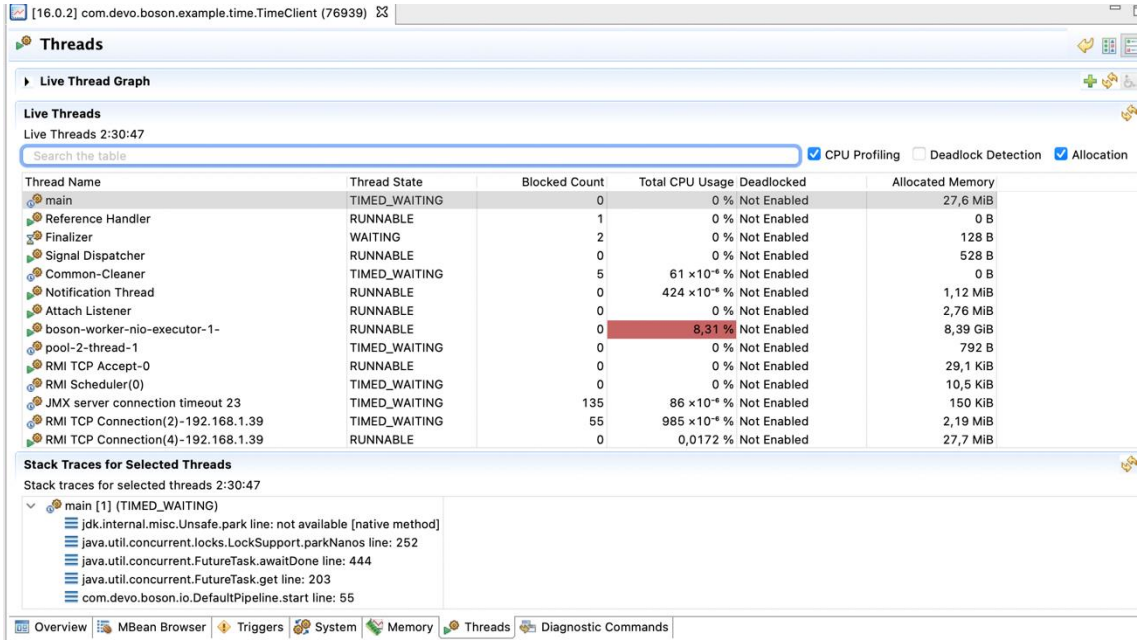

Figura 22. Monitorización de los hilos de un agente usando JDK Mission Control.

Para agentes pasivos también ha sido de gran utilidad esta vista, ya que nos permite ver de un vistazo si se ha creado el hilo Acceptor y cuántos Worker están procesando conexiones, como en la Figura 23, donde vemos que se ha creado un `boson-acceptor-nio-executor-3` y un `boson-worker-nio-executor-1`. Estos hacen referencia a un agente pasivo con un hilo en el canal de entrada.

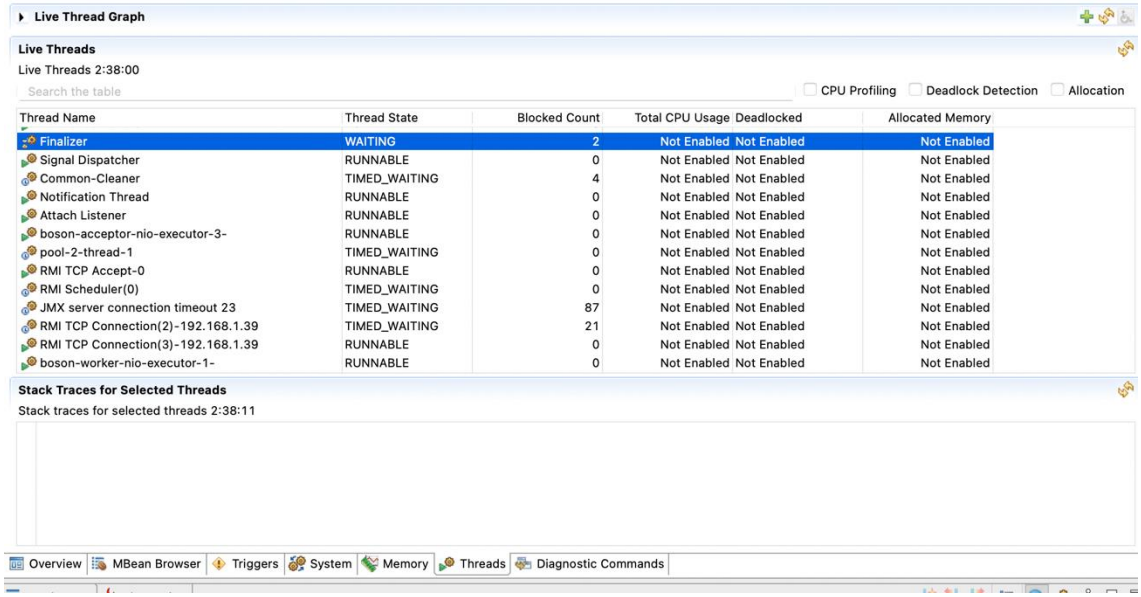

Figura 23. Monitorización de hilos de un agente pasivo.

Es importante remarcar que el framework crea sus propios ThreadFactory [60] para asignar nombres intuitivos y fáciles de reconocer en la búsqueda de errores. Por ese motivo, se hace aún más fácil, usar Mission Control.

6

Conclusiones y Líneas Futuras

6.1 Conclusiones

Se han llevado a cabo todas las fases del proceso de desarrollo de un framework de agentes de transmisión de datos. Hemos visto que el proceso de desarrollar aplicaciones a bajo nivel requiere de una base fuerte de conocimientos para no perderse por el camino. El aprendizaje no es una tarea que se consiga de forma instantánea, requiere de tiempo y dedicación para moldear nuestro cerebro.

Se han analizado otras tecnologías y empresas de la competencia y se ha visto que se enfrentan a los mismos retos, pero desde diferentes puntos de vista, planteando diferentes soluciones para satisfacer las necesidades de los clientes.

En una década la tecnología ha avanzado a pasos agigantados. Los modelos de E/S que se usaban para diseñar servidores con grandes capacidades de concurrencia, se quedaron obsoletos y necesitaron adaptarse a los nuevos modelos que los lenguajes de programación iban proporcionando conforme los sistemas operativos evolucionaban. Al fin y al cabo, un ordenador es un sistema de

múltiples piezas interconectadas para realizar una función de forma conjunta; pocas pueden trabajar de manera independiente. Sacar el máximo potencial de una aplicación requiere exprimir cada una de estas piezas.

También hemos comprobado que la ingeniería del software es muy importante a la hora de desarrollar código, pero sobre todo al desarrollar una librería común como es un framework. Los conceptos de abstracción, encapsulación, polimorfismo o composición nos permiten crear librerías de código que abstraigan la complejidad de los problemas o faciliten su uso y reutilización, como hemos visto en la forma de reutilizar filtros de procesamiento. Los patrones de diseño existen porque existen los lenguajes y paradigmas de programación. Cada patrón se enfoca en resolver un problema específico de la mejor forma posible.

El desarrollo de este framework se ha enfocado en tomar las buenas prácticas de diseño de otras tecnologías para construir una base de código sólida, que facilite el desarrollo de agentes de datos y se pueda extender y mantener de forma sostenible. La toma de decisiones de diseño no siempre es acertada, pero fundamentarla en casos prácticos conocidos es la base para no partir de cero frente a las tecnologías ya maduras. El diseño de un hilo para múltiples canales es un ejemplo claro de simplificación frente a los posibles problemas de concurrencia.

Por último, hemos aprendido que el uso de herramientas para probar y monitorizar nuestro código es tan importante como las fases anteriores, porque permite adelantar problemas que de otra forma acabaríamos encontrando en producción.

6.2 Líneas Futuras

Este desarrollo no es más que el primer grano de arena de una materia que tiene mucho recorrido. A continuación se detallan algunas líneas de desarrollo que podrían ser estudiadas como trabajos futuros:

Múltiples destinos de datos: Por simplificación, en la primera versión del framework se ha decidido que los agentes solo pudiesen tener un canal de salida. Como análisis futuro, sería muy útil que los agentes puedan tener un motor de enrutamiento basado en múltiples reglas que decidan como son guiados los datos por las tuberías desde el canal de entrada a un canal de salida.

Soporte de protocolos de manera nativa: La variedad de protocolos existentes es enorme, pero sería muy interesante que nativamente Boson ofreciera capas de protocolos de uso muy común preparadas para ser reutilizadas en el desarrollo de agentes, evitando así la repetición de código.

E/S asíncrona: Este tipo de modelo de E/S aún no tiene un uso muy extendido, en gran parte por el soporte de los sistemas operativos, aunque se hace interesante analizar en profundidad cómo poder aprovechar sus capacidades en el diseño de agentes pasivos mega concurrentes.

Referencias

[1] *Devo: Cloud-Native Logging, SIEM, Security Analytics & AIOps*. [En línea] Disponible en: https://www.devo.com/ [Accedido 25 Septiembre 2021].

[2] *Sending data to Devo*. [En línea] Disponible en: https://docs.devo.com/confluence/ndt/v7.1.0/sending-data-to-devo [Accedido 25 Septiembre 2021].

[3] *Devo Relay*. [En línea] Disponible en: https://docs.devo.com/confluence/ndt/latest/sending-data-to-devo/devo-relay [Accedido 12 Septiembre 2021].

[4] *The Syslog Protocol*. [En línea] Disponible en: https://datatracker.ietf.org/doc/html/rfc5424 [Accedido 15 Septiembre 2021].

[5] Santos-González, I., Rivero-García, A., Molina-Gil, J. and Caballero-Gil, P., 2017. *Implementation and Analysis of Real-Time Streaming Protocols*. [En línea] Disponible en: https://www.researchgate.net/publication/316944787_Implementation_and_Analysis_of_Real-Time_Streaming_Protocols [Accedido 25 Septiembre 2021].

[6] *JMC 8.1.0 GA Release*. [En línea] Disponible en: https://jdk.java.net/jmc/8/ [Accedido 25 Septiembre 2021]

[7] Gradle Build Tool. [En línea] Disponible en: https://gradle.org/ [Accedido 21 Septiembre 2021].

[8] *Code Quality and Code Security | SonarQube*. [En línea] Disponible en: https://www.sonarqube.org/ [Accedido 25 Septiembre 2021].

[9] *Splunk | Turn Data into Doing*. [En línea] Disponible en: https://www.splunk.com/ [Accedido 20 Septiembre 2021].

[10] *Elastic*. [En línea] Disponible en: https://www.elastic.co/es/ [Accedido 1 Septiembre 2021].

[11] *Universal Forwarder for Remote Data Collection | Splunk*. [En línea] Disponible en: https://www.splunk.com/en_us/download/universal-forwarder.html [Accedido 5 Septiembre 2021].

[12] *Install an add-on in a distributed Splunk Enterprise deployment - Splunk Documentation*. [En línea] Disponible en: https://docs.splunk.com/Documentation/AddOns/released/Overview/Distributedinstall [Accedido 2 Septiembre 2021].

[13] *¿What is Elasticsearch?*. [En línea] Disponible en: https://www.elastic.co/es/what-is/elasticsearch [Accedido 25 Septiembre 2021].

[14] *Logstash: Recopila, parsea y transforma logs | Elastic*. [En línea] Disponible en: https://www.elastic.co/es/logstash/ [Accedido 25 Septiembre 2021].

[15] *Beats: Agentes de datos para Elasticsearch | Elastic*. [En línea] Disponible en: https://www.elastic.co/es/beats/ [Accedido 25 Septiembre 2021].

[16] *Meet the New Logstash Java Execution Engine*. [En línea] Disponible en: https://www.elastic.co/es/blog/meet-the-new-logstash-java-execution-engine [Accedido 25 Septiembre 2021].

[17] *Libbeat – Framework for building the Beats*. [En línea] Disponible en: https://github.com/elastic/beats/tree/master/libbeat [Accedido 25 Septiembre 2021].

[18] *Netty*. [En línea] Disponible en: http://netty.io/ [Accedido 25 Septiembre 2021].

[19] Maurer, N. and Wolfthal, M., 2016. *Netty in action*. Shelter Island, NY: Manning Publications.

[20] *Apache Tomcat* [En línea] Disponible en: http://tomcat.apache.org/ [Accedido 25 Septiembre 2021].

[21] Laurie, B. and Laurie, P., 2003. *Apache: The Definitive Guuide*. Beijing: O'Reilly & Associates.

[22] *Project Grizzly*. [En línea] Disponible en: https://javaee.github.io/grizzly/ [Accedido 25 Septiembre 2021].

[23] *Control Groups — The Linux Kernel documentation*. [En línea] Disponible en: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html [Accedido 25 Septiembre 2021].

[24] *Empowering App Development for Developers | Docker*. [En línea] Disponible en: https://www.docker.com/ [Accedido 25 Septiembre 2021].

[25] *Swarm mode overview*. [En línea] Disponible en: https://docs.docker.com/engine/swarm/ [Accedido 25 Septiembre 2021].

[26] ¿What is Kubernetes?. [En línea] Disponible en: https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/ [Accedido 25 Septiembre 2021].

[27] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E. and Wilkes, J., 2021. *Large-scale cluster management at Google with Borg*. [En línea] Disponible en: https://research.google/pubs/pub43438/ [Accedido 25 Septiembre 2021].

[28] *What We Announced Today and Why it Matters | Mirantis*. [En línea] Disponible en: https://www.mirantis.com/blog/mirantis-acquires-docker-enterprise-platform-business/ [Accedido 25 Septiembre 2021].

[29] *k3s-io/k3s: Lightweight Kubernetes*. [En línea] Disponible en: https://github.com/k3s-io/k3s [Accedido 25 Septiembre 2021].

[30] Wiegers, K., 2003. *Software Requirements 2: Practical Techniques for Gathering and Managing Requirements Throughout the Product Development Cycle*. Redmond, Wash.: Microsoft Press.

[31] Beedle, M., Devos, M., Sharon, Y., Schwaber, K. and Sutherland, J., 2021. *SCRUM: An extension pattern language for hyperproductive software development*. Disponible en: http://jeffsutherland.org/scrum/scrum_plop.pdf [Accedido 25 Septiembre 2021].

[32] *Manifesto for Agile Software Development*. [En línea] Disponible en: https://agilemanifesto.org/ [Accedido 9 Septiembre 2021].

[33] TCP/IP David Clark (1997). Interoperation, Open Interfaces, and Protocol Architecture. The Unpredictable Certainty: White Papers. National Research Council. ISBN 9780309060363. Clark, D., 2021. The Unpredictable Certainty.

[34] P1003.1 - Standard for Information Technology--Portable Operating System Interface (POSIX) Base Specifications, Issue 8. [En línea] Disponible en: https://standards.ieee.org/project/1003_1.html#Standard [Accedido 25 Septiembre 2021].

[35] *Berkeley sockets*. [En línea] Disponible en: https://en.wikipedia.org/wiki/Berkeley_sockets [Accedido 5 Septiembre 2021].

[36] Stevens, W., Rudoff, A. and Fenner, B., 2003. *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*. Addison-Wesley Professional.

[37] *The Apache Tomcat 5.5 Servlet/JSP Container - Changelog*. [En línea] Disponible en: https://tomcat.apache.org/tomcat-5.5-doc/changelog.html [Accedido 12 Septiembre 2021].

[38] *The Architecture of Open Source Applications (Volume 2): nginx*. [En línea] Disponible en: http://www.aosabook.org/en/nginx.html [Accedido 15 Septiembre 2021].

[39] *Efficient data transfer through zero copy*. [En línea] Disponible en: https://developer.ibm.com/articles/j-zerocopy/ [Accedido 16 Septiembre 2021].

[40] *kqueue, kevent -- kernel event notification mechanism*. [En línea] Disponible en: https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2 [Accedido 23 Septiembre 2021].

[41] *Kernel Space Definition*. [En línea] Disponible en: http://www.linfo.org/kernel_space.html [Accedido 20 Septiembre 2021].

[42] *java.nio (Java SE 16 & JDK 16)*. [En línea] Disponible en: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/nio/package-summary.html [Accedido 21 Septiembre 2021].

[43] *Memory Management in the Java HotSpot Virtual Machine*. [En línea] Disponible en: https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf [Accedido 5 Septiembre 2021].

[44] Mastrangelo, L., Ponzanelli, L., Mocci, A., Lanza, M., Hauswirth, M. and Nystrom, N., 2021. *Use at Your Own Risk: The Java Unsafe API in the Wild*. Faculty of Informatics, Universita della Svizzera italiana.

[45] *ByteBuffer (Java SE 16 & JDK 16)*. [En línea] Disponible en: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/nio/ByteBuffer.html [Accedido 15 Septiembre 2021].

[46] Corbet, J., Rubini, A. and Kroah-Hartman, G. ed., 2005. Memory Mapping and DMA. In: *Linux Device Drivers*, 3rd ed. [En línea] Disponible en: https://www.oreilly.com/openbook/linuxdrive3/book/ch15.pdf [Accedido 2 Septiembre 2021].

[47] *JEP 393: Foreign-Memory Access API (Third Incubator)*. [En línea] Disponible en: https://openjdk.java.net/jeps/393 [Accedido 12 Septiembre 2021].

[48] Robert C. Martin Martin, R., 2002. *Agile software development, principles, patterns, and practices*.

[49] *SocketAddress (Java SE 16 & JDK 16)*. [En línea] Disponible en: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/net/SocketAddress.html [Accedido 2 Julio 2021].

[50] Lea, D., *Scalable IO in Java*. [En línea] Disponible en: http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf

[51] *AsynchronousChannelGroup (Java SE 16 & JDK 16)*. [En línea] Disponible en: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/nio/channels/AsynchronousChannelGroup.html [Accedido 2 Septiembre 2021].

[52] *ExecutorService (Java SE 16 & JDK 16).* [En línea] Disponible en: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ExecutorService.html [Accedido 1 Septiembre 2021].

[53] *Round-robin scheduling.* [En línea] Disponible en: https://en.wikipedia.org/wiki/Round-robin_scheduling [Accedido 19 Septiembre 2021].

[54] *Core J2EE Patterns - Intercepting Filter.* [En línea] Disponible en: https://www.oracle.com/java/technologies/intercepting-filter.html [Accedido 2 Septiembre 2021].

[55] *FluentInterface.* [En línea] Disponible en: https://martinfowler.com/bliki/FluentInterface.html [Accedido 10 Septiembre 2021].

[56] *rfc868.* [En línea] Disponible en: https://datatracker.ietf.org/doc/html/rfc868. [Accedido 26 Septiembre 2021].

[57] *rfc1305.* [En línea] Disponible en: https://datatracker.ietf.org/doc/html/rfc1305 [Accedido 26 Septiembre 2021].

[58] *Netty.docs: User guide for 4.x.* [En línea] Disponible en: https://netty.io/wiki/user-guide-for-4.x.html [Accedido 26 Septiembre 2021].

[59] *Redis.* [En línea] Disponible en: https://redis.io/ [Accedido 26 Septiembre 2021].

[60] *ThreadFactory (Java SE 16 & JDK 16).* [En línea] Disponible en: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ThreadFactory.html [Accedido 26 Septiembre 2021].

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA