Universidad de Valladolid

Escuela de Ingeniería Informática

Grado en Ingeniería Informática
Mención De Computación

TRABAJO FIN DE GRADO

Comunicación distribuida de procesos en R mediante un modelo basado en actores

Autor:
Samuel Sjur Bassols Citores

Tutor:
Dr. César Llamas Bello

Resumen

El aumento de las necesidades de cómputo de la comunidad científica y comercial tienen su solución natural en la utilización de plataformas de cómputo distribuido. Uno de los desafíos principales en este tipo de sistemas es garantizar la tolerancia frente a fallos y el relanzamiento de sus componentes frente a caídas. En este trabajo, se pretende reflejar el análisis, diseño y construcción de una plataforma práctica de este tipo, donde el programador-usuario podrá dividir su tarea en partes donde cada una de estas partes se asigna a un nodo de cómputo por el sistema. De esta manera se puede obtener el grado de paralelismo deseado por el usuario.

En concreto se ha creado un sistema distribuido de cómputo que permite que el usuario paralelice y ejecute código escrito en el lenguaje R. El sistema ofrece una asignación transparente y flexible sobre un *pool de cómputo*. Para este sistema se han realizado pruebas sobre una plataforma física frente a varios tipos de cargas de trabajo. Se ha comprobado la tolerancia y robustez del sistema frente a diversas situaciones de rotura-caída y la mejora consistente en el rendimiento con un modelo de varios nodos de cómputo.

Palabras clave: Sistema Distribuido, Elixir, R, resiliencia, tolerancia frente a fallos, pool de cómputo.

Abstract

The increasing computing needs of the scientific and commercial community have their natural solution in the use of distributed computing platforms. One of the main challenges in this type of systems is to guarantee the tolerance to failures and the restart of its components in case of crashes. In this work, we intend to reflect the analysis, design and construction of a practical platform of this type, where the user-programmer could divide his task into parts where each of these parts is assigned to a compute node by the system. In this way, the degree of parallelism desired by the user can be obtained.

Specifically, a distributed computation system has been created that allows the user to parallelize and execute code written in the R language. The system provides a transparent and flexible allocation over a computation pool. For this system, tests have been performed on a physical platform against various types of workloads. The system has been tested for tolerance and robustness to various break-crash situations and for consistent performance improvement with a multi-node compute model.

Key words: Distributed computing, Elixir, R, fault-tolerant system, computing pool.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

Presentación

Un sistema distribuido es un sistema en el que varios componentes se encuentran repartidos a lo largo de una red. Los sistemas distribuidos se caracterizan por la necesidad que tienen de ser resilientes al fallo, ya que distintos componentes pueden dejar de estar disponibles por motivos de conexión. Un uso para los sistemas distribuidos es la computación distribuida, en la que una tarea es divida en partes y cada una de estas partes es computada por un componente del sistema. De esta manera se puede obtener un grado de paralelismo mucho mayor.

El objetivo principal de este Trabajo de Fin de Grado es construir un sistema que pueda aprovechar la potencia y la base de usuarios de la plataforma de cálculo estadístico R en un contexto distribuido donde se puede desglosar el cómputo de algunas tareas especialmente onerosas en el aspecto de tiempo o almacenamiento, en otros nodos que utilizaremos como *nodos de cómputo*.

Al mismo tiempo, este trabajo tiene como finalidad importante, explorar las capacidades de los lenguajes basados en un modelo de actores, para proporcionar robustez y resiliencia a la plataforma de cómputo. Al respecto se ha decidido utilizar Elixir (un lenguaje de programación basado en Erlang OTP), lo que ha ofrecido insospechadas posibilidades al proyecto original.

Estructura de la Memoria

La creación de este documento sigue las pautas establecidas para un documento de un proyecto de Ingeniería Software. La estructura es la siguiente:

En el Capítulo 1 se introduce el proyecto que se va a realizar, explicando con más detalle los aspectos mencionados en el resumen. También se plantean los objetivos del proyecto, así como las tareas y recursos que forman parte del proyecto.

En el Capítulo 2 se realiza la planificación del proyecto, la división de las tareas, la gestión del tiempo, de los recursos y de los riesgos.

En el Capítulo 3 se explica la fundamentación teórica necesaria para entender el resto del proyecto y se plantean los antecedentes tecnológicos que pueden ayudar a la toma de decisiones a lo largo del proyecto, así como paquetes de Software que pueden ser de interés.

El Capítulo 4 corresponde al Análisis Software: Casos de uso, requisitos de diferentes tipos y la explicación en detalle de los casos de uso y sus diagramas de secuencia.

En el Capítulo 5 se realiza el Diseño Software del sistema.

En el Capítulo 6 se describen los resultados de la implementación, explicando algunas de las partes del código del sistema que tienen especial interés en cuanto a su implementación.

En el Capítulo 7 se muestran las pruebas que se han realizado para validar el funcionamiento del sistema, además de las pruebas de rendimiento con distintas cargas y las pruebas de usabilidad del paquete R.

En el Capítulo 8 se plantean las conclusiones extraídas del proyecto y se discuten los resultados, así como las posibles lineas futuras que podrían existir.

En la parte final del documento se muestra la guía de instalación del sistema.

Agradecimientos

Quisiera dar las gracias a mi tutor César Llamas Bello por proponerme un tema para el TFG que ha resultado ser muy interesante. También quiero agradecerle lo mucho que me ha enseñado durante estos meses, su inestimable ayuda y su infinita paciencia.

Agradecer también a mis compañeros de INDAT por su amistad, apoyo y por las fructíferas discusiones que han hecho que todo este tiempo sea mucho más agradable.

También, quisiera dar las gracias a mi familia, por sus ánimos y apoyo incondicional durante toda mi vida.

Capítulo 1

Introducción

1.1. Introducción

Durante los últimos años, el campo del análisis de datos no ha dejado de crecer y no se prevé que deje de hacerlo en un futuro. El análisis de estos datos, conocidos como *Big Data*, cada vez tiene más importancia, ya que se trata de grandes volúmenes de información que se recopilan en una gran variedad de entornos. Con la llegada de internet se ha comenzado a almacenar una gran cantidad de datos sobre los internautas, cuyo análisis ha pasado a ser inabarcable con metodologías estadísticas tradicionales. Además, en todo tipo de procesos industriales, trabajos de investigación o análisis sociológicos, cada vez se recogen volúmenes mayores de información que necesita ser analizada. Todo esto requiere de tecnologías adecuadas para la captura, almacenamiento, distribución y análisis de la información.

El auge de servicios como *Google Cloud*, *Amazon Web Services* o *Micosoft Azure* pone de manifiesto la increíble demanda que hay para almacenar toda esta información. En la Figura 1.1 se puede ver el crecimiento del volumen de datos producidos a diario a nivel mundial y cómo se espera que crezca en el próximo año.

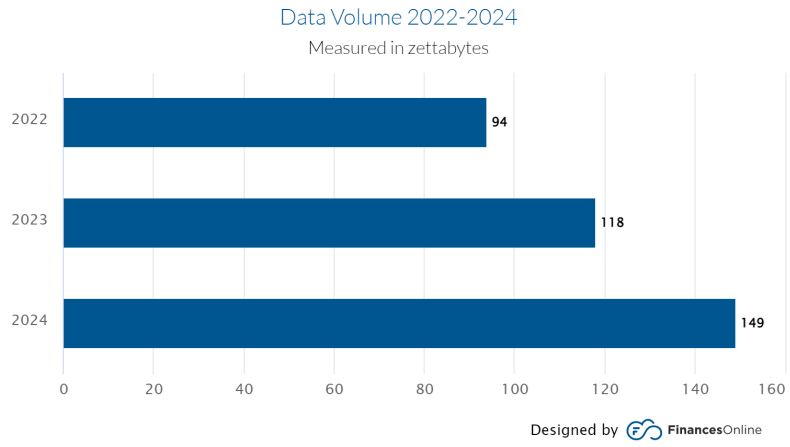

Figura 1.1: Evolución de la cantidad de datos generados a diario en *zettabytes* [1]

Esta gran cantidad de datos ha llevado no solo a la necesidad de tener cada vez servidores más extensos,

sino que se han desarrollado nuevas tecnologías para poder analizar estos datos. Uno de los campos con mayor interés en este aspecto es la computación paralela [2]. Una de las ramas principales de la computación paralela es el desarrollo de nuevos algoritmos que puedan beneficiarse de tareas que se pueden ejecutar en paralelo. Una vez se ha encontrado una manera de paralelizar las tareas, es necesario desarrollar un sistema que las pueda llevar a cabo. Un ejemplo de un sistema que realiza computaciones paralelas es un procesador multinúcleo, en el que cada uno de estos núcleos puede realizar una operación independiente. Sin embargo, cuando se trabaja con volúmenes de datos como los ilustrados anteriormente, cobra interés la idea de realizar estas computaciones en varias máquinas, de tal manera que cada una realice una tarea pequeña y sencilla que posteriormente se combine en el resultado final.

La computación distribuida es el campo de la ingeniería informática que estudia los sistemas distribuidos. Un sistema distribuido es un sistema en el que los distintos componentes se encuentran en distintos puntos de la red. Estos componentes o nodos se comunican entre ellos mediante mensajes. Por tanto, se puede dividir una computación entre los varios nodos de una red de tal manera que cada uno realice una operación abarcable a sus capacidades de computación [3].

Uno de los lenguajes con más relevancia en el campo de los sistemas distribuidos es *Erlang* [4]. *Erlang* es un lenguaje funcional que se usa para construir sistemas cuyas principales propiedades son:

- **Distribuido y Escalable**: El código Erlang se ejecuta en procesos ligeros que están aislados y se comunican mediante mensajes, lo que permite añadir nuevos procesos sin afectar el funcionamiento del resto de procesos. Estos procesos se ejecutan en una máquina virtual *Erlang VM* y representan actores en un sistema.
- **Tolerante a los fallos**: Si algún componente del sistema falla, es posible recuperarse de este fallo sin poner en riesgo el funcionamiento del sistema completo.
- **Hot Swapping** (Actualización de código en caliente): Aplicaciones que pueden ser modificadas sin detener el sistema, lo que permite realizar modificaciones en el código o en la arquitectura sin necesidad de detener por completo el sistema.

Para lograr estás propiedades, *Erlang* suele ir acompañado de OTP (*Open Telecom Platform*), una serie de librerías y principios de diseño que permiten desarrollar estos sistemas. Por ejemplo OTP incluye bases de datos distribuidas como ETS (*Erlang Term Storage*) o comportamientos para servidores como puede ser *GenServer*, un servidor génerico, o *Supervisor*, para un supervisor. Hoy en día *Erlang* se usa a nivel profesional especialemente en servicios distribuidos para dar soporte a aplicaciones de mensajería instantánea, como *WhatsApp*.

Elixir [5] es un lenguaje de programación que se ejecuta sobre la VM de *Erlang* y por tanto integra las funcionalidades y propiedades de *Erlang* y aporta nuevas herramientas para realizar aplicaciones distribuidas.

Un sistema de computación distribuida tiene que permitir además realizar análisis mediante las herramientas habituales para un experto. En general existen dos lenguajes principales con los que se trabaja en el análisis de datos, R y *Python*:

- **R** es un lenguaje de programación interpretado diseñado para trabajar con distintas estructuras de datos. R está diseñado desde un punto de vista estadístico y por tanto se especializa en trabajar con tablas e implementar operaciones eficientes sobre estas.
- ***Python*** es un lenguaje orientado a objetos utilizado en una gran variedad de campos. En los últimos años ha ganado relevancia en el aspecto del análisis de datos y especialmente en el campo de la redes neuronales.

En este proyecto se propone y se implementa un sistema distribuido tolerante al fallo funcionando sobre *Elixir* que permite distribuir una computación paralela en R entre una serie de nodos que calculan los resultados parciales y finalmente se reconstruye el resultado final. La principal característica del sistema distribuido que se quiere desarrollar en este proyecto es que cada uno de los nodos del sistema debe tener el mismo funcionamiento, y por tanto, el mismo código. Esto permite que se puedan añadir nodos nuevos a la red sin necesidad de modificar el código.

1.2. Motivación

La motivación principal de este proyecto es adentrarme en el conocimiento de nuevos lenguajes funcionales y tecnologías no estudiadas a lo largo del grado y ser capaz de utilizarlos para resolver problemas actuales como el análisis masivo de datos, mediante la computación distribuida.

Este proyecto implica aprender los lenguajes de programación *Elixir y Erlang*, aprender a utilizarlos para crear un sistema distribuido resiliente al fallo y la manera de integrarlos con otros lenguajes. Se aborda también el tema de cómo realizar comunicaciones a través de la red para las distintas partes del sistema y de cómo desarrollar un proyecto software en R. Además, la puesta en marcha del sistema en **Linux** supone aprender sobre cómo establecer un servidor web y un servidor SFTP.

Además, la mayoría de lenguajes utilizados a lo largo del grado siguen el paradigma *Object Oriented Programming*, lo que hace que se desconozca y resulte difícil cambiar a otros paradigmas. Realizar proyectos fuera de este paradigma permite obtener una nueva perspectiva de cómo afrontar distintos problemas.

Otra importante motivación, es la relevancia que han ido adquiriendo estas tecnologías poco exploradas en el grado y la importancia que puede tener para el futuro laboral conocerlas.

Y por último, siempre existe la motivación de obtener un sistema que sea útil para profesionales del sector, y que puedan, a partir de este, crear sistemas que se ajusten a sus cargas de trabajo.

1.3. Objetivos y etapas del proyecto

1.3.1. Objetivos

El objetivo general de este proyecto es el desarrollo de un sistema distribuido resiliente y tolerante al fallo que permita dividir una operación computacionalmente costosa en R y repartir la carga entre cada uno de los diferentes nodos del sistema. Además, este sistema debe informar al usuario de la carga en cada nodo y los distintos eventos que ocurren en estos. Los objetivos específicos son:

- Obtener los conocimientos necesarios para poder utilizar el lenguaje de programación *Elixir* y usar sus prestaciones para desarrollar un sistema distribuido.
- Desarrollar un paquete R que ofrezca al usuario una serie de operaciones, fraccione estas operaciones, las envié, y, una vez realizadas, reconstruya los resultados y se los muestre al usuario.
- Diseñar y desarrollar un sistema distribuido en *Elixir* en el que cada nodo pueda realizar una computación y que en caso de fallar, esta sea retomada por otro nodo de manera transparente al usuario del sistema.
- Desarrollar un servidor web que ofrezca una web al usuario en la que se muestre información del funcionamiento del sistema.
- Validar la eficiencia del sistema sobre distintos cálculos intensivos.
- Estudiar la intuitividad del uso del paquete R para un usuario frecuente de R.

Por tanto, este proyecto se va a dividir en varias etapas en las que en cada una se van a abordar unas tareas específicas.

1.3.2. Etapas

Las etapas de desarrollo y resolución de los mencionados objetivos son las siguientes:

Primera Etapa: Estudio del lenguaje Elixir

Antes de poder abordar el diseño e implementación del sistema distribuido se deben adquirir los conocimientos de programación necesarios. Para esta etapa se usan varios libros y tutoriales [6] [7] [8].

Tareas a realizar:

- Estudio del funcionamiento básico del lenguaje *Elixir*.
- Estudio de las prestaciones que ofrece el lenguaje para sistemas distribuidos.
- Estudio de las distintas formas que se plantean para realizar un sistema tolerante al fallo.
- Estudio del *Framework* usado para el servidor web.

Segunda etapa: Paquete R (RPack)

Se debe diseñar e implementar un paquete software en el que se ofrezcan unas operaciones, procese los datos que le proporciona el usuario, se conecte con el sistema distribuido, y muestre al usuario los resultados de la operación.

Tareas a realizar:

- Estudio de los requisitos software.
- Estudio de paquetes en R equivalentes.
- Diseño del paquete.
- Preparación e instalación del entorno de programación y carga del software.
- Implementación de la aplicación a partir del diseño establecido.

Tercera etapa: Sistema distribuido (RNodePool): lógica de la aplicación

Se debe diseñar e implementar un sistema distribuido que sea capaz de recuperase ante los fallos y que ejecute una computación en el lenguaje R. Por tanto, se debe desarrollar tanto el sistema distribuido como la integración de este con el lenguaje R.

Tareas a realizar:

- Estudio de los requisitos software.
- Estudio de aplicaciones equivalentes.
- Diseño del sistema distribuido de manera acorde a los requisitos.
- Preparación e instalación del entorno de programación y carga del software.
- Integración del sistema distribuido con el lenguaje R.
- Implementación del sistema distribuido a partir del diseño establecido.

Cuarta etapa: Servidor Web (RHub), Sistema distribuido (RNodePool), vistas y controladores

Se debe diseñar e implementar un Servidor Web, la vista y los controladores que permita recibir las peticiones que le llegan desde el paquete R y poner en funcionamiento el sistema distribuido en función de lo que se solicite. Además, este debe ofrecer una web en la que muestre el funcionamiento del sistema.

Tareas a realizar:

- Estudio de los requisitos software.

- Estudio de aplicaciones equivalentes.
- Diseño del servidor de manera acorde a los requisitos.
- Preparación e instalación del entorno de programación y carga del software.
- Implementación del servidor a partir del diseño establecido.
- Implementación de la web para mostrar el funcionamiento.

Quinta etapa: Integración y pruebas

Por último, se integran las tres partes desarrolladas y se pone en funcionamiento el sistema. Se realizan pruebas para comprobar que el sistema funciona de manera cohesiva. Además se realizan pruebas para comprobar el rendimiento del sistema y su intuitividad para un programador usual de R.

Tareas a realizar:

- Pruebas y comprobación del correcto funcionamiento del sistema.
- Estudios sobre el rendimiento del sistema completo ante varias cargas de trabajo.
- Estudio de la usabilidad del paquete R para un programador.

1.4. Recursos utilizados

Para el desarrollo de este proyecto y la puesta en marcha del sistema se han usado los siguientes recursos físicos:

- Ordenador Personal con instalación de Ubuntu 22.04.2.
- Tres Ordenadores *minipc*.
- Tres fuentes de alimentación.
- Switch Gigabit.
- KVM Switch.
- Teclado.
- Ratón.
- Cinco cables de red.
- Regleta eléctrica.
- Monitor.
- Memoria USB.

1.5. Caso de Negocio

1.5.1. Agentes implicados en el proyecto

Los siguientes actores están implicados en el proyecto:

- **Cliente**: Usuario del sistema distribuido y administrador del sistema.
- **Beneficiario**: Los usuarios del sistema.
- **Usuario del proyecto**: Programador que ejecuta una instancia de R con el paquete desarrollado y administrador del sistema distribuido.

1.5.2. Presupuesto

En una primera aproximación al coste estimado de este proyecto, se han de tener en cuenta tanto factores de costes de los materiales físicos usados en el proyecto, como las horas de trabajo del desarrollador. El desglose y detalle de estos costes se puede consultar en el capítulo 3.

1.5.3. Impacto

Las prestaciones que ofrece este sistema de computación distribuida permiten realizar análisis de datos que no son posibles con un solo dispositivo. El principal impacto de este sistema es económico. Además, con este sistema las organizaciones pueden llevar a cabo análisis de datos que hasta el momento no se podían realizar sin contratar un servicio externo o comprando un dispositivo más potente. Este sistema permite combinar la potencia de dispositivos de bajo rendimiento, que de otra manera no tendrían utilidad analizando datos.

Capítulo 2

Planificación del Proyecto

En este capítulo se lleva a cabo la planificación del proyecto siguiendo la guía de PMBOK [9]. La elaboración del sistema definido en los objetivos se realiza mediante un proceso iterativo, ya que se puede dividir fácilmente en etapas bien definidas.

En primer lugar, se realiza una planificación inicial con el diseño de las tareas a partir de las etapas del proyecto definidas en la sección 1.3.2, en la que además se definen los hitos y por tanto los entregables. Posteriormente, mediante un cronograma se muestra la gestión del tiempo. También se realiza la gestión de los riesgos que pueden aparecer a lo largo del proyecto y por último se realiza una estimación de los costes materiales y laborales.

2.1. Planificación inicial

En este proyecto, debido a la clara división de las tareas en etapas bien diferenciadas, el desarrollo más adecuado para este tipo de proyecto es un **desarrollo en cascada**, donde al final de cada etapa se finaliza un entregable. Esto facilita la división del trabajo en cuanto al análisis y diseño, en partes independientes. El proceso es lineal, no se avanza a otra etapa hasta que se completa la etapa anterior, por lo tanto, se debe definir un entregable que sea la integración de esas etapas independientes.

Diseño de tareas

Se definen estas tareas a partir de las etapas descritas en la sección 1.3.2. Estas etapas son inferidas de los objetivos propuestos, por lo que se deben concretar un conjunto de tareas que engloben estas etapas y el entregable que produce cada una de ellas. Se definen las siguientes tareas a partir de las etapas del proyecto:

- **Preparación**. En esta fase inicial se concentran todas las subtareas relacionadas con la preparación del proyecto. Debido a la naturaleza de este proyecto, en el que se utilizan tecnologías de las que se tienen pocos conocimientos previos, se prevé que esta etapa sea más extensa de lo habitual en un proyecto software.

- **Desarrollo del componente Rpack**: Se analizan los requisitos de este componente y se realiza un diseño apropiado. Para esto, se estudian otros paquetes que puedan ser similares al que se prevé desarrollar. Una vez se ha realizado el diseño del componente, este es implementado produciendo el **entregable 1**.

- **Desarrollo de la lógica de la aplicación de RNodePool**: En esta tarea se analizan los requisitos pertinentes al funcionamiento de la lógica de la aplicación distribuida. Para definir estos requisitos se deben primero estudiar diseños similares para poder entender qué se espera de este tipo de sistemas. Al igual que en la tarea anterior, se realiza el diseño del componente y posteriormente su implementación. Esta tarea es la que más aspectos técnicos presenta, por lo que supone un riesgo para el resto del desarrollo si no se lleva a cabo correctamente. Esta tarea produce el **entregable 2**.

- **Desarrollo de las vistas y controladores de RNodePool**: De manera equivalente a las tareas anteriores, se realiza un análisis de los requisitos de este componente y se realiza el desarrollo. Esta tarea produce el **entregable 3**.

- **Desarrollo del servidor web RHub**: En esta tarea se analiza, diseña e implementa el componente **RHub**. Esta tarea produce el **entregable 4**.

- **Integración de las tres partes desarrolladas**: Esta tarea produce el **entregable 5**, que contiene una versión funcional de la aplicación y las distintas baterías de pruebas de funcionamiento y rendimiento a la que se somete a la aplicación completa.

- **Documentación**: en esta tarea se documentan las distintas partes del proyecto, creando manuales y guias de uso para los componentes cuando sea necesario. Esta tarea es global a lo largo del proyecto ya que cada una de las tareas incluye la creación de la documentación. Esta tarea incluye también el desarrollo de la memoria. Una vez se han finalizado el resto de tareas, termina esta tarea y se produce el entregable final del proyecto.

En la Tabla 2.1 aparecen fijados los hitos del proyecto. En cada uno de ellos, se debe generar el entregable correspondiente.

Entregable	Fecha
1	31/03/2023
2	30/04/2023
3	15/05/2023
4	31/05/2023
5	15/06/2023
Finalización del proyecto	15/06/2023

Tabla 2.1: Fechas en las que se deben producir los entregables

2.2. Gestión del tiempo

Para llevar a cabo la gestión del tiempo se ha utilizado el programa gratuito Gantt-Project. En la Figura 2.1 se muestra el diagrama de Gantt de las actividades que se han realizado en este proyecto.

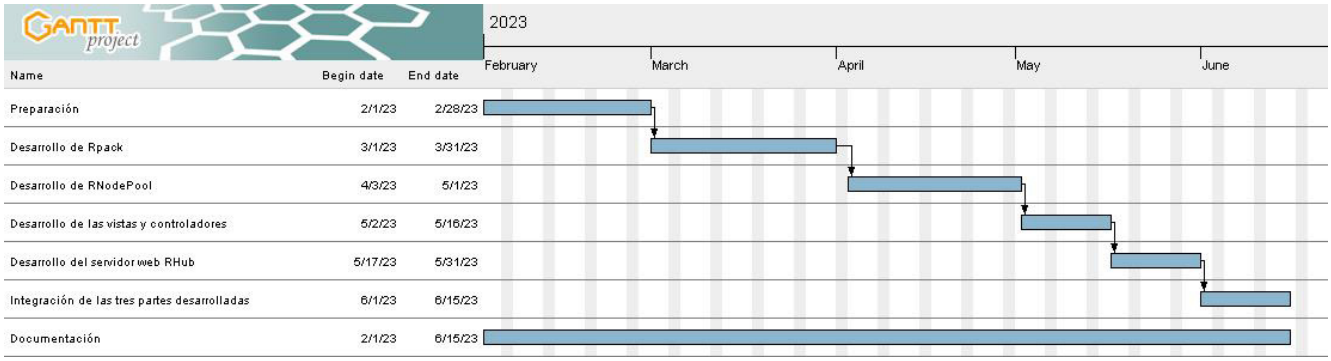

Figura 2.1: Diagrama de Gannt de actividades

2.3. Gestión de riesgos

Como en todo proyecto se debe realizar una identificación de los principales riesgos que se pueden producir durante el desarrollo de este proyecto. Se debe además, llevar a cabo un análisis cuantitativo de estos y plantear los posibles planes de actuación en caso de que se den los riesgos.

Los riesgos identificados comprenden aquellos que son comunes a las distintas partes del desarrollo, así como los que son más específicos de un estudiante con otras cargas de trabajo simultáneas al desarrollo del proyecto. En la Tabla 2.2 se identifican los riesgos y a continuación se destacan algunos de estos por el problema que pueden suponer para el desarrollo del proyecto.

- *Planificación*: en este proyecto se usan tecnologías poco exploradas anteriormente en el grado, por lo que las estimaciones que se hagan acerca de las duraciones y los costes de algunas partes pueden ser imprecisas.

Nº	Riesgo
1	Planificación poco precisa
2	Uso de tecnologías desconocidas
3	Problemas para llevar a cabo la implementación a partir del diseño realizado
4	Problemas de análisis de requisitos que no se detectan hasta que se pone el sistema completo en funcionamiento
5	Problemas para realizar la instalación del sistema y probar su funcionamiento
6	Menor disponibilidad de la que se había propuesto
7	Fecha de finalización posterior a la prevista

Tabla 2.2: Riesgos del proyecto

- *Problemas para llevar a cabo la implementación a partir del diseño realizado*: Es posible que al realizar el diseño de alto nivel del sistema, por falta de conocimientos del funcionamiento e implementación posterior, se diseñe un sistema que sea más complicado de los esperado.
- *Problemas para realizar la instalación del sistema y probar su funcionamiento*: En las partes finales del proyecto se debe recopilar todo lo desarrollado en una única maquina y poner en funcionamiento un sistema en el que los componentes se comunican a través de la red y esto puede suponer dificultades no previstas.

Para analizar cuantitativamente los riesgos y estudiar su impacto, se utiliza una matriz de probabilidad e impacto, en la que se representan las probabilidades de que ocurra un riesgo y la amenaza que supone para la finalización exitosa del proyecto. En la Tabla 2.3 se muestra esta matriz. Se han definido 3 categorías de riesgo: bajo, medio y alto.

Probabilidad	Amenazas				
0.90	0.05	0.09	0.18	0.36	0.72
0.70	0.04	0.07	0.15	0.28	0.56
0.50	0.03	0.05	0.1	0.2	0.4
0.30	0.02	0.03	0.06	0.12	0.28
0.10	0.005	0.01	0.02	0.04	0.08
	Muy Baja 0.05	Baja 0.1	Moderada 0.2	Alta 0.4	Muy Alta 0.8

Tabla 2.3: Matriz de probabilidades e Impacto
Los colores indican la categoría de riesgo. Verde (bajo), amarillo (medio) y rojo (alto).

Los riesgos del proyecto y sus calificaciones se muestran en la Tabla 2.4. Como se puede observar los riesgos 3 y 5 pueden suponer una amenaza alta para el proyecto, por lo que habrá que prestarles especial atención.

Riesgo	Probabilidad	Impacto	Calificación
1	0.8	0.3	0.24
2	0.7	0.2	0.14
3	0.5	0.7	0.35
4	0.8	0.3	0.24
5	0.5	0.9	0.45
6	0.8	0.1	0.08
7	0.5	0.5	0.25

Tabla 2.4: Calificación de los riesgos

A continuación se muestra el plan de actuación para cada riesgo en caso de que ocurra.

1. Realizar una replanificación del proyecto teniendo en cuenta los errores que hubiera en la primera planificación.

2. Asignar más tiempo al estudio de estas tecnologías a lo largo de todo el proyecto.
3. Asignar mayores recursos y realizar de nuevo el diseño con los conocimientos adquiridos en la implementación.
4. Realizar modificaciones en el sistema en la medida de lo posible para acomodar estos requisitos y si no es posible, descartar los requisitos o plantearlos de otra manera.
5. Asignar más tiempo a trabajar en la configuración del sistema, probando varias posibilidades hasta que este esté en funcionamiento. En caso de que esto no sea posible, desplegar la aplicación solo en un dispositivo.
6. Aumentar el tiempo asignado al proyecto en la medida de lo posible.
7. Incrementar la fecha de entrega hasta su máximo y en caso de que no sea posible, retrasar la fecha de entrega a la convocatoria extraordinaria.

2.4. Gestión de costes

- Costes de Componentes Físicos: Se ha realizado el proyecto en un ordenador personal con un coste aproximado de 1000€. Además se deben considerar los costes de los componentes que se usan para el despliegue. Los costes son aproximaciones al coste de un componente con prestaciones similares en el caso de que no esté en producción.
  - Cada *miniPC* tiene un coste aproximado de 100€
  - El coste del *switch Gigabit* es de 30€
  - El coste del *KVM switch* es de 20€
  - El coste del resto de componentes como cables, teclado y ratón se estima en 30€.

  Por tanto el coste material del proyecto se eleva a 1380€ en total, si consideramos que se amortiza completamente durante este proyecto. Si tenemos en cuenta un prorrateo de este costo con un plazo de amortización de materiales de 4 años, bastante habitual, el coste de los materiales supone un gasto de 345€ al año.
- Costes Laborales: Este proyecto que presento como Trabajo Fin de Grado supone 12 ECTS (European Credit Transfer System) de la formación académica del Grado en Ingeniería Informática. Estos ECTS traducidos a horas de trabajo implican una dedicación aproximada de 300 h. El proyecto estaba previsto desarrollarlo en unas 17 semanas, trabajando unas 18 h semanales. El sueldo medio de un ingeniero informático junior es de alrededor de 23150€ [10] por lo que para un proyecto de 300 horas el coste estimado es aproximadamente de 3620€ en costes laborales.

Capítulo 3

Fundamentación Teórica y Antecedentes Tecnológicos

En este capítulo se explica la fundamentación teórica en la que se basa este proyecto, para poder entender las decisiones que se toman en cuanto a qué tecnologías se usan y qué parecido existe entre este proyecto y tecnologías que se han utilizado previamente en problemas similares. Se va a exponer el funcionamiento de R que es relevante en cuanto a operaciones paralelas y distribuidas. Además, se explicarán las partes relevantes de *Elixir*, así como los paquetes y *frameworks* que pueden ser de interés en el desarrollo del sistema propuesto en los objetivos.

3.1. Funcionamiento de R

Como ya se ha mencionado en la introducción, R es un lenguaje interpretado usado especialmente en cálculos estadísticos. Esto ha hecho que R se diseñe en torno al trabajo con datos dimensionales. Por ello, la mayoría de las operaciones en R están definidas para vectores, matrices o *DataFrames*. Un *DataFrame* es una matriz de datos diseñada para indexar por nombres de filas y columnas.

Como en muchos otros lenguajes, R trabaja con lo que se conoce en el mundo matemático como *Functor* [11]. Un *Functor* es un *mapping* entre dos categorías. En el ámbito de la programación esto se refiere a operaciones que transforman datos de un tipo a otro. El ejemplo más conocido de esto es la función *map*, que recibe un tipo de dato que sea iterable, y le aplica a cada elemento una función, devolviendo un dato de la misma dimensión que el dato de entrada en el que se ha aplicado a cada elemento la función. La función *map* tiene la ventaja de que aplica una función a cada elemento por lo que se puede dividir el dato en partes y aplicarle el mismo *map* a cada parte. También se debe conocer una manera de reagrupar estos fragmentos. En la Figura 3.1 se muestra cómo podemos dividir un conjunto de datos en varias partes, aplicarles a cada uno la función F y reconstruir los datos si se conoce la operación que los concatena.

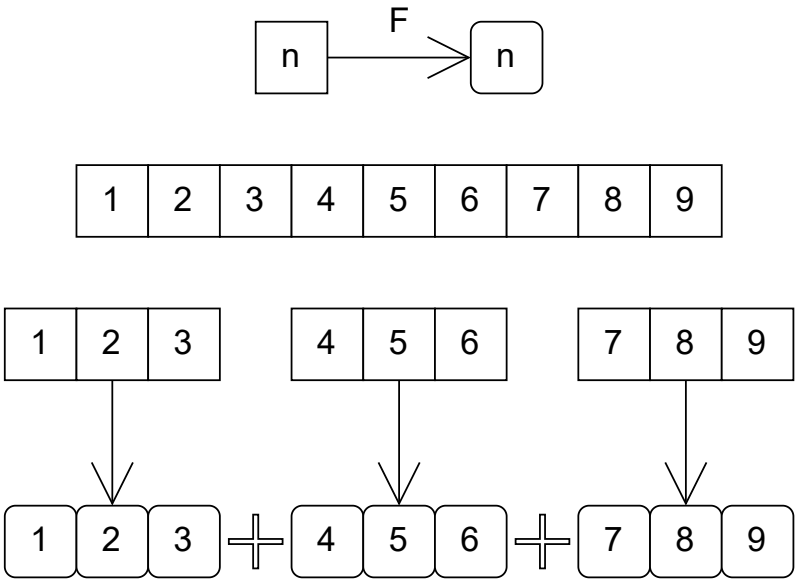

Figura 3.1: Ejemplo de la partición de datos en *map*

En R se ofrecen una serie de operaciones que permiten aplicar una función sobre diferentes conjuntos de datos. A continuación se presentan las funciones de mayor interés y que se van a implementar en este proyecto. Estas funciones reciben más parámetros que los que se muestran a continuación, como por ejemplo parámetros que indican en qué formato se devuelven los datos. Para más información consultar el manual de cada una de estas funciones en `www.rdocumentation.org/`.

- `apply(X,MARGIN,F)`. La función `apply()` recibe como argumento un dato bidimensional y aplica la función F en la dirección que le indica el parámetro `MARGIN`. `MARGIN` puede tomar los valores 1 o 2 o ser un vector (1,2) o (2,1). `MARGIN=1` indica que aplica la función `F` a cada fila de X y `MARGIN=2` indica que se aplica la función `F` a cada columna de X. En el caso de que `MARGIN` sea un vector, este indica que se aplica la función a cada elemento y en qué orden se recorre, primero filas o primero columnas.

- `tapply(X,INDEX,F)`: En R existe un tipo de dato llamado factor. Un factor es una variable que indica la categoría a la que pertenece. De esta manera la función `tapply()` divide los datos en función de la categoría, indicada en el parámetro `Index`, a la que pertenece cada dato y aplica la función `F` a cada conjunto que tiene la misma categoría.

- `lapply(X,F)`: La función `lapply()` aplica a cada elemento de una lista la función `F` y retorna una lista de la misma dimensión.

- `sapply(X,F)`: La función `sapply()` es muy similar a la función `lapply()` ya que también aplica a cada elemento de una lista la función `F`, pero además si es posible, simplifica la lista a un vector o una matriz.

Todas estas operaciones se pueden implementar para que se hagan en ejecuciones paralelas, sin embargo, estas funciones en R no están paralelizadas de ninguna manera. Si se quieren paralelizar las operaciones se puede usar el paquete `parallel` [12] que ofrece estas funciones y otras que se realizan

en paralelo a partir del número de ejecuciones simultáneas que define el usuario. Uno de los objetivos de este proyecto es ofrecer estas operaciones en R pero de manera distribuida, por lo que el paquete `parallel` tiene especial interés en este proyecto.

3.2. Funcionamiento de Elixir

Erlang y *Elixir* son lenguajes funcionales utilizados especialmente en campos de las distribución y comunicación de distintas entidades a través de la red. *Elixir* es una extensión del lenguaje Erlang, es decir, su código se ejecuta de la misma manera que el de *Erlang* y muchas de sus partes están implementadas directamente en código *Erlang*. Como este trabajo tiene como objetivo desarrollar un sistema que sea robusto y distribuido, *Elixir* ofrece una serie de herramientas y facilidades que no existen en otros lenguajes.

Elixir ejecuta su código sobre la máquina virtual *Erlang (Erlang VM)*. Esta MV permite la creación de procesos ligeros que pueden encargarse de distintos aspectos de una aplicación. *Elixir* está diseñado en torno a la creación de una gran cantidad de procesos y es recomendable, siempre que sea posible, crear nuevos procesos para realizar tareas antes que cargar más los procesos ya puestos en funcionamiento.

En *Elixir* los procesos se comunican mediante el paso de mensajes. De esta manera se puede crear un nuevo proceso mediante la función `spawn_link()` que devuelve el PID del proceso y a partir de ahí se pueden mandar mensajes al proceso mediante la función `send()`.

Por tanto estos procesos que se crean en *Elixir* pueden considerarse actores en un sistema distribuido. A lo largo de este documento cuando se hace referencia a procesos *Elixir* estos son a su vez actores en un sistema distribuido ya que lo que define a un actor es la creación de nuevos actores, enviar mensajes a otros actores y recibir mensajes entrantes [13]. En la Figura 3.2 se muestra como un actor o proceso recibe un mensaje que almacena en una bandeja de llegada y que es procesado en función de sus contenidos.

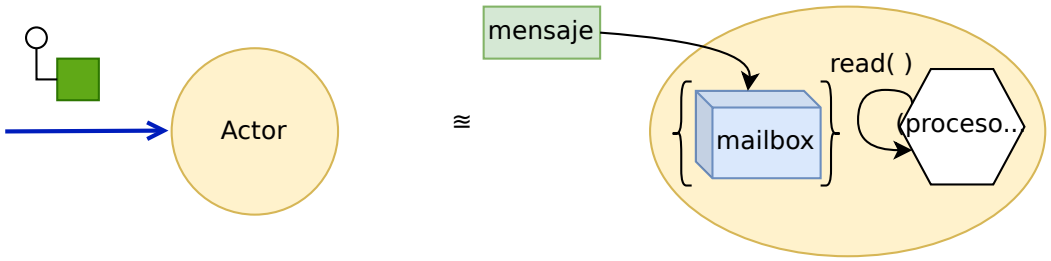

Figura 3.2: Diagrama del funcionamiento de un actor ante la llegada de un mensaje.
Imagen proporcionada por el tutor

A la vista de cómo se comunican los procesos, se entiende que puede aumentar rápidamente la complejidad de una aplicación si la única manera de relacionar procesos es mediante este mecanismo. Por ello en las aplicaciones *Elixir* se define una jerarquía en la creación de procesos. Existen dos tipos de procesos:

- *Supervisor*: un supervisor es un proceso que se encarga de monitorizar el comportamiento de los procesos que son sus *children*. El supervisor es responsable de establecer la estrategia de reinicio de los procesos cuando estos terminan con un fallo o ocurre algún fallo inesperado que detiene el proceso. En las aplicaciones *Elixir* todos los procesos tienen un supervisor. Cuando se define el inicio de una aplicación, lo que se está definiendo es un supervisor para que los distintos procesos que lanza la aplicación estén supervisados. Un proceso que es supervisado puede supervisar otros procesos, creándose así en las aplicaciones un árbol de supervisión. En la Figura 3.3 se muestra un árbol de supervisión.
- *Worker*: un proceso trabajador es aquel que realiza una tarea y está supervisado por un Supervisor.

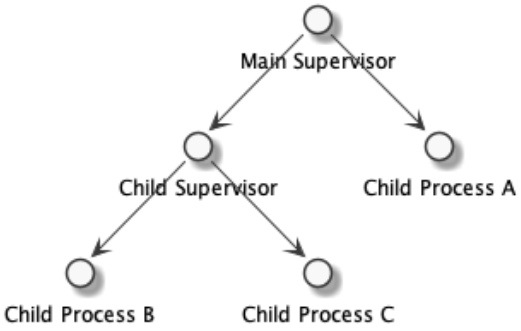

Figura 3.3: Árbol de supervisión

Otra mecanismo para añadir funcionalidad a distintas partes de una aplicación son los *behaviours*. Un *behaviour* es una manera de separar la parte génerica de un proceso de la parte especifica en un componente. En un *behaviour* se definen unas funciones llamadas *callbacks* que deben implementar la parte específica del componente. Los *behaviours* son similares a las interfaces en los lenguajes de programación orientados a objetos.

Uno de los *behaviours* más importantes de `Elixir` es GenServer. GenServer (*Generic Server*) ofrece la funcionalidad básica de un actor. Como se ha explicado anteriormente, un proceso recibe y procesa mensajes. GenServer ofrece los *callbacks* para el procesamiento de estos mensajes y la comunicación con otros procesos. De esta manera, GenServer representa la funcionalidad mínima que debe implementar un actor en un sistema, ya que se puede usar para monitorizar procesos, realizar tareas periódicas, o notificar a otros procesos cuando ocurren ciertos eventos. Además, GenServer se puede usar para mantener el estado del sistema mediante el envió de mensajes a sí mismo. En la Figura 3.4 se muestra cómo un actor recibe un mensaje y se produce una acción resultante, sin embargo, este mensaje puede ser enviado por el mismo actor. De esta manera se pueden crear una serie de estados en los que el flujo se controla por el actor mandándose el mensaje oportuno a sí mismo.

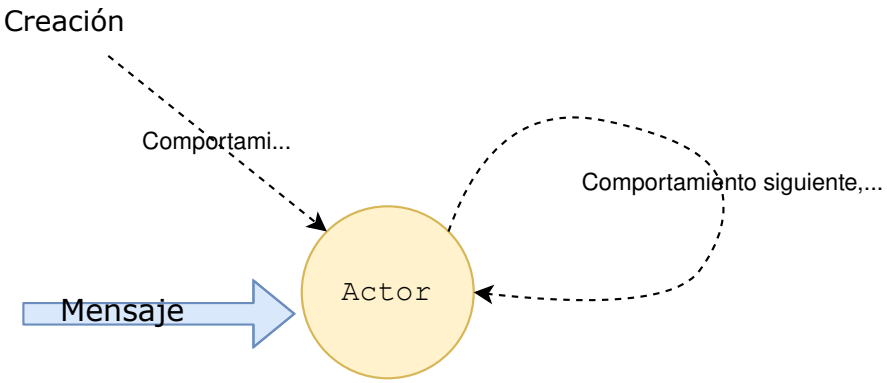

Figura 3.4: Diagrama del mecanismo que mantiene el estado mediante el paso de mensajes. Imagen proporcionada por el tutor

3.2.1. Paquetes de interés en este proyecto

Uno de los principales motivos por los que *Elixir* tiene cada vez más relevancia en aplicaciones y servicios modernos, es la gran variedad de paquetes con diversas funcionalidades que ofrece. Algunos de los paquetes más relevantes para la implementación de sistemas como el que se propone desarrollar en el proyecto son:

- `Libcluster` [14]: Este paquete proporciona un mecanismo automático de formación de clusters de nodos, encargándose de la funcionalidad de conectar nuevos nodos cuando aparecen. Además este paquete ofrece varias estrategias para la formación de estos clusters. En este proyecto existen dos estrategias que pueden ser de interés. En primer lugar existe una definición estática del cluster, en el que se define el nombre y dirección de cada uno de los nodos. Cuando un nodo arranca trata de conectarse al resto de nodos que tiene definido, produciendo un aviso si alguno no está disponible. En segundo lugar ofrece una estrategia de formación del cluster automática, mediante un *broadcast* en la red indicando su nombre y posición.

- `Horde` [15]: Este paquete ofrece dos módulos, un supervisor y un registro, que permiten la supervisión y registro de procesos de manera distribuida sobre un *pool* de nodos.

- `Phoenix` [16]: Phoenix es un *framework* de desarrollo de aplicaciones web escrito en *Elixir* para aplicaciones que utilizan el modelo *server-side*. Este *framework* ofrece toda la funcionalidad necesaria de principio a fin de una aplicación web.

Capítulo 4

Análisis

En esta fase del desarrollo del proyecto se obtiene la información acerca de las necesidades y objetivos de este. Se deben detallar las distintas funcionalidades y componentes que debe haber en cada una de las partes del sistema.

Estos requisitos han surgido de entrevistas con el cliente, en este caso se trata de una combinación de un usuario del lenguaje de programación R y las aportaciones del tutor sobre lo que se esperaría siendo un usuario de un sistema de computación distribuido. También se debe estudiar cómo va a interactuar un administrador del sistema con las siguientes partes y componenetes físicos.

Es importante destacar que, a pesar de que los distintos componentes del sistema están descritos en la sección de diseño, se mencionan elementos del sistema, **RPack** es el paquete R que se ejecuta desde una consola R del usuario, **Rhub** es el componente que se encarga de distribuir tareas en el sistema y **RNodePool**, el sistema distribuido formado por varios nodos. En está sección se utiliza el término **nodo** para hacer referencia a un nodo de **RNodePool**. Cuando se habla de **ejecución** o **ejecutar** se refiere a una ejecución de cierto código que quiere realizar el usuario en el sistema o la acción de realizar esta.

En el análisis y el diseño de este proyecto se utiliza el lenguaje de modelado SysML [17]. Se ha escogido este lenguaje por su flexibilidad a la hora de modelar diversos sistemas y por sus similaridades con UML (*Unified Modeling Language*).

4.1. Identificación de usuarios

Existen dos usuarios del sistema, un usuario que utiliza **RPack** desde su consola R para realizar computaciones distribuidas y un administrador, que es el responsable del correcto funcionamiento del sistema. Las principales características de cada uno de estos es:

- Usuario: Se presupone que el usuario tiene conocimientos acerca del lenguaje de programación R, por lo que en cuanto al análisis se trata de hacer el sistema intuitivo teniendo en cuenta las prácticas habituales de los paquetes de este lenguaje. Es importante considerar que los usuarios de R suelen

ser profesionales en diversos campos y usan R como herramienta de *scripting* para analizar sus datos, y por tanto, pueden no tener tanto conocimiento técnico acerca del funcionamiento interno de R o de conceptos como un sistema distribuido. A lo largo del análisis se busca simplificar la experiencia del usuario al máximo, de tal manera que el sistema aporte su funcionalidad manteniendo el sistema distribuido oculto al usuario.

- Administrador: El administrador tiene conocimientos acerca del funcionamiento del sistema y sus diferentes partes. Se presupone que tiene conocimientos moderados de administración de sistemas. Además debe tener conocimientos básicos sobre R, ya que es responsable de instalar paquetes externos que el usuario le solicite. Por último, es el responsable de manejar la entrada o salida de nodos en **RNodePool**.

4.2. Requisitos

En esta sección del capítulo se enumeran y describen los requisitos que se obtienen a partir de los objetivos propuestos.

4.2.1. Requisitos funcionales

Los requisitos funcionales tienen la meta de describir con exactitud las funcionalidades que el sistema o sus componentes deben implementar. Estos requisitos están centrados en los aspectos de las distintas ejecuciones que puede solicitar el usuario.

- RF-1: El sistema deberá permitir al usuario **ejecutar** sus propias lineas de código R en un nodo remoto.
- RF-2: El sistema deberá permitir al usuario **ejecutar** la operación `apply()` distribuida entre los nodos.
- RF-3: El sistema deberá permitir al usuario **ejecutar** la operación `tapply()` distribuida entre los nodos.
- RF-4: El sistema deberá permitir al usuario **ejecutar** la operación `lapply()` distribuida entre los nodos.
- RF-5: El sistema deberá permitir al usuario **ejecutar** la operación `sapply()` distribuida entre los nodos.
- RNF-6: El sistema deberá dividir las operaciones y los datos en partes cuando sea posible antes de distribuirlas entre los nodos.
- RF-7: El sistema no deberá bloquear al usuario mientras se realiza una **ejecución**.
- RF-8: El sistema deberá permitir al usuario obtener los resultados de una **ejecución**.

- RF-9: El sistema deberá mostrar al usuario el progreso de las **ejecuciones**.
- RF-10: El sistema podrá tener más de un nodo funcionando simultáneamente.
- RF-11: El sistema deberá mostrar la información relevante para la administración de los nodos activos.
- RF-12: El sistema deberá permitir al administrador introducir nuevos nodos en el sistema.
- RF-13: El sistema deberá informar al usuario de errores en sus **ejecuciones**.
- RF-14: El sistema deberá equilibrar la carga entre los nodos.
- RF-15: El sistema deberá disponer de un medio para almacenar los datos de las **ejecuciones**.
- RF-16: El sistema deberá permitir al administrador administrar el almacenamiento de datos.

4.2.2. Requisitos no funcionales

Los requisitos no funcionales o de calidad nos indican cómo se deben llevar a cabo los requisitos funcionales. Es decir, indican características y restricciones acerca del funcionamiento del sistema.

- RNF-1: La comunicación entre el componente **RPack** y **RHub** se realizará mediante el protocolo HTTP [18].
- RNF-2: El sistema deberá ofrecer la operación `apply()` de manera equivalente a la del paquete base de R.
- RNF-3: El sistema deberá ofrecer la operación `lapply()` de manera equivalente a la del paquete base de R.
- RNF-4: El sistema deberá ofrecer la operación `tapply()` de manera equivalente a la del paquete base de R.
- RNF-5: El sistema deberá ofrecer la operación `sapply()` de manera equivalente a la del paquete base de R.
- RNF-6: El sistema deberá equilibrar la carga en cada nodo utilizando las peticiones HTTP que recibe como criterio de reparto.
- RNF-7: El sistema deberá mostrar mediante una página Web la información sobre las **ejecuciones**.
- RNF-8: El sistema deberá mostrar mediante una página Web la información sobre el estado de los nodos.
- RNF-9: El sistema deberá informar de posibles fallos en la **ejecución** de una operación mediante una excepción en la consola R al **ejecutar** la operación fallida.

- RNF-10: El sistema deberá recuperar el estado de una **ejecución** y retomarla en la medida de lo posible cuando un nodo falla.
- RNF-11: El sistema identificará las **ejecuciones** mediante un token.

Requisitos no funcionales de Accesibilidad

Estos requisitos nos indican de qué manera serán accesibles las partes del sistema por el usuario.

- El sistema debe garantizar que una vez desplegado el sistema completo, el usuario puede usar de manera transparente el *pool* de nodos utilizando el paquete R proporcionado.

Requisitos no funcionales de Seguridad

Estos requisitos no funcionales determinan los diferentes requisitos de seguridad que existen en el sistema.

- RNFS-1: El sistema debe garantizar que el intercambio de datos de las **ejecuciones** se realiza a través de canales seguros. El nivel de seguridad debe ser equivalente al proporcionado por el protocolo SSH [18].
- RNFS-2: El sistema deberá rechazar las peticiones que no provengan de su propio sistema en la medida de lo posible.

Requisitos no funcionales de Usabilidad

Estos requisitos indican la facilidad de uso que debe tener el sistema.

- RNFU: El sistema debe actualizar la información de las paginas Web indicadas en los casos de uso RNF-7 y RNF-8 de manera automática cuando ocurre un cambio en el sistema.

Requisitos no funcionales de Disponibilidad

- RNFD-1: El sistema deberá implementar los elementos **RNodePool** de tal manera que estos sean resilientes al fallo.

Requisitos no funcionales de Información

Estos requisitos describen los datos que debe almacenar el sistema.

- RNFI-1: El sistema deberá almacenar los siguientes datos de configuración en el componente **RPack**:
  - Direcciones del servidor Web, dirección de la página Web y de las direcciones a las que hacer solicitudes.
  - Dirección, usuario y contraseña del servidor de datos para la transferencia de datos.

  - Nombre del usuario.
  - Número máximo de **ejecuciones** que se permiten en el **RNodePool**.

- RNFI-2: El sistema deberá almacenar en el **RHub**:
  - La dirección IP y puerto que pone a disposición de cada uno de los nodos del **RNodePool** para que estos puedan ser localizados.
- RNFI-3: Cada nodo de **RnodePool** deberá almacenar la siguiente información:
  - Número máximo de ejecuciones que se permiten en cada nodo.
  - Dirección, usuario y contraseña del servidor de datos para la transferencia de datos.
  - Listado de los nodos en el **Sistema Distribuido**.
  - Listado de **ejecuciones** que hay en el sistema.
  - Listado de Paquetes R externos necesarios que ha solicitado el usuario para usar el sistema.
- RNFI-4: El sistema deberá almacenar la siguiente información sobre cada **ejecución** en el **RNodePool**:
  - Nombre de usuario.
  - Nodo en que se está realizando la **ejecución**.
  - Fecha y hora de comienzo.
  - Fecha y hora de fin.
  - Token que identifica la operación.
  - Estado de la **ejecución**.
- RNFI-5: El sistema deberá almacenar los siguientes datos correspondientes a una **ejecución**:
  - Objeto de datos R procesados de los datos que proporciona el usuario.
  - Código R correspondiente a la **ejecución**.
  - Objeto R con los resultados de una **ejecución** en caso de que esta haya sido exitosa.
  - Mensaje de Error de la **ejecución** en caso de que esta ocurra.

4.3. Diagrama de paquetes del proyecto

En la Figura 4.1 se muestra el diagrama de paquetes para el sistema en esta etapa de análisis.

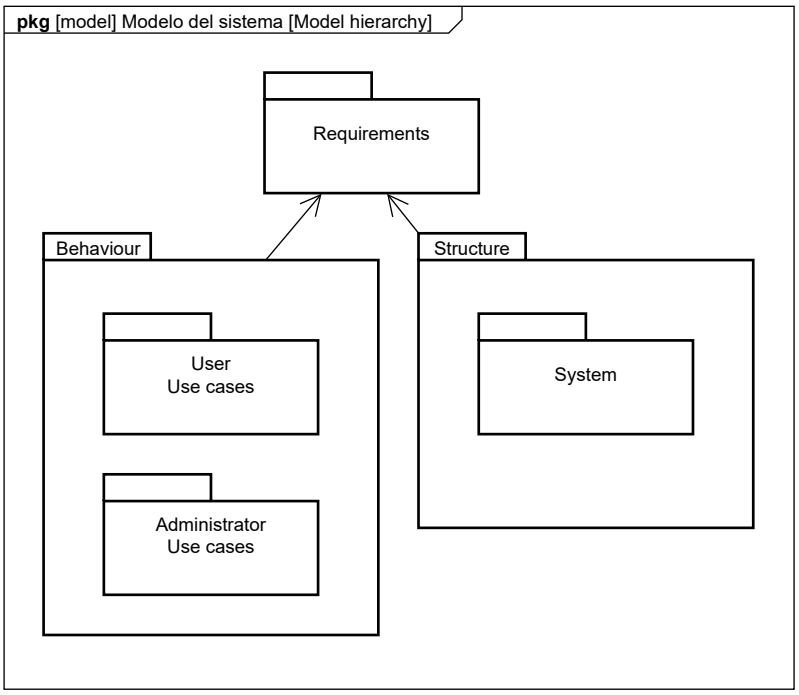

Figura 4.1: Casos de Uso del Usuario

4.4. Casos de uso

Un caso de uso es una descripción de las actividades que realiza un actor del sistema cuando utiliza el sistema. Estos casos de uso quedan definidos para dejar reflejadas las formas que tienen los usuarios de interactuar con el sistema y qué se espera del sistema en cada uno de estos escenarios.

4.4.1. Diagramas de casos de uso

En la Figura 4.2 se muestran los casos de uso para el actor usuario.

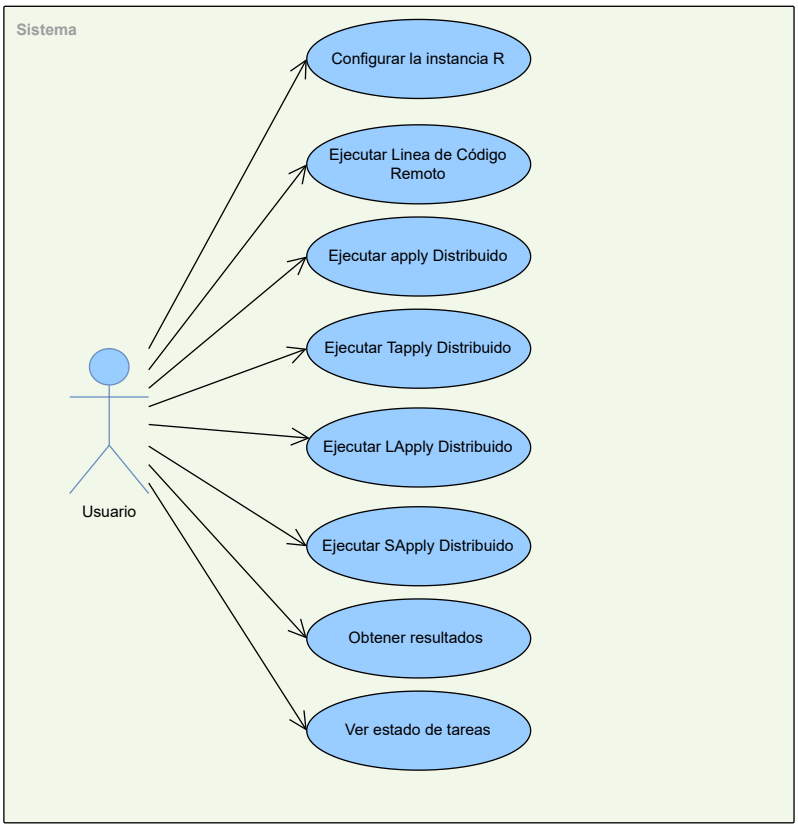

Figura 4.2: Casos de Uso del Usuario

En la Figura 4.3 se muestran los casos de uso para el Administrador.

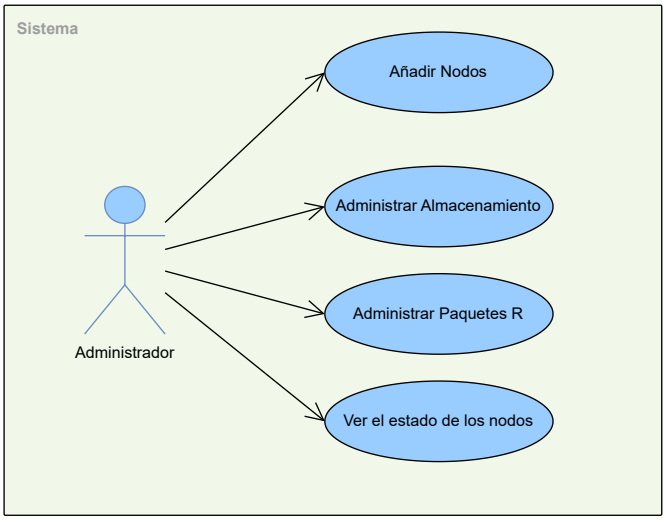

Figura 4.3: Casos de Uso del Administrador

4.4.2. Especificación de los casos de uso

En la especificación de los casos de uso se muestran las precondiciones, la secuencia principal y posibles alternativas del caso de uso y las postcondiciones una vez ha terminado.

Se ha decidido por brevedad agrupar los casos de uso 4, 5, 6 y 7 en el mismo flujo ya que a nivel de análisis las 4 operaciones `apply()`, `tapply()`, `lapply()` y `sapply()` se comportan de la misma manera. Como se ha explicado en el Capitulo 3, dependiendo de qué operación de estas 4 sea la correspondiente, se puede dividir la operación y los datos en varias ejecuciones de manera diferente. En este flujo cuando se habla de operación se referencia a cualquiera de las 4 definidas.

También es importante entender que el administrador maneja los componentes del sistema directamente, es decir tiene acceso, ya sea físico o remoto a los componentes **RNodePool** y **RHub**, por lo que interactúa con el sistema de manera muy distinta al usuario. En este proyecto se ha decidido que la parte administrativa del sistema se realice con herramientas externas que permiten llegar a los archivos de configuración del sistema en cada componente y al almacenamiento.

CU-1	**Configurar el Paquete R**
Actor	**Usuario**
Descripción	El sistema deberá establacer la configuración que le indique el usuario.
Precondición	El usuario ha cargado en su entorno R el paquete R que proporciona este sistema.
Secuencia Normal	1. El usuario ejecuta la función R correspondiente a la configuración estando esta configuración en un archivo de texto que proporciona. 2. El sistema establece la configuración en el entorno del usuario y le muestra la Web donde puede obtener más información acerca del estado de sus ejecuciones.
Postcondición	El entorno de R del usuario está configurado.
Excepciones	
Variación	**Acción**
2b	El sistema informa al usuario de que el archivo de configuración no existe.

Tabla 4.1: Caso de Uso 1 -Configurar el Paquete R

CU-2	**Ejecutar Lineas de Código Remoto**
Actor	**Usuario**
Descripción	El sistema ejecuta las lineas de código con los datos que proporciona el usuario.
Precondición	(CU-1) El entorno R del usuario debe haber sido configurado.
Secuencia Normal	1. El usuario proporciona las lineas de código y los datos para la ejecución. 2. El sistema procesa las lineas y los datos, los almacena y comienza su ejecución en un nodo de **RNodePool**. 3. El sistema devuelve al usuario un token único que identifica la operación.
Postcondición	Se ha iniciado una nueva ejecución en el sistema.
Excepciones	
Variación	**Acción**
2b	El sistema informa al usuario de que no se han podido almacenar los datos.
3b	El sistema informa al usuario de que no se ha podido establecer contacto con el componente **RHub**.

Tabla 4.2: Caso de Uso 2 - Ejecutar Lineas de Código Remoto

CU-3	**Obtener resultados**
Actor	**Profesor**
Descripción	El usuario solicita los resultados de una ejecución o una operación de la que tiene su token correspondiente.
Precondición	El usuario debe tener el token correspondiente a una ejecución u operación anterior.
Secuencia Normal	1. El usuario proporciona al sistema el token de una ejecución u operación. 2. El sistema consulta el estado de la ejecución u operación. 3. El sistema comprueba que esta ha terminado correctamente. 4. El sistema carga en el entorno R los resultados de la ejecución u operación y se los devuelve al usuario.
Postcondición	El usuario obtiene los resultados de su ejecución u operación.
Excepciones	
Variación	**Acción**
2b	El sistema informa al usuario que no se ha podido establecer contacto con el componente **RHub**.
3c	1. El sistema comprueba que la ejecución ha terminado con error de ejecución en la ejecución u operación. 2. El sistema obtiene el error que ha ocurrido en la ejecución. 3. El sistema informa al usuario de que ha ocurrido un error y le muestra el mensaje de error.
3d	1. El sistema comprueba que la ejecución u operación aun no ha finalizado. 2. El sistema se bloquea esperando a que la ejecución termine, bloqueando el resto del entorno R del usuario. 3. El caso de uso se retoma en el paso 4.
4b	El sistema informa al usuario de que no se han podido obtener los datos de los resultados.

Tabla 4.3: Caso de Uso 3 - Obtener resultados

CU-4,5,6,7	**Ejecutar operación distribuida**
Actor	**Usuario**
Descripción	Se realiza la operación de la manera indicada por el usuario dividiendo la operación en varias ejecuciones.
Precondición	(CU-1) El entorno R del usuario debe haber sido configurado.
Secuencia Normal	1. El usuario proporciona los parámetros relevantes a la operación. 2. El sistema divide los datos y ejecuciones en varias partes y crea para cada parte una ejecución. 3. El sistema procesa las lineas y los datos de cada ejecución, los almacena y comienza cada ejecución en un nodo de **RNodePool**. 4. El sistema devuelve al usuario un token único que identifica la operación.
Postcondición	Se han iniciado una o más ejecuciones en el sistema.
Excepciones	
Variación	**Acción**
2b	El sistema informa al usuario de que no se han podido almacenar los datos.
3b	El sistema informa al usuario de que no se ha podido establecer contacto con el componente **RHub**.

Tabla 4.4: Caso de Uso 4, 5, 6 y 7 - Ejecutar Operación Distribuida

CU-8	**Ver estado de tareas**
Actor	**Usuario**
Descripción	Se muestra una web con la información acerca del estado de las ejecuciones.
Precondición	-
Secuencia Normal	1. El usuario se dirige a la web que muestra la información de las ejecuciones. 2. El sistema muestra la información sobre las ejecuciones que hay en el sistema. 3. El sistema actualiza de manera automática la información de cada ejecución y nuevas ejecuciones cuando hay cambios.
Postcondición	Se muestra la información de las ejecuciones en el sistema.

Tabla 4.5: Caso de Uso 8 - Ver estado de tareas

CU-9	**Añadir Nodo**
Actor	**Administrador**
Descripción	El sistema añade un nodo nuevo **RNodePool**.
Precondición	-
Secuencia Normal	1. El administrador añade la dirección del nuevo nodo a las configuraciones de **RHub** y **RNodePool**. 2. El administrador lanza la nueva instancia del programa de cada nodo en el nuevo nodo. 3. El sistema conecta automáticamente el nuevo nodo al **pool** de nodos disponibles sin necesidad de detenerse.
Postcondición	Se ha añadido un nodo al **pool** de nodos disponibles.
Excepciones	
Variación	**Acción**
3b	El sistema informa al administrador mediante un aviso de que no se ha podido establecer la conexión con el resto de nodos y si se conoce, informa sobre la posible causa.

Tabla 4.6: Caso de Uso 9 - Añadir Nodo

CU-10	**Administrar almacenamiento**
Actor	**Administrador**
Descripción	El administrador accede al almacenamiento de datos.
Precondición	-
Secuencia Normal	1. El administrador accede al almacenamiento de datos. 2. El sistema muestra sus contenidos y le permite borrar o añadir archivos.
Postcondición	Se muestran al administrador los contenidos del sistema y se le permite realizar operaciones en este.
Excepciones	
Variación	**Acción**
2b	El sistema informa al administrador de que no se ha podido acceder al almacenamiento de datos.

Tabla 4.7: Caso de Uso 10 - Administrar almacenamiento

CU-11	**Administrar paquetes R**
Actor	**Administrador**
Descripción	El administrador accede a los nodos y añade los paquetes R relevantes al usuario.
Precondición	-
Secuencia Normal	1. El administrador añade los paquetes y su linea de instalación al archivo de configuración en cada nodo. 2. El sistema instala el paquete en el nodo la primera vez que el usuario necesite usarlo en una ejecución.
Postcondición	Se muestran al administrador los contenidos del sistema y se le permite realizar operaciones en este.
Excepciones	
Variación	**Acción**
2b	El sistema informa al administrador mediante un error en el nodo, de que no se ha podido instalar el paquete.

Tabla 4.8: Caso de Uso 11 - Administrar paquetes R

CU-12	**Ver estado de los nodos**
Actor	**Administrador**
Descripción	Se muestra una web con la información acerca del estado de las ejecuciones.
Precondición	-
Secuencia Normal	1. El Administrador se dirige a la web que muestra la información de las ejecuciones. 2. El sistema muestra la información sobre el estado de los nodos en el sistema. 3. El sistema actualiza de manera automática la información de los nodos cuando ocurren cambios.
Postcondición	Se muestra la información de los nodos en el sistema.

Tabla 4.9: Caso de Uso 12 - Ver estado de los nodos

4.5. Diagrama de Bloques

En la Figura 4.4 se muestra un diagrama de bloques para el sistema. Este diagrama de análisis muestra a alto nivel los diferentes bloques que se corresponden con los principales componentes del sistema, así como los subcomponentes que los forman. Además, este diagrama indica que cada componente y subcomponente representa un elemento físico, que por tanto puede estar en distintos puntos físicos una vez se despliegue.

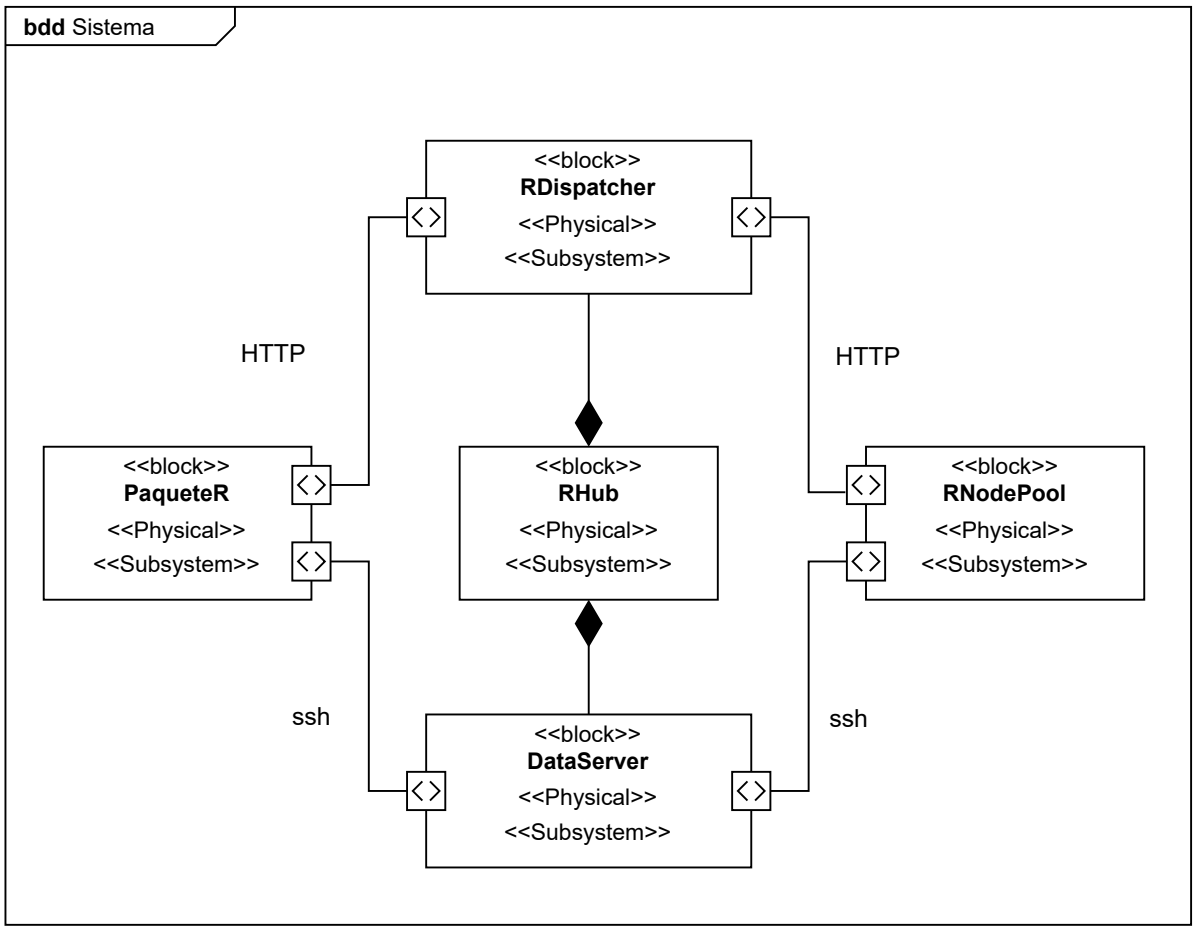

Figura 4.4: Diagrama de Bloques del sistema

4.6. Diagramas de Actividades

En esta sección se muestra detalladamente el transcurso de actividades en los casos de uso más representativos. Los casos de uso que se omiten presentan un diagrama poco relevante o existen diagramas que ilustran su funcionamiento de manera similar.

4.6.1. CU-2: Ejecutar Lineas de Código Remoto

En la Figura 4.5 se muestra el diagrama de actividades del CU-2. Aquí debe quedar claro que una vez se devuelve el token, el usuario deja de estar esperando. El sistema funciona de manera asíncrona, es

decir, este sigue realizando actividades una vez el usuario ha terminado el caso de uso. Por tanto, se ha decidido en este diagrama y en el pertinente al CU-4,5,6,7 mostrar el resto del funcionamiento del sistema una vez el caso de uso ha finalizado para el usuario, con objeto de ver qué actividades desencadena iniciar estos casos de uso.

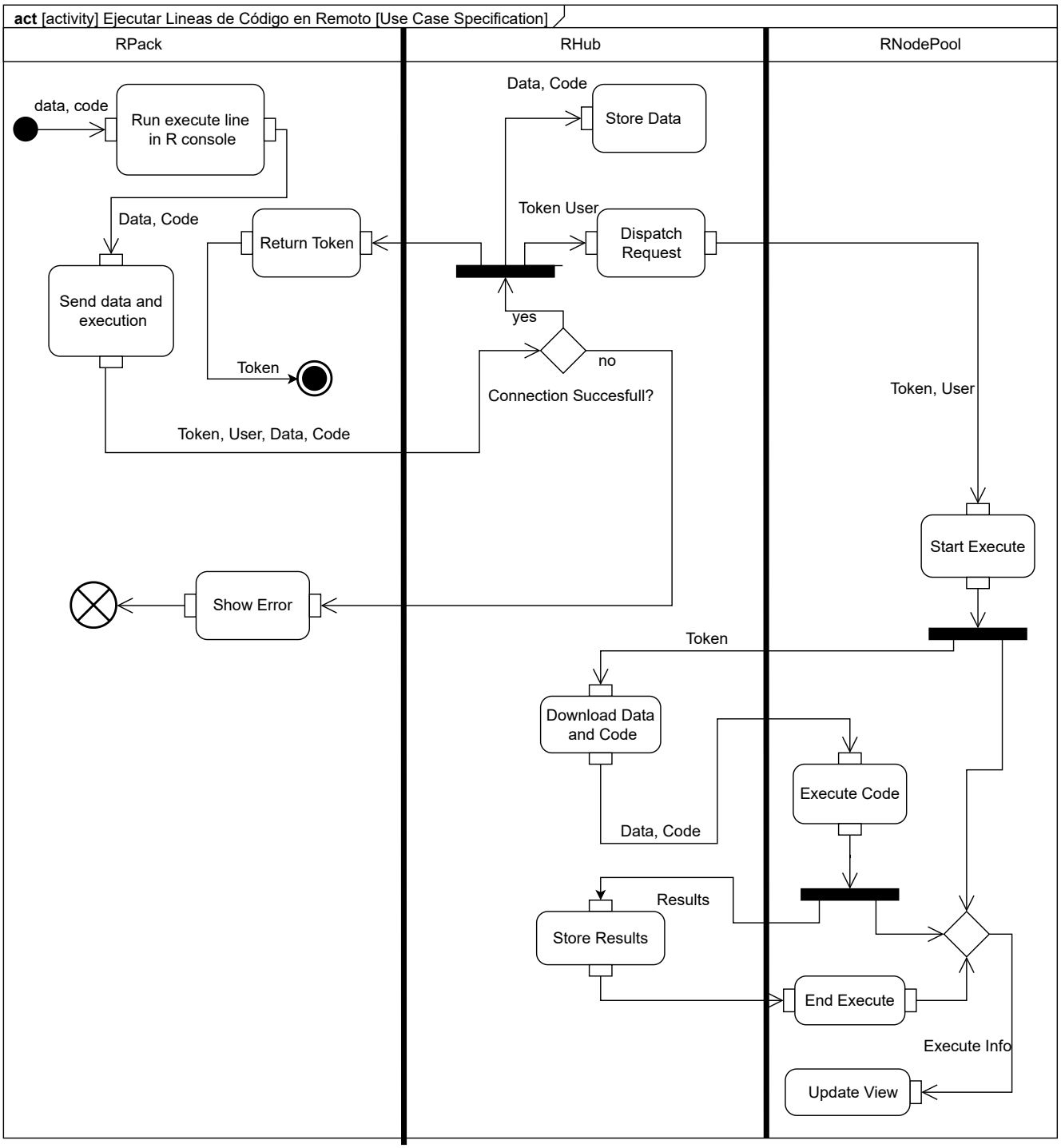

Figura 4.5: Diagrama de Actividades del CU-2: Ejecutar Lineas de Código Remoto

4.6.2. CU-3: Obtener resultados

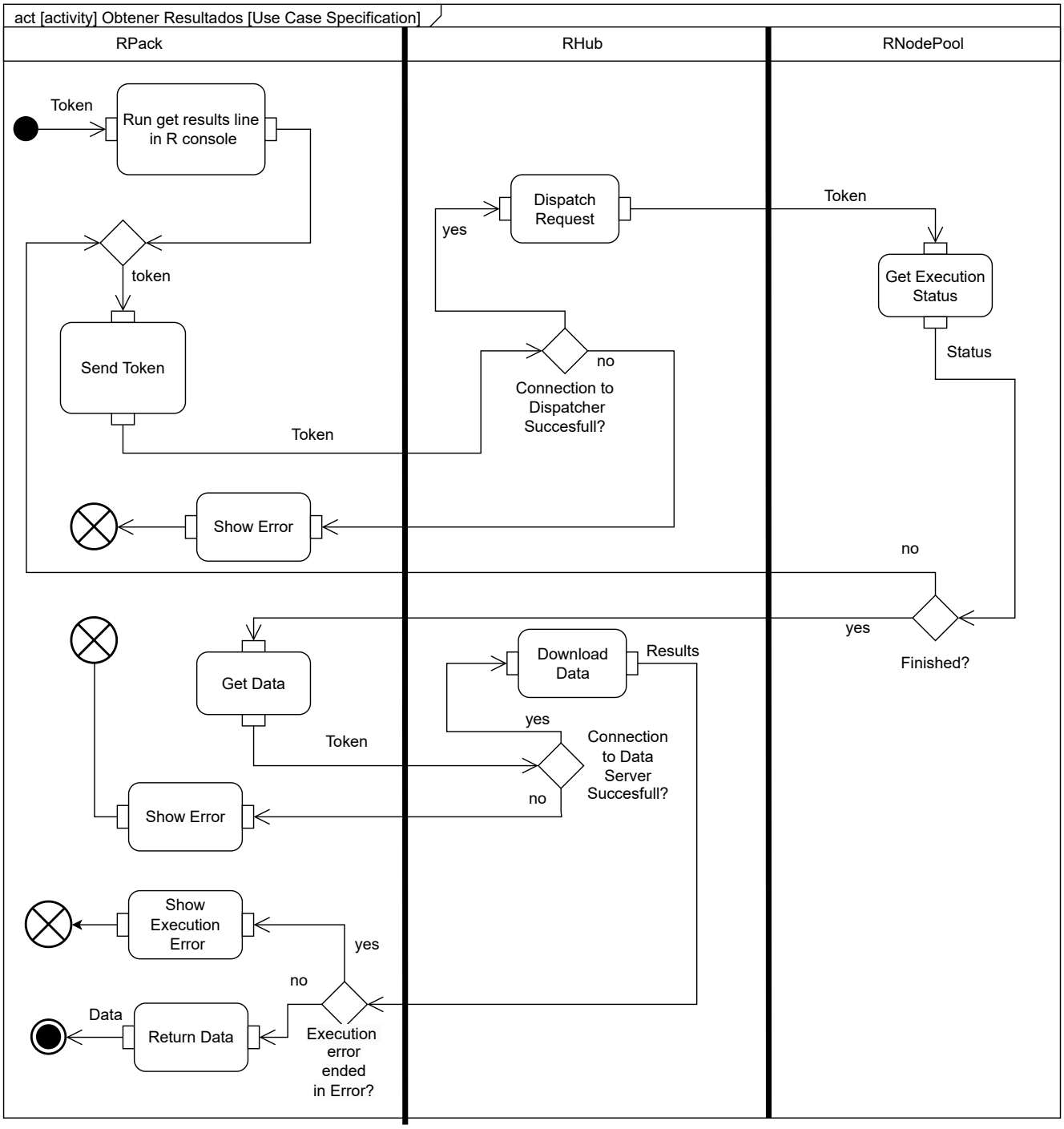

Figura 4.6: Diagrama de Actividades del CU-3: Obtener resultados

4.6.3. CU-4,5,6,7: Realizar operación distribuida

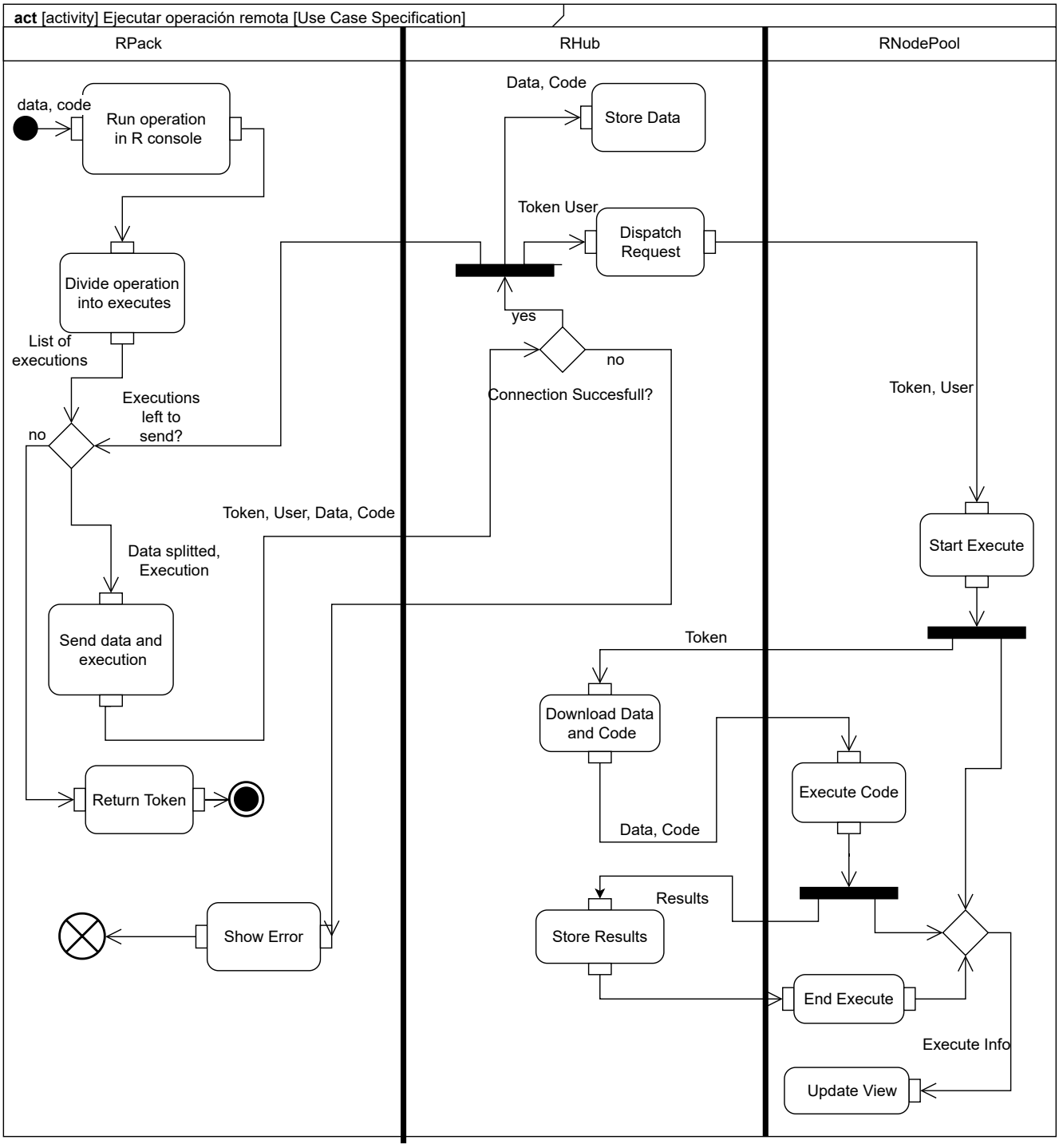

Figura 4.7: Diagrama de Actividades del CU-4,5,6,7: Realizar operación distribuida

4.6.4. CU-8: Ver estado de tareas

En la Figura 4.8 se muestra el diagrama de actividades del CU-8. Aquí, se observa que la vista, en este caso una web, se actualiza automáticamente sin que el usuario realice ninguna acción.

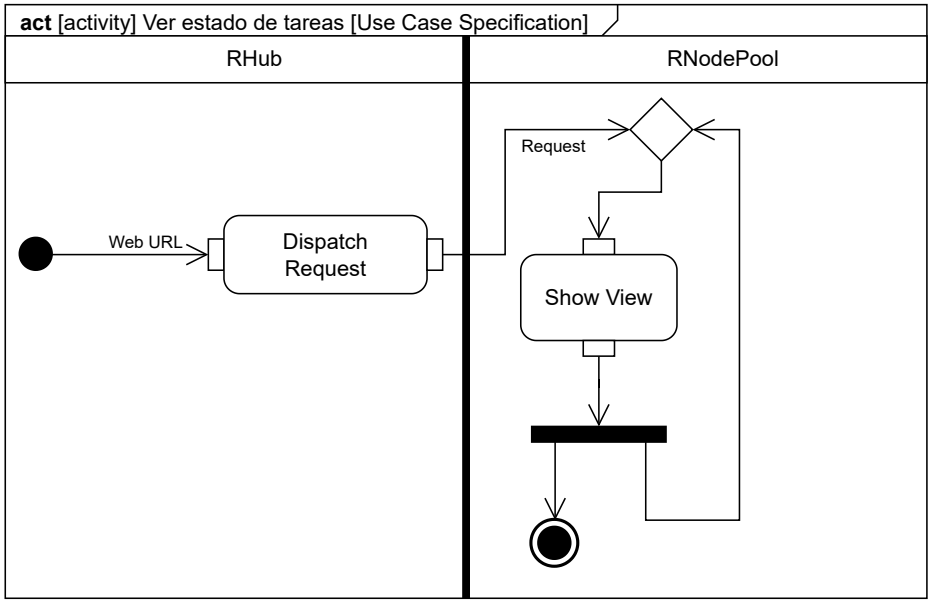

Figura 4.8: Diagrama de Actividades del CU-8: Ver estado de tareas

4.6.5. CU-9: Añadir Nodo

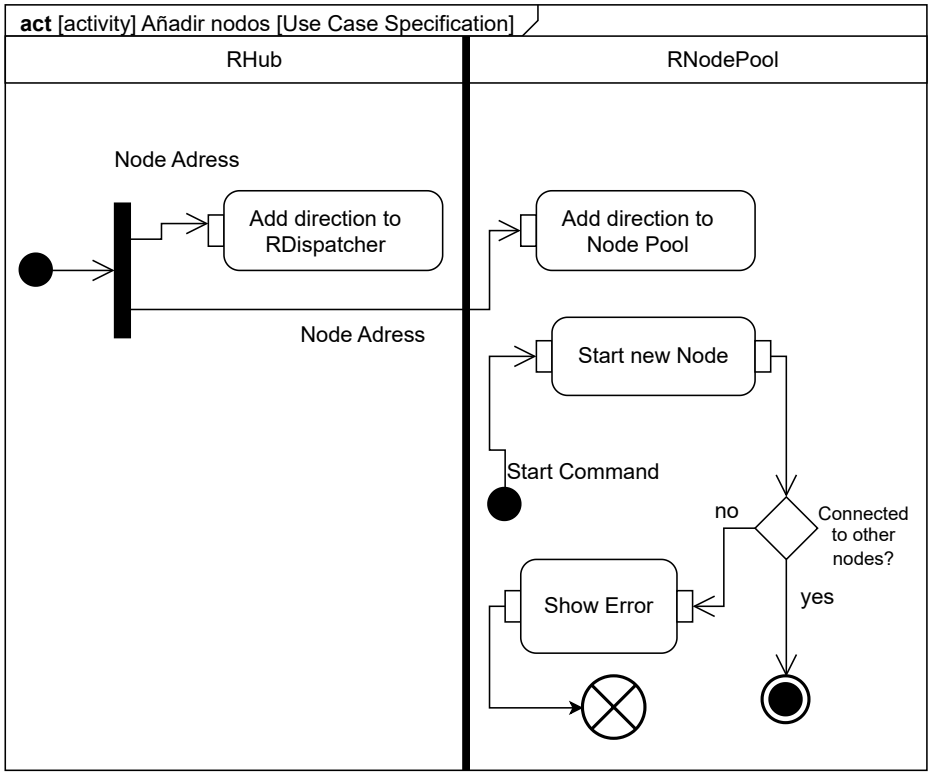

Figura 4.9: Diagrama de Actividades del CU-9: Añadir Nodo

Capítulo 5

Diseño

El diseño de un sistema supone, una vez analizado este, la especificación del funcionamiento y arquitectura de cada una de las partes. Debido al tipo de sistema que se plantea en este proyecto, la estructura de esta sección del documento se desvía ligeramente de lo habitual. Ya que se plantea diseñar una aplicación que va a utilizar dependencias externas, se ha decidido dar mayor importancia a las partes que se van a desarrollar en este proyecto y posteriormente en el Capítulo de Implementación se ilustrarán las dependencias externas con mayor detalle.

Es importante hacer incapié en la importancia del diseño del componente **RNodePool**, ya que es este el que concentra la gran mayoría de la funcionalidad, así como ser la parte del proyecto que representa la mayor complejidad al estar formado por varios nodos con el mismo funcionamiento. Es por eso que esta sección comienza con la exploración de este componente y posteriormente se estudia la conexión entre las distintas partes del sistema.

5.1. RNodePool

RNodePool es una aplicación distribuida, es decir, está siendo ejecutada simultáneamente en varios nodos, y estos nodos se comunican entre ellos para mantener el estado del sistema. Es importante distinguir entre partes globales al sistema y partes especificas a cada nodo, aunque hay que entender que no existe ningún componente que mantenga estas partes globales fuera de los nodos. Cuando se habla de un componente global del sistema, se refiere a que este está replicado entre los nodos y estos se comunican entre ellos para garantizar que los contenidos concuerden. Por tanto, en este sistema no existe un supervisor global que resida en una parte específica del sistema, es decir, cada nodo realiza su funcionalidad y además es responsable de supervisar el resto de nodos. Este diseño de supervisión todos a todos es lo que permite tener resiliencia al fallo en un sistema descentralizado de nodos en el que todos los nodos tienen la misma funcionalidad. Este diseño también facilita la entrada de nodos nuevos al sistema, ya que solo deben ser descubiertos por el resto de nodos y que comience a ser supervisado y a utilizar los recursos compartidos.

También cabe destacar que este diseño está orientado a un sistema que posteriormente va a ser imple-

mentado en el lenguaje Elixir que, como ya se ha explicado en el Cápitulo 3, está basado en un sistema en el que existen varios procesos, todos supervisados por el proceso que los lanzó, que se comunican a través del paso de mensajes. Es importante no confundir la supervisión de un proceso por el proceso que lo ha creado, con la supervisión que se ha mencionado en el párrafo anterior, que existe a mayores para mantener un *pool* de nodos. A lo largo de esta sección se ejemplifica mejor esta diferencia.

En la Figura 5.1 se muestra el Diagrama de Bloques para el componente **RNodePool**.

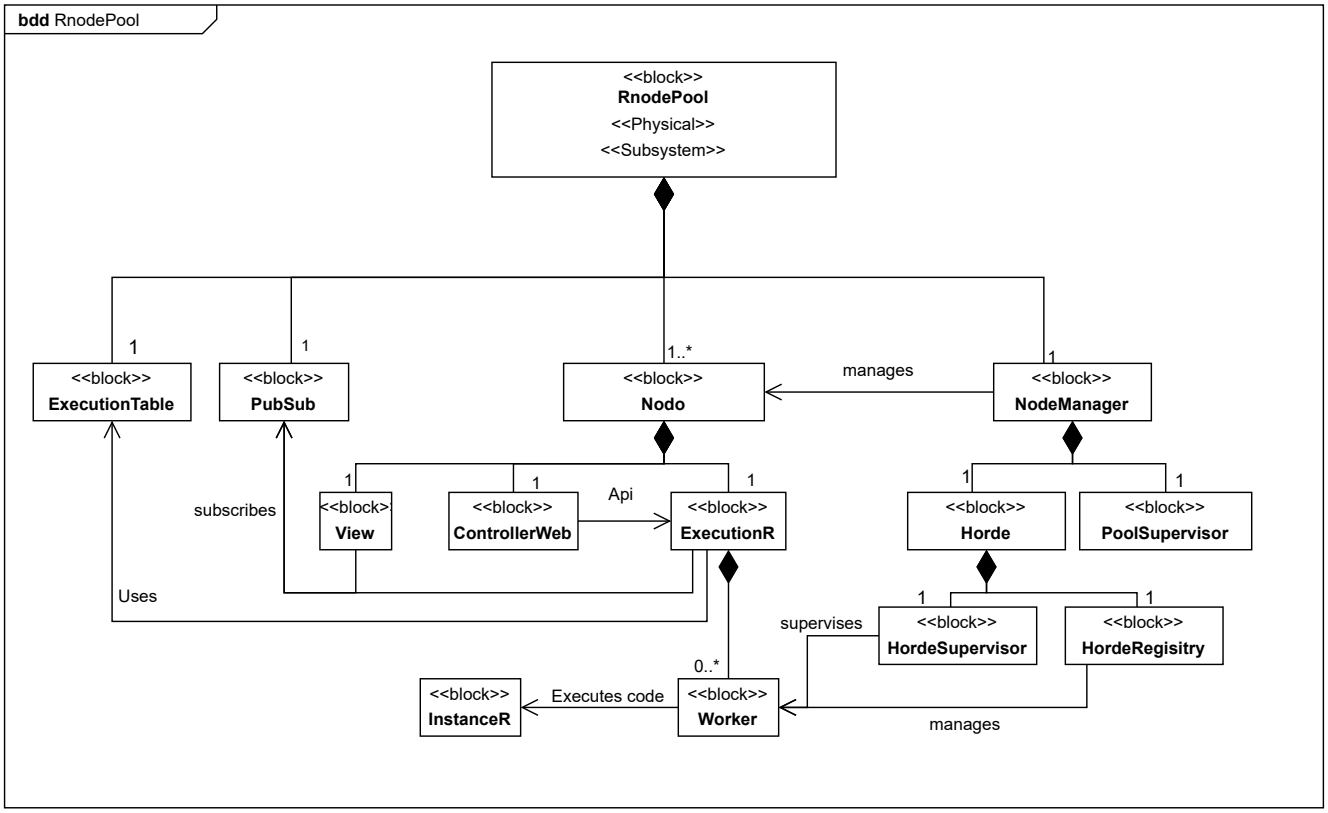

Figura 5.1: Diagrama de Bloques del componenete **RNodePool**

- **Nodo**: Un nodo es la principal unidad de funcionamiento **RNodePool**. Cada nodo está formando por 3 componentes principales. Una vista, un controlador de las peticiones y la lógica del nodo, **ExecutionR**.

    - **View**: Se encarga de cargar las vistas de manera apropiada con la llegada de nueva información acerca de las ejecuciones en el sistema.
    - **ControllerWeb**: Recibe las peticiones del usuario y las dirige al componente responsable de ellas y genera la respuesta para el usuario.
    - **ExecutionR**: Se encarga de la lógica de la aplicación, no solo de la lógica de cada ejecución, sino también de la lógica de mantener el estado y la consistencia en el resto de componentes globales. Ofrece una API o interfaz con la que se puede interactuar para crear y obtener información de ejecuciones. Para cada una de estas ejecuciones, se crea un trabajador **Worker** que es el responsable de llevar a cabo una ejecución y ejecutar el código R en una instancia R.

- **ExecutionTable**: Esta tabla se encarga de mantener el estado y la información de las ejecuciones del sistema y es un componente global en **RNodePool**, es decir, todos los nodos tienen acceso a la información global. Sin embargo, como se ha explicado antes, esta tabla reside en cada uno de los nodos. Se debe garantizar en la medida de lo posible que esta tabla mantenga la consistencia entre todos los nodos.

- **NodeManager**: Este componente es responsable de la administración de los nodos del sistema. Como ya se ha explicado antes, **NodeManager** está presente en todos los nodos y cada uno de estos nodos supervisa y administra a todos los demás.

  - **PoolSupervisor**: Se encarga de la formación automática del *pool* de nodos y de establecer el enlace entre los nodos cuando se detecta uno nuevo.
  - **Horde**: Supervisa y administra el estado de los nodos. Es responsable de tratar los cambios en el *pool* de nodos. Es responsable de mantener un registro de los nodos del sistema y de supervisar los procesos de los nodos. Por tanto, este componente es responsable de reiniciar un **Worker** en otro nodo en caso de que el nodo que está realizando el trabajo caiga.

- **PubSub**: Para que varios procesos se puedan comunicar con todos los demás procesos, debido a que la comunicación es mediante paso de mensajes, el componente PubSub ofrece un mecanismo de *Publish-Subscribe* por el que varios procesos pueden conocer cambios en el resto de ellos subscribiéndose al mismo tema y escuchando las emisiones apropiadas.

A continuación se explica con más detalle el funcionamiento y estructura de cada uno de los componentes de **RNodePool**

5.1.1. ExecutionR

ExecutionR ofrece una API o interfaz para interactuar con sus partes. Ofrece tres funcionalidades que delega en sus componentes internos :

- `start_execute( token, usuario)` inicia un **Worker** para realizar una ejecución.
- `is_ready(token)` Retorna el estado de la ejecución correspondiente al token.
- `get_table()` Retorna la información de todas las ejecuciones que hay en **RNodePool**.

En la Figura 5.2 se muestra el diagrama de actividades para la operación `start_execute`. En este diagrama aparecen los componentes de **NodeManager**, ya que aunque sean componentes globales de **RNodePool**, forman parte de cada nodo.

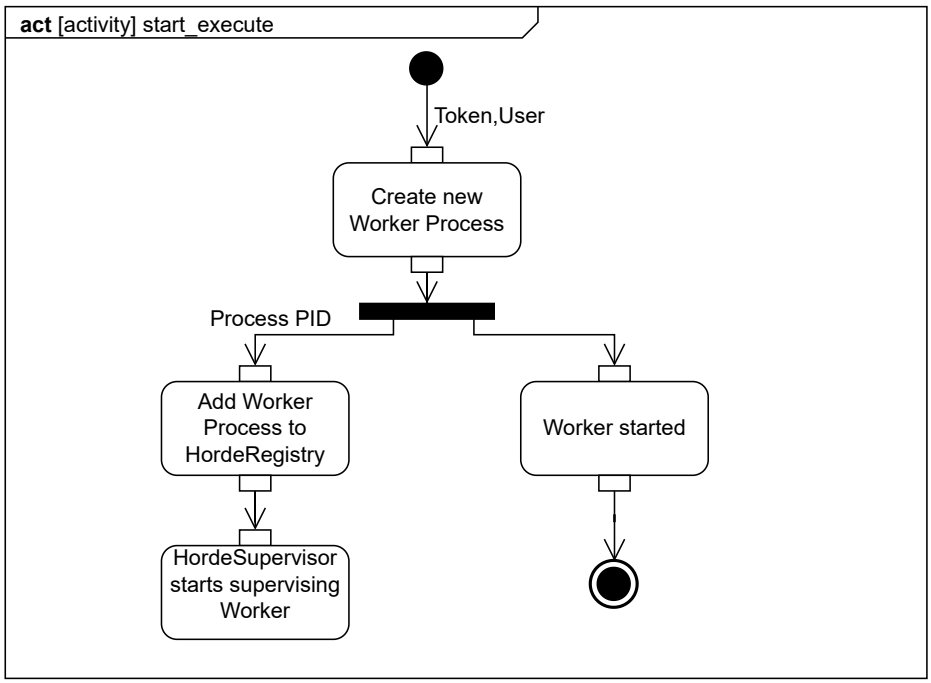

Figura 5.2: Diagrama de Actividades para la operación `start_execute`.

Un **Worker** es un proceso que se encarga de realizar una tarea. Como se ve en el diagrama de arranque de una ejecución, se crea el proceso, se le asigna a HordeSupervisor y se registran los datos de su ejecución para que esta pueda ser lanzada de nuevo en caso de un fallo. En la creación también se subscribe a las emisiones del resto de **Workers** de su mismo nodo. Una vez un **Worker** está creado, este puede comenzar a realizar las tareas que le corresponden.

Cada **Worker** implementa el *behaviour* GenServer, por lo que es un proceso que se mantiene activo a la espera de la llegada de mensajes. Este proceso mantiene el estado de las ejecuciones mediante el mecanismo de mandarse mensajes a sí mismo. Además, utiliza el mecanismo *Publish-Subscribe* para comunicarse con el resto de procesos que están en su mismo nodo. En la Figura 5.3 se muestra el diagrama de estado para un trabajador.

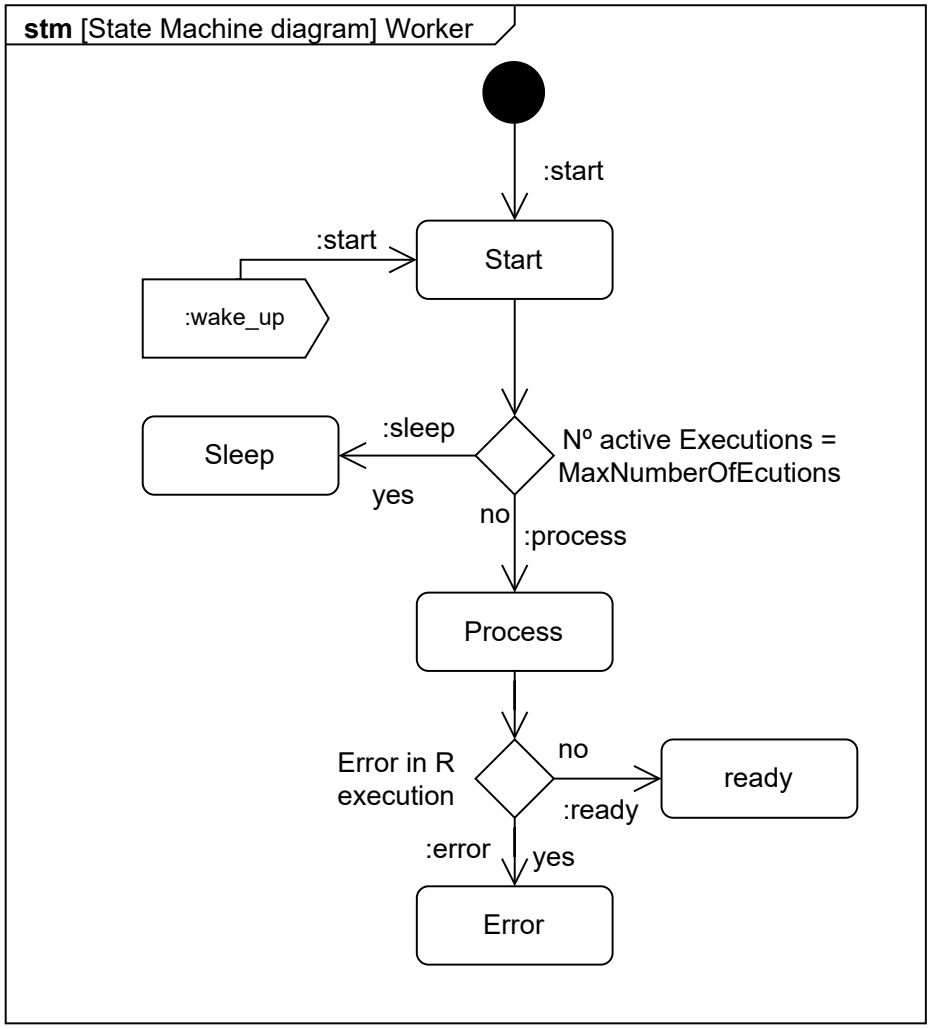

Figura 5.3: Diagrama de estados de **Worker**.

Como se observa, un **Worker** se pausa si ya se ha llegado al máximo de procesos en ejecución permitidos en el nodo. Una vez un **Worker** llega a este estado solo puede salir si recibe un mensaje de otro **Worker** indicándole que ha terminado y por tanto, es posible que exista un hueco para realizar la ejecución. Una vez un **Worker** llega al estado *Error* o *Ready*, el proceso finaliza.
Un **Worker** realiza las siguientes acciones en cada estado:

- Start: se notifica que ha habido cambios en la tabla a través de **PubSub** para que la vista sea notificada de los cambios.
- Process: se actualiza la tabla y se notifica a la vista. Se lleva a cabo la ejecución, conectando a la instancia R para poder ejecutar el código correspondiente. Además, se aumenta el número de procesos activos en el nodo.
- Ready: se actualiza la tabla, se notifica a la vista y se termina el proceso. Además, notifica al resto de **Workers** que ha finalizado. El número de procesos activos en el nodo disminuye.
- Error: se actualiza la tabla, notifica a la vista y se termina el proceso. Además, notifica al resto de **Workers** que ha finalizado. El número de procesos activos en el nodo disminuye.

- Sleep: el **Worker** se detiene esperando la llegada de un mensaje de otro **Worker** indicándole que ha finalizado.

5.1.2. Comunicación con la instancia R

Los diferentes **Workers** deben poder comunicarse con la instancia R que se encuentra instalada en el dispositivo que reside en el nodo. En el Capítulo de implementación se expone con más detalle una manera de realizar esta comunicación. En la Figura 5.4 se muestra el diagrama de actividades del componente responsable de llevar a cabo esta interacción.

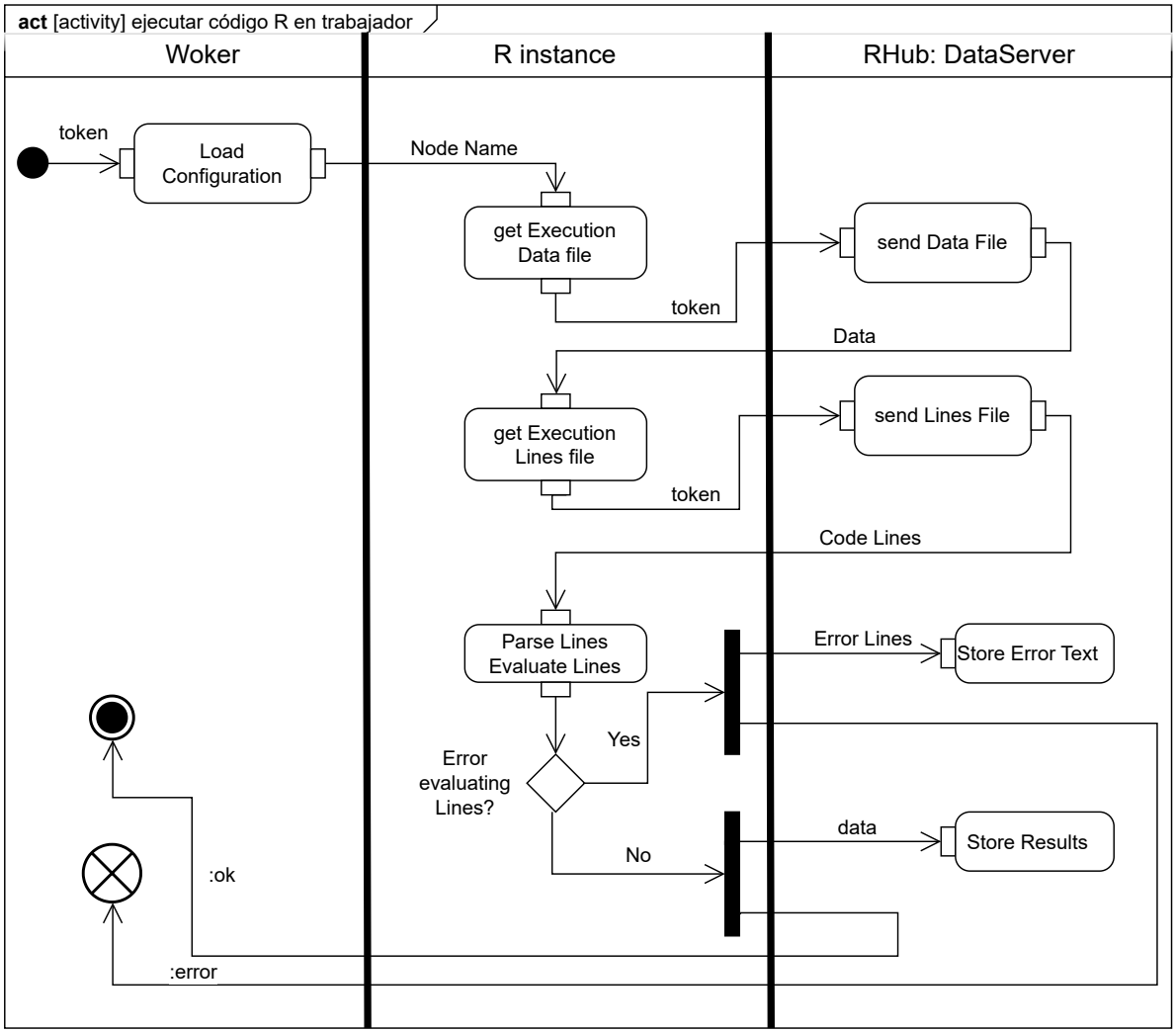

Figura 5.4: Diagrama de actividades del componente responsable de ejecutar código R.

5.1.3. NodeManager

En esta sección se ilustra el funcionamiento de **Nodemanager**. En la Figura 5.5 se representa y explica el escenario cuando ocurre el evento de caída de un nodo.

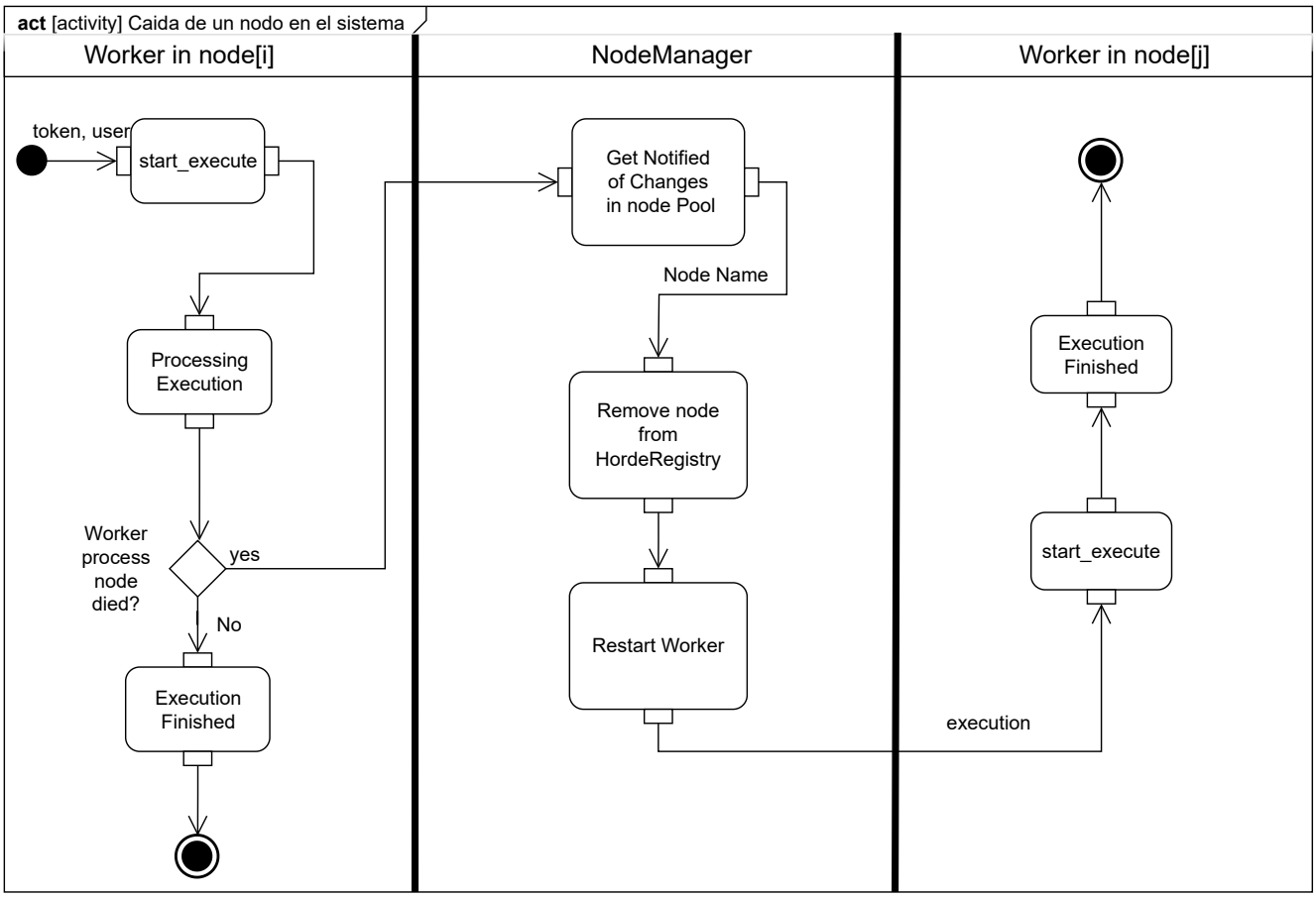

Figura 5.5: Diagrama de actividad para el reinicio de un **Worker** cuando cae un nodo.

En el evento de caída de un nodo ocurren las siguientes actividades:

1. **PoolSupervisor** es notificado de la caída de un nodo. Uno de sus componentes es un GenServer que escucha los mensajes `:node_up` y `:node_down` y modifica **HordeSupervisor** y **HordeRegistry** acorde a los cambios. Cuando cae un nodo el evento que ocurre es `:node_down`.

2. **HordeSupervisor** es notificado de la caída de los **Workers** del nodo que ha caído. Recupera la información de las ejecuciones que se estaban llevando a cabo mediante **HordeRegistry** y arranca los procesos **Worker** en otro nodo disponible.

3. Las ejecuciones se llevan a cabo en el resto de nodos de manera equivalente.

Es importante tener en cuenta que cuando un nodo cae, se ha de poner el contador de procesos de ese nodo a cero, ya que si no, en el caso de que vuelva a conectarse, al no existir ya los procesos, estos nunca finalizarían y no volvería a disminuir el contador.

5.1.4. View y ControllerWeb

ControllerWeb es el componente responsable de dirigir las conexiones entrantes por el protocolo HTTP a un nodo **RNodePool**, específicamente al componente del nodo correspondiente según la dirección, por tanto se puede decir que también es un *router*. Se definen las siguientes direcciones:

- `POST /execute`: Dirige la petición de una ejecución a **ExecutionR**, usando la API que ofrece este. En la petición se debe incluir como parámetro, el usuario y como cuerpo de la petición, el token que identifica la ejecución. Hay dos posibles respuestas:
  - `200, starting`: La ejecución se ha comenzado con éxito.
  - `200, already_started`: En el caso de que ya exista una ejecución con el mismo token en **RnodePool**.
- `POST /ready`: Dirige la ejecución a la API de **ExecutionR**. La petición debe contener como cuerpo, el token que la identifica. Las posibles respuestas son las siguientes:
  - `200, not_found`: No existe un token para la ejecución solicitada.
  - `200, ready`: La ejecución ha terminado.
  - `200, error`: La ejecución ha terminado con error.
  - `200, executing`: La ejecución está en progreso.
- `GET /`: Petición para solicitar la web que muestra el estado de las ejecuciones.
- `GET /dashboard`: Petición para solicitar el **Dashboard**, es decir, la web que muestra el estado del sistema.

Existen dos vistas: **ExecutionWeb** y **Dashboard**.

ExecutionWeb

Esta vista muestra la información sobre las ejecuciones que están ocurriendo en **RNodePool**. Para esto, utiliza un pequeño controlador que se encarga de escuchar un *broadcast* al que está subscrito y cuando recibe una actualización se recarga la tabla de ejecuciones que muestra está web. En el diagrama de actividades de la Figura 5.6 se muestra el mecanismo de actualización de la tabla en **ExecutionWeb**.

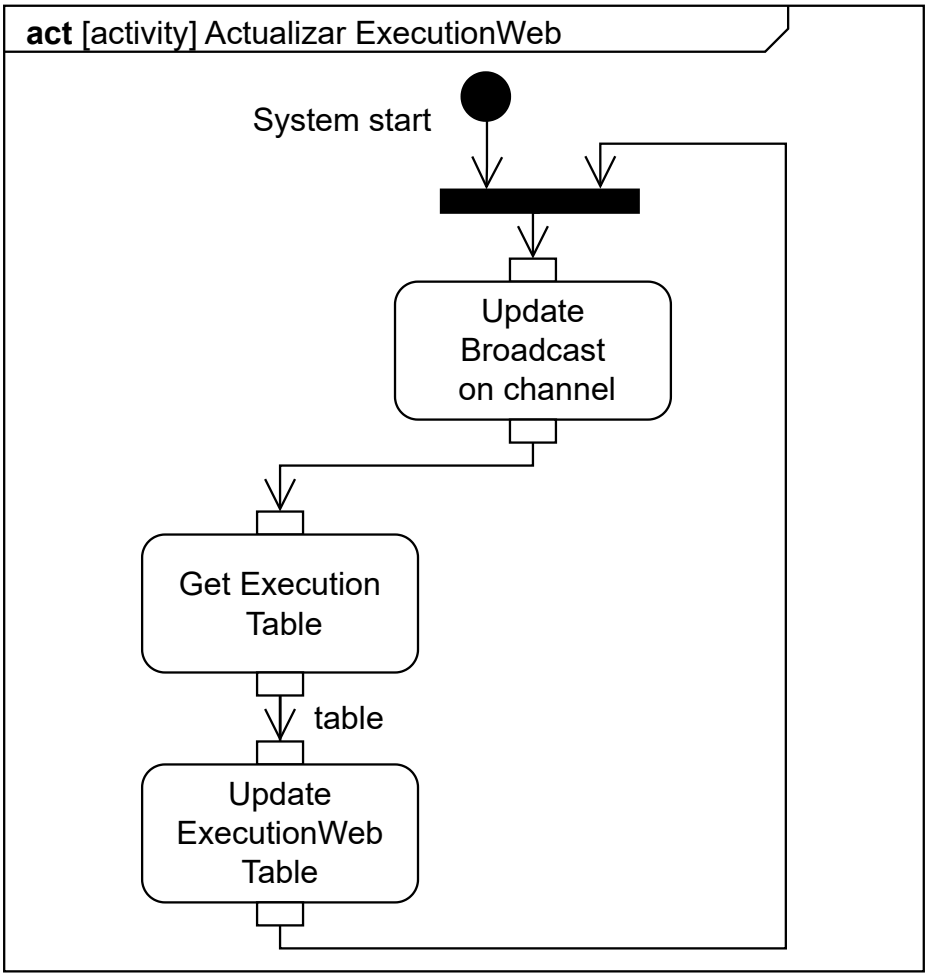

Figura 5.6: Diagrama de actividades de actualizar la vista de **ExecutionWeb**.

En la Tabla 5.1 se muestran los campos y contenidos que se espera que muestre esta web cuando se accede a ella. Las entradas de esta tabla deben estar ordenadas a partir del tiempo de inicio de la ejecución.

User	Token	Status	Start time	End time	Node
Usuario1	t1	error	2025-01-01 14:45:10	2025-01-01 14:50:10	n1@192.168.1.34
Usuario2	t2	ready	2025-01-01 14:50:11	2025-01-01 14:55:11	n2@192.168.1.34

Tabla 5.1: Tabla que se debe mostrar en **ExecutionWeb**.

Dashboard

El **dashboard** es un componente que proviene de una dependecia externa, por lo que no hay que tomar decisiones acerca del diseño de las vistas o de sus mecanismos para obtener la información. Este **Dashboard** obtiene información de los nodos, del sistema operativo, de los puertos web y de los sockets asignados. También recopila información sobre la ErlangVM, por lo que muestra las tablas y los procesos activos en el nodo. Debido a que proporciona información muy específica acerca del sistema, se debe utilizar como una herramienta de administración o de desarrollo, pero nunca mostrarse a un usuario del

sistema, en un producto final. El mecanismo de recopilación de estos datos, es un proceso que escucha periódicamente los cambios en el sistema y actualiza la información.

5.2. RHub

RHub está formado por dos componentes **RDispatcher** y **DataServer**.

- **RDispatcher** Conoce las direcciones y puertos de cada nodo del sistema. Su función es repartir la carga de las peticiones entre los nodos disponibles. Mantiene un único puerto de entrada al que se pueden enviar las peticiones y estas se reparten equilibrando las cargas de cada nodo.
- **DataServer** El servidor de datos se utiliza para intercambiar datos entre la instancia R del usuario y la instancia R en los nodos de **RNodePool**. La transferencia de los datos se realiza mediante el protocolo SSH. Existen dos directorios en la base de datos, `/in` y `/out`, uno para la entrada de datos y código y otro para los resultados o mensajes de error.

5.3. RPack

Este componente es el responsable de preprocesar los datos que introduce el usuario, conectar con **DataServer** y **RDispatcher** y por último generar el token de la operación o ejecución. Este componenete ofrece las siguientes funciones:

- `setupWorkspace(filepath)`: carga la configuración en la instancia R del usuario.
- `executeLines(lines,objects)`: crea una ejecución de las lineas de código especificadas sobre el objecto de datos que recibe. Envía mediante HTTP la petición a **RDispatcher** y los datos y lineas a **DataServer** mediante SSH. Esta operación es asíncrona, envía y no espera a una respuesta y devuelve inmediatamente el token de la ejecución.
- `distApply(X,MARGIN,...)`: divide los datos en varios fragmentos y procesa la operación. `MARGIN` y `...` representan los argumentos de `apply` del paquete base de R. Esta función crea varias ejecuciones de `apply` en función de `MARGIN` y utiliza la función `executeLines` para enviar cada una de estas ejecuciones.
- `distTApply(X,INDEX,...)`: divide los datos en varios fragmentos y procesa la operación. `INDEX` y `...` representan los argumentos de `tapply` del paquete base de R. Esta función crea varias ejecuciones de `apply`, ya que **tapply** se puede divir en varios **apply** sobre cada grupo que se especifica en `INDEX` y utiliza la función `executeLines` para enviar cada una de estas ejecuciones.
- `distLApply(X,...)`: divide los datos en varios fragmentos y procesa la operación. `...` representan los argumentos de `lapply` del paquete base de R. Esta función crea varias ejecuciones de `lapply` y utiliza la función `executeLines` para enviar cada una de estas ejecuciones.

- `distSApply(X,...)`: divide los datos en varios fragmentos y procesa la operación. `...` representan los argumentos de `sapply` del paquete base de R. Esta función crea varias ejecuciones de `sapply` y utiliza la función `executeLines` para enviar cada una de estas ejecuciones.
- `getResults(token)`: Obtención síncrona de los resultados de la ejecución u operación del token. Bloquea al usuario hasta que los resultados estén listos, descarga los resultados de **DataServer** y los devuelve al usuario.

Capítulo 6

Implementación

En este capítulo se detalla el entorno de desarrollo y los detalles de la implementación para cada parte del sistema. También se muestra el funcionamiento y la integración de dependencias externas en el sistema y qué funcionalidades y componentes que se han mencionado en el análisis y diseño son llevados a cabo por estas dependencias.

Todo el código desarrollado se puede consultar en el repositorio de github como se indica en el Anexo A.1. Además, en los anexos se puede encontrar una guía de instalación que especifica los detalles de la puesta en marcha y mantenimiento del sistema.

6.1. Implementación de RPack

RPack se distribuye como un paquete R llamado *distR*. Este paquete ha sido desarrollado en el entorno de desarrollo *RStudio* utilizando el paquete `devtools` [19] para la creación de este y `Roxygen2` [20] para la generación de la documentación.

Para disponer de una configuración global para la sesión, que no sea leída del archivo de configuración en cada instante, se establece un entorno del paquete, opaco al usuario. Internamente, este entorno funciona de manera equivalente al entorno de ejecución de una función, donde las variables internas de esta, son solo visibles a nivel de la función. En R se crea un entorno para el paquete de la siguiente manera:

```
pkg.env <- NULL

.onLoad <- function(...) {
  pkg.env <<-
    new.env(parent = emptyenv())
}
```

Source Code 6.1: Creación de un entorno para un paquete R

La función `.onload()` se ejecuta al iniciar el paquete y crea el entorno. Se asigna y accede al entorno de la siguiente manera:

```
assign("n_nodes", 1, envir = pkg.env)
pkg.env$n_nodes
```

Source Code 6.2: Asignación y acceso a un entorno para un paquete R

Una de las partes más importantes es cómo se ejecuta una linea de código del usuario u otras operaciones del paquete. Primero, se deben obtener los nombres de los objetos para poder crear la linea. Se ha optado por crear una función que envuelva al código y tenga como argumentos los datos necesarios y posteriormente llamar a esta función. A continuación se muestra la linea generada por la operación `executeLines()`.

```
objects<-list(a=1,b=2)
token<-executeLines("a+b",objects)
[1] "f<-function( a,b){a+b};f(a,b);"
```

Source Code 6.3: linea generada por la operación `executeLines()`

Otra parte importante del paquete que se debe ilustrar, es el mecanismo de las funciones distribuidas que ofrece. Como se ha explicado anteriormente, el objetivo es usar los mismos parámetros que la función equivalente en el paquete base R. A continuación se muestra cómo extraer estos argumentos y sus nombres, para posteriormente procesarlos.

```
arggs <- c(as.list(environment()), list(...))
arggsString <- argumentsToString(arggs)
```

Source Code 6.4: Obtención de argumentos cuando el número de estos es variable

Por último, para que se obtengan los resultados correctos, cada función distribuida tiene un mecanismo distinto de partición de datos:

- `distApply(X,MARGIN,...)`: La división varía dependiendo de la dirección en la que se recorren los datos:
  - `MARGIN=1`: Se dividen los datos en número de (nodos $\times 2$) fragmentos, cada fragmento contiene filas.
  - `MARGIN=2`: Se dividen los datos en número de (nodos $\times 2$) fragmentos, cada fragmento contiene columnas.
  - `MARGIN=c(1,2) o =c(2,1)`:Se dividen los datos en número de (nodos $\times 2$) fragmentos, recorriendo los datos primero por filas o primero por columnas respectivamente.

- `distTApply(X,...)`: Se dividen los datos por grupos, ya que se necesita que en cada fragmento esté el grupo completo para poder realizar los cálculos. Una vez divididos los datos funciona de manera equivalente a `distApply()`.
- `distLApply(X,...)`: Funciona de manera idéntica a `distTApply(X,...)` solo que las lineas que se generan utilizan `sapply` en vez de `lapply`.

Todas las dependencias de este paquete se pueden encontrar en CRAN (*The Comprehensive R Archive Network*) [21], y son las siguientes:

- uuid [22]: Este paquete genera tokens de identificación únicos. Estos tokens son una cadena de 128 bits.
- httr [23]: Este paquete ofrece funcionalidades para hacer peticiones HTTP. Un ejemplo de una petición con este paquete se muestra a continuación:

```
resp<-httr::POST("http://192.168.1.34/execute?user=Usuario2"),
    body = token)
```

Source Code 6.5: POST con el paquete httr

- RCurl [24]: Este paquete ofrece una implementación de *curl* [25] en R. En el sistema se usa para enviar y descargar archivos del servidor **DataServer** mediante el protocolo sftp. Un ejemplo del uso de este paquete se muestra a continuación:

```
RCurl::ftpUpload(
    file,paste(pkg.env$in_dir, token, ".rds", sep = ""),
    userpwd = pkg.env$userpass)
```

Source Code 6.6: POST con el paquete httr

- abind [26]: Este paquete ofrece funciones para agrupar arrays n-dimensionales, extendiendo la funcionalidad del R base permitiendo agrupar datos de más de dos dimensiones.

6.2. Implementación de RHub

Rhub está formado por dos componentes. Ambos componentes no requieren programar y solo es necesario configurarlos correctamente.

6.2.1. RDispacther

Para distribuir las peticiones web se utliza el servidor web *NGINX* [27], que es una potente herramienta para el despliegue de un servidor web. Como en este proyecto solo es de interés la parte de equilibrar

las peticiones entre varios nodos, se puede mantener la configuración por defecto tras su instalación, cambiando solo el archivo de configuración que se muestra a continuación, en el que se deben incluir la dirección y los puertos de los nodos del sistema.

```
#etc/nginx/sites-available/default
upstream loadbalancer {
  server 192.168.1.34:4001 weight=1;
  server 192.168.1.34:4002 weight=1;
  server 192.168.1.32:4001 weight=1;
}
server {
  location / {
    proxy_pass http://loadbalancer;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "Upgrade";
  }
}
```

Source Code 6.7: Archivo de configuración de NGINX

6.2.2. DataServer

Para almacenar los datos se utiliza un servidor SFTP. Este servidor utiliza los comandos ssh y sftp de los sistemas *Linux*.

6.3. Implementación de RNodePool

RNodePool es una aplicación desarrollada en *Elixir*, con algún componente escrito en C++. En esta tarea ha sido una ayuda inestimable los ejemplos mostrados en los tutoriales de Ricardo García Vega[1]. Para su desarrollo se utiliza el entorno Visual Studio Code con la extensión de lenguaje *Elixir*. Para administrar los paquetes se usa *Hex* [28], el administrador de paquetes en *Elixir* por excelencia. Para administrar el proyecto, resolver conflictos, y manejar las dependencias se utiliza *Mix* [29]. Esta aplicación elixir recibe el nombre de `DistR`.

Esta aplicación se ejecuta en cada nodo, y es la misma en cada nodo, por lo que aunque a continuación se va a mostrar la implementación de la aplicación, cuando se habla de varios nodos se hace referencia a varias instancias de la aplicación, cada una con sus procesos individuales, que se comunican entre ellas.

[1] «Bigardone.dev» URL: `https://bigardone.dev`.

En primer lugar, todas las aplicaciones deben tener un punto de entrada. En *Elixir* este módulo debe utilizar el paquete *Application* [30] que ofrece la funcionalidad necesaria. A continuación se muestra un fragmento de este módulo para mostrar cómo la aplicación crea un proceso supervisor para cada uno de los procesos hijo que aparecen.

```
defmodule DistR.Application do
  use Application
  def start(_type, _args) do
    children = [
      {Cluster.Supervisor, [topologies(),
        [name: BackgroundJob.ClusterSupervisor]]},
      DistR.HordeRegistry,
      DistR.HordeSupervisor,
      DistR.NodeObserver,
      {DistR.Execute.Queue, []},
      {Phoenix.PubSub, name: DistR.PubSub},
      DistRWeb.Telemetry,
      DistRWeb.Endpoint
    ]
    opts = [strategy: :one_for_one, name: DistR.Supervisor]
    Supervisor.start_link(children, opts)
  end
```

Source Code 6.8: Fragmento del módulo de inicio de la aplicación.

A continuación se muestra la implementación de cada uno de estos procesos y como interactúan entre ellos. Es importante entender que el programa Elixir está formado por módulos, y un módulo puede tener funciones que utilizan otros módulos, o puede poner en marcha un proceso trabajador o supervisor. Por eso, los procesos que se lanzan al inicio de la aplicación muestran aquellos que funcionan como un servicio, es decir, están siempre funcionando a lo largo del sistema, algunos de estos incluso compartiendo información entre varias instancias de la aplicación.

Primero, para formar el *pool* de nodos de manera automática se utiliza el paquete *Libcluster*, ya explicado en el Capítulo 3. Como ya se vio, este paquete ofrece varias estrategias para la formación automática de un *pool* de nodos. A continuación se muestra una definición de la topología, aunque a lo largo del desarrollo de la aplicación también se ha comprobado el funcionamiento de otras estrategias, como estrategias de detección automática de nuevos nodos.

```
defp topologies do
[
  example: [
    strategy: Cluster.Strategy.Epmd,
    config: [
      hosts: [
        :"n1@192.168.1.34",
        :"n2@192.168.1.32",
        :"n2@192.168.1.34"
      ]]]]
end
```

Source Code 6.9: Fragmento del módulo de inicio de la aplicación en una red privada.

A continuación vemos cómo la aplicación inicializa 3 procesos que se encargan de la supervisión de los nodos y registro de nodos. `DistR.HordeRegistry` utiliza el *behaviour* Horde.Registry que ofrece el paquete *Horde* introducido en el Capítulo 3. Este *behaviour* define la funcionalidad para registrar los procesos. `DistR.HordeSupervisor` utiliza el *behaviour* `Horde.DynamicSupervisor`. Ambos definen la siguiente función para definir los nodos que se están registrando y supervisando. Esta función añade el nodo y el resto de nodos al registro a ser supervisados, respectivamente.

```
defp members() do
    Enum.map([Node.self() | Node.list()], &{__MODULE__, &1})
end
```

Source Code 6.10: Función para definir los nodos del *pool*.

Además de tener el registro y el supervisor, también es necesario tener otro proceso que informe de los cambios que ocurren en el *pool* de nodos. `NodeObserver` utiliza el *behaviour* GenServer e implementa las respuestas a los mensajes *:node_up* y *:node_down*. A continuación se muestran las dos funciones *handle_info*.

```
def handle_info({:nodeup, _node, _node_type}, state) do
    set_members(HordeRegistry)
    set_members(HordeSupervisor)

    {:noreply, state}
end
def handle_info({:nodedown, node, _node_type}, state) do
    set_members(HordeRegistry)
    set_members(HordeSupervisor)
    Queue.restart_counter(node)
    {:noreply, state}
end
```

Source Code 6.11: Funciones *handle_call* de `NodeObserver`.

En el caso de que ocurra el evento de caída de un nodo, también hay que reiniciar el contador de procesos activos para ese nodo, por si se diera la situación de que se vuelve a conectar y su contador no esté a 0.

Para almacenar la información y el estado de las ejecuciones que lleva a cabo el sistema, se define *DistR.Execute.Queue*. Este módulo define un *behaviour*, es decir, una serie de *callbacks* que delega en otro módulo. De esta manera, si se quiere cambiar la implementación de la tabla, esta debe implementar una serie de funcionalidades que están definidas en este módulo. A continuación, en el código, se muestra la definición de uno de los *callbacks* definidos en el *behaviour*.

```
@adapter Application.compile_env(:dist_r, __MODULE__)[:adapter]
@callback insert(Execute.t()) :: result
@type result :: {:ok, Execute.t()} | {:error, term}
defdelegate insert(execute), to: @adapter
```

Source Code 6.12: Fragmento de la definición de uno de los *callbacks* de *DistR.Execute.Queue*.

En la primera linea, se indica a qué módulo se le delegan las operaciones definidas. Este mecanismo para definir el módulo en el que delega una funcionalidad se usa varias veces a lo largo de la aplicación. El nombre de este módulo aparece en el archivo de configuración del proyecto de la siguiente manera:

```
#./config/config.exs
config :dist_r, DistR.Execute.Queue,
  adapter: DistR.Execute.Queue.Nebulex
```

Source Code 6.13: Fragmento del archivo de configuración de la aplicación.

En el *callback* mostrado, se ve que se añade un tipo de dato `Execute`. Este tipo de dato está definido en el módulo `DistR.Execute`. A continuación se muestra un fragmento de la definición de esta estructura de datos.

```
    @type state :: :pending | :processing | :error | :ready |:sleep

    @type t :: %Execute{
          token: String.t(),
          state: state,
          start_time: DateTime.Calendar,
          end_time: DateTime.Calendar,
          node: String.t(),
          user: String.t()
    }
    defstruct [
        :token, :state,
        :start_time,
        :end_time,
        :node,
        :user
    ]
```

Source Code 6.14: Fragmento de la defínción del tipo de dato `Execute`.

En la configuración de la aplicación se muestra que la implementación de la tabla se delega al módulo `DisR.Execute.Queue.Nebulex`. Este módulo utiliza el paquete *Nebulex* [31], un paquete que ofrece un *cache* distribuido. En esta aplicación se utiliza *Nebulex* con funcionamiento replicado, es decir, con la información que almacena replicada en cada nodo, de tal manera que cuando ocurre un cambio en la tabla en uno de los nodos, se avisa al resto de tablas que modifiquen sus entradas. De esta manera, se puede garantizar que la información sobre las ejecuciones esté disponible en todos los nodos. Por el diseño de esta aplicación en concreto, no se va a dar la situación de que dos procesos intenten escribir en la misma entrada de la tabla, pero en el caso de que ocurriera, el paquete garantiza escrituras atómicas en la medida de lo posible. A continuación, se muestra la definición de las propiedades del módulo y la implementación de una función:

```
    use Nebulex.Cache,
      otp_app: :dist_r,
      adapter: Nebulex.Adapters.Replicated

    alias DistR.{Execute, Execute.Queue}
    require Logger
    @behaviour Queue

    @impl Queue
    def restart_counter(node) do
      :ok = put(node, 0)

      {:ok, 0}
    end
```

Source Code 6.15: Fragmento de la implementación del módulo `DisR.Execute.Queue.Nebulex`.

Una vez tenemos el resto de componentes, la parte más importante de la aplicación son los **Workers**, responsables de las ejecuciones y de recorrer los estados de cada ejecución. A continuación se muestra la especificación de un trabajador y como se le añade a `HordeRegistry`.

```
defp worker_spec(%Execute{token: token} = execute) do
    %{
        id: {Worker, token},
        start: {Worker, :start_link,
            [[execute: execute, name: via_tuple(token)]]},
        type: :worker,
        restart: :transient
    }
end

defp via_tuple(token) do
    {:via, Horde.Registry, {HordeRegistry, {Execute, token}}}
end
```

Source Code 6.16: Definición del proceso **Worker** .

Como se ha mencionado en el diseño, un **Worker** utiliza el *behaviour* GenServer. Este tiene por tanto, una función de inicialización que inicializa el GenServer, y varias funciones que reciben mensajes y realizan una acción. A continuación se muestra la función de `init(execute)`.

```
    @impl GenServer
    def init(execute) do
      schedule(:start, 1)
      PubSub.subscribe(DistR.PubSub, to_string(node()))
      {:ok, execute}
    end
```

Source Code 6.17: Definición del proceso **Worker**.

Cuando se inicializa el **Worker**, se envía a sí mismo el mensaje `:start`. Se utiliza la función `schedule` para establecer un retraso exacto si se quiere estudiar el funcionamiento de manera controlada. Por último, el **Worker** se subscribe a `DistR.PubSub`, el mecanismo de *Publish-Subscribe*, subscribiéndose al tema que lleva el nombre del nodo, de manera que todos los trabajadores del mismo nodo estén subscritos al mismo tema y puedan notificarse unos a otros.

A continuación se muestra el código de una de las funciones `handle_info` de este GenServer, la que recibe el mensaje `:process` y es la responsable de delegar la ejecución del código R del usuario:

```
def handle_info(:process, %Execute{token: token} = execute) do
  {:ok, new_execute} =
    execute
    |> Execute.with_node()
    |> Execute.with_processing_state()
    |> Queue.update()
    broadcast(@topic_web,:ok)
  case System.cmd("./execute_command", [token]) do
      {_,0}->
        schedule(:ready, 1)
        {:noreply, new_execute}
      {_,1}->
        schedule(:error, 51)
        {:noreply, new_execute}
  end
end
```

Source Code 6.18: Comportamiento de **Worker** al recibir el mensaje `:process` .

En esta función se procesa la ejecución, se modifica en la tabla de ejecuciones y se notifica a la vista mediante una emisión de que hay cambios en la tabla. Después de esto, se utiliza un comando del sistema para lanzar el proceso `execute_command`. Una vez se ha ejecutado continúa el flujo del trabajador al estado error o al estado terminado dependiendo de lo que devuelva `execute_command`.
`execute_command` es un programa escrito en C++ que mediante las librerías R *Rcpp* [32] y *RInside* [33]

crea una instancia R con la que se interactúa desde código C++. A continuación se muestra un extracto del código en el que se crea la instancia y se leen las lineas y los datos de **DataServer**:

```
string token=argv[1];
RInside R(argc, argv);
...
R.parseEvalQ("lines<-getURL(lines_file, userpwd = userpass)");
string try_eval="tryCatch({"
    "fout<-eval(parse(text=substr(lines,1,nchar(lines)-2)))},"
    "error = function(e){"
    "e<-as.character(e);"
    "ftpUpload(I(e),error_d, userpwd = userpass);"
    " stop()})";
R.parseEvalQ(try_eval);
...
R.parseEvalQ("saveRDS(fout,file=fil)");
R.parseEvalQ("ftpUpload(fil,out_d, userpwd = userpass)");
```

Source Code 6.19: Funcionamineto de `RInside` para ejecutar lineas de R .

Se puede ver cómo la librería *RInside* permite crear una instancia de R en la que se pueden definir variables e indicar qué lineas se quieren ejecutar. Realizar esta conexión desde C++ otorga mayor control y si en el futuro se quisiera añadir un proceso elixir que monitorice este proceso, la integración con código C++ es mucho más sencilla.

Una vez mostrada la lógica de la aplicación, pasamos ahora a la parte de los controladores y las vistas. `DistR` es una aplicación desarrollada con el *framework* `Phoenix`, utilizado ampliamente en el desarrollo de aplicaciones web. Todas las aplicaciones `Phoenix` deben tener un `Endpoint`, en el que se definen distintas propiedades del servicio web, como las características de la sesión o los distintos *plugs* que se utilizan en la aplicación. Las aplicaciones *Phoenix* permiten implementar funcionalidades mucho más avanzadas que las que se necesitan en este trabajo, por lo que muchas partes del código esqueleto que se genera al empezar un proyecto, queda inutilizado. Uno de estos *plugs* que si tiene importancia en la aplicación es el *router*. En este módulo se definen los destinos para las diferentes entradas de peticiones. A continuación se muestra un fragmento de `DistRWeb.Router`. Como ya se ha mencionado previamente, el `dashboard` es una herramienta muy potente que expone información del funcionamiento de la aplicación por lo que solo se incluye en el entorno de desarrollo. En caso de querer incluirlo en un producto se debería restringir mediante un inicio de sesión.

```
    scope "/", DistRWeb do
        pipe_through :browser
        post "/execute", WebHookController, :execute
        post "/ready", WebHookController, :ready
        post "/delete", WebHookController, :delete
        live "/", PageLive, :index
    end
    if Mix.env() == :dev do
        import Phoenix.LiveDashboard.Router
        scope "/" do
          pipe_through :browser
          live_dashboard "/dashboard",metrics: DistRWeb.Telemetry
        end
    end
```

Source Code 6.20: Fragmento de código de `DistRWeb.Router`.

El módulo `DistRWeb.WebHookController` se encarga de recibir las conexiones y ejecutar la función indicada dependiendo de la dirección de la conexión. A continuación se muestra la función que arranca una ejecución en el sistema. Utiliza la API de la lógica de la aplicación, específicamente la función `DistR.start_execute()` para crear una nueva ejecución.

```
    def execute(conn, _opts) do

        {:ok, body, conn} = read_body(conn, length: 10_000)
        {code,msg}=case DistR.is_ready(body) do
          {:error, _}->
            DistR.start_execute(body,conn.params["user"])
            {200,"starting"}
          {:ok, _}->
            {200,"already started"}
        end
        send_resp(conn,code, msg)
    end
```

Source Code 6.21: Fragmento de código del controlador.

Otra parte de la aplicación `Phoenix` es la vista de **ExecutionWeb**. Esta vista recibe la tabla de ejecuciones cuando uno de los trabajadores notifica cambios mediante el mecanismo de *Publish-Subscribe*. Para realizar el código de la vista se utiliza *html + Embedded Elixir*. A continuación se muestra un fragmento de código de **ExecutionWeb**:

```
<%= unless @table == [] do %>
        <%= for i <- @table do %>
        <tr>
          <td scope="row" class="...">
            <%= i.user %>
          </td>
          <td scope="row" class="...">
            <%= i.token %>
          </td>
          ...
        </tr>
        <% end %>
      <% end %>
```

Source Code 6.22: Ejemplo de combinación de *html* y *Embedded Elixir* en **ExecutionWeb**.

En la figura 6.1 se muestra la vista de la tabla en el navegador.

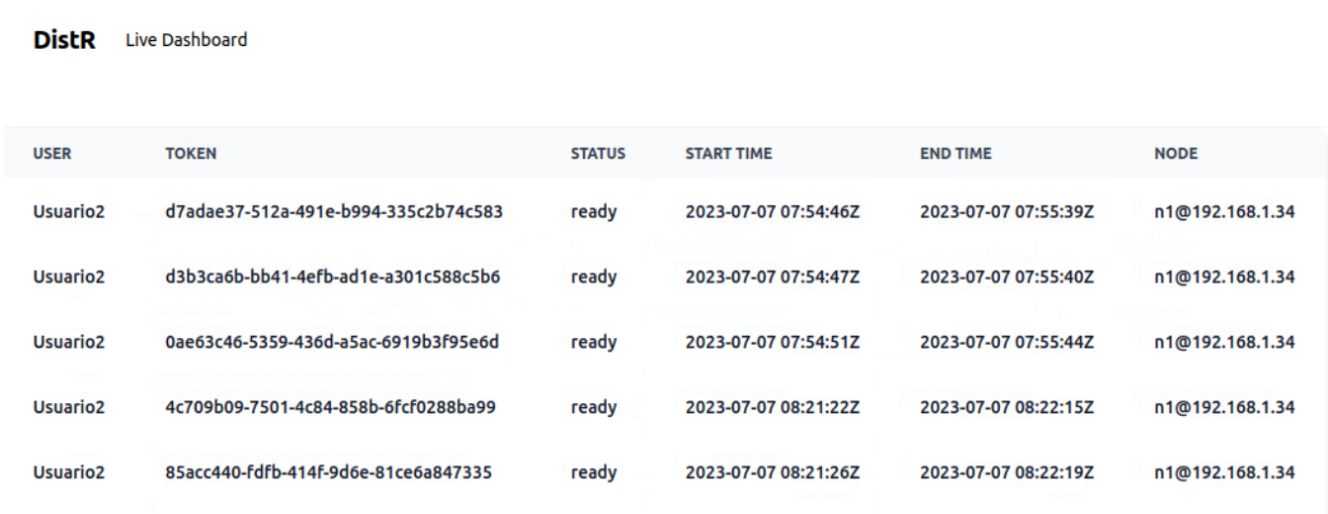

Figura 6.1: Vista de **ExecutionWeb**.

En la figura 6.2 se muestra una de las vistas del `dashboard`. En esta se presenta información sobre las versiones y los límites del sistema. Además, mediante el desplegable situado en la esquina superior derecha se puede cambiar entre los diferentes nodos.

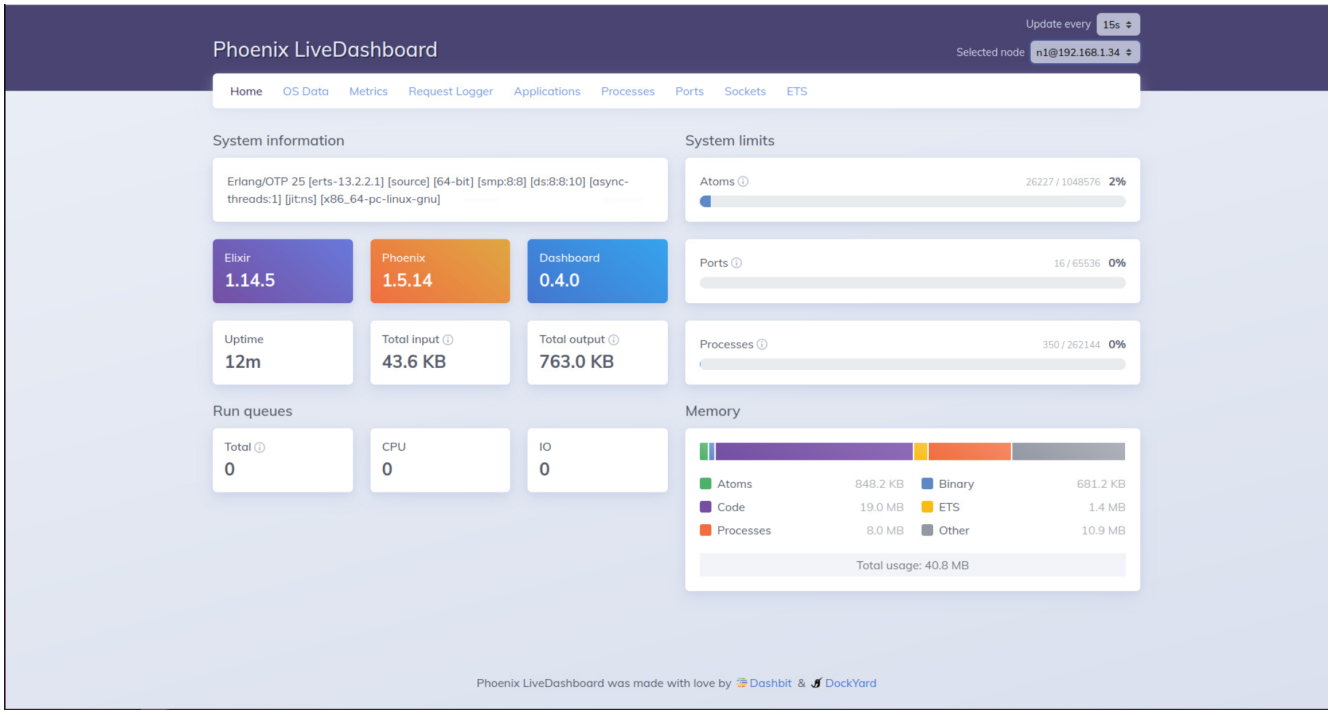

Figura 6.2: Vista del `dashboard`.

En cuanto a las dependencias del proyecto, la mayoría son dependencias para que `Phoenix` funcione. A continuación se muestran las dependencias del proyecto.

```elixir
defp deps do
[
  {:phoenix, "~> 1.5.9"},
  {:phoenix_live_view, "~> 0.15.1"},
  {:floki, ">= 0.30.0", only: :test},
  {:phoenix_html, "~> 2.11"},
  {:phoenix_live_reload, "~> 1.2", only: :dev},
  {:telemetry_metrics, "~> 0.4"},
  {:telemetry_poller, "~> 0.4"},
  {:jason, "~> 1.0"},
  {:plug_cowboy, "~> 2.0"},
  {:phoenix_live_dashboard, "~> 0.1"},
  # app logic deps
  {:libcluster, "~> 3.3"},
  {:horde, "~> 0.8.3"},
  {:nebulex, "~> 2.1"}
]
end
```

Source Code 6.23: Dependencias del proyecto.

Capítulo 7

Pruebas y *Benchmarking*

En este capítulo se realizan las pruebas necesarias para comprobar el correcto funcionamiento del sistema. Se realizan 2 tipos de pruebas, pruebas del funcionamiento y pruebas del rendimiento. Además, en último lugar se realiza un estudio de la usabilidad del sistema por un usuario habitual del lenguaje R.

Para estas pruebas se usa una configuración de 3 nodos. Cada nodo es un *minipc* con las siguientes características:

1. CPU: Intel Celeron(R) CPU N3000 @1.04GHz $\times$ 2
2. RAM: 4Gb

Para garantizar que el sistema no se ralentiza por otros motivos, se han alojado los componentes de **Rhub** en otra máquina que no va a pertenecer al *pool* de nodos. Para las pruebas de funcionamiento no se usan todos los nodos, ya que tener uno o dos nodos es suficiente para comprobar el funcionamiento. El paquete `distR` desde donde se lanzan las ejecuciones mediante una instancia R, está alojado en otra maquina en la misma red. Todos los componentes del sistema están alojados en la misma red local.

7.1. Pruebas del funcionamiento correcto del sistema

Existen dos tipos de pruebas, las pruebas del correcto funcionamiento de las instrucciones R y las pruebas del funcionamiento del sistema distribuido ante situaciones como un número elevado de procesos o la caída de un nodo.

7.1.1. Pruebas del funcionamiento del R

Para estas pruebas es suficiente comprobar el funcionamiento de un nodo, ya que el objetivo es ver si los resultados que se calculan en el nodo son los esperados por el usuario. En este proyecto se han realizado muchas más pruebas que las que se muestran a continuación, estas se pueden encontrar en la carpeta `vignettes/` del paquete R.

Prueba 1: Establecer configuración

Prueba 1	Establecer configuración
Objetivo	El usuario establece la configuración del paquete y se le muestra la dirección web donde puede ver las ejecuciones.
Actor	Profesor
Precondición	Ninguna
Acción esperada	Se establece la configuración y se muestra la dirección web
Resultado	**Positivo**

Tabla 7.1: Prueba 1 Establecer configuración

Para las siguientes pruebas se muestra el fragmento de código que se ejecuta y se comprueba que coincide con los resultados esperados ejecutando las mismas instrucciones en la instancia local del R. En los casos de error se comprueba que el error se muestra.

Prueba 2: Ejecutar Lineas de Código Remoto y obtención de los resultados correctos

```
a<-c(1,2,3)
b<-c(2,3,4)
objects<-list(a=a,b=b)
token<-executeLines("c<-a+b;c*2",objects)
c<-getResults(token)
identical(c,(a+b)*2)
```

Source Code 7.1: Prueba 2: Ejecutar Lineas de Código Remoto y obtención de los resultados correctos

Resultado positivo. En esta prueba se comprueba que ambas ejecuciones en local y en remoto son iguales y que se pueden introducir varías lineas separadas por punto y coma en la cadena de código que se envía.

Prueba 3: Ejecutar Lineas de Código Remoto y obtención de los resultados con error de ejecución

```
...
token<-executeLines("c<-a+b,c*2",objects)
c<-getResults(token)
```

Source Code 7.2: Prueba 3: Ejecutar Lineas de Código Remoto y obtención de los resultados con error de ejecución

Resultado esperado. Se le indica al usuario que hay una coma inesperada después de `c<-a+b,`.

Prueba 4: Ejecutar `apply` distribuido con MARGIN=1 y obtención de los resultados correctos

```
funct<-function(data,c1,c2){return(mean(data)+c1+c2)}
dat<-matrix(c(1:20),ncol=2)
token<-distApply(dat,FUN=funct,MARGIN=1,c1=1,c2=2)
c<-getResults(token)
identical(unname(c),apply(dat,FUN=funct,MARGIN=1,c1=1,c2=2))
```

Source Code 7.3: Prueba 4: Ejecutar `apply` distribuido con MARGIN=1 y obtención de los resultados correctos

Resultado positivo. Se comprueba que se obtiene el mismo resultado numérico. Además se comprueba el nodo que lo ejecuta y se ve que la tarea se ha divido en número de nodos$\times$2.

Prueba 5: Ejecutar `apply` distribuido con MARGIN=2 y obtención de los resultados correctos

```
...
token<-distApply(dat,FUN=funct,MARGIN=2,c1=1,c2=2)
c<-getResults(token)
identical(unname(c),apply(dat,FUN=funct,MARGIN=2,c1=1,c2=2))
```

Source Code 7.4: Prueba 5: Ejecutar `apply` distribuido con MARGIN=2 y obtención de los resultados correctos

Resultado positivo. Se comprueba además, que al haber solo dos columnas la división solo puede hacerse en 2 fragmentos.

Prueba 6: Ejecutar `apply` distribuido con MARGIN=c(1,2) y obtención de los resultados correctos

```
...
token<-distApply(dat,FUN=funct,MARGIN=c(1,2),c1=1,c2=2)
c<-getResults(token)
identical(unname(c),apply(dat,FUN=funct,MARGIN=c(1,2),c1=1,c2=2))
```

Source Code 7.5: Prueba 6: Ejecutar `apply` distribuido con MARGIN=c(1,2) y obtención de los resultados correctos

Resultado positivo. Se devuelve un *array* de la misma dimensión que la matriz de datos.

Prueba 7: Ejecutar `tapply` distribuido y obtención de los resultados correctos

```
...
price <- data_set$price
type <- factor(data_set$type,
               labels = c("toy", "food", "electronics", "drinks"))
token<-distTApply(price,INDEX=type,FUN=mean)
c<-getResults(token)
tap<-tapply(price,type,mean)
```

Source Code 7.6: Prueba 7: Ejecutar `tapply` distribuido y obtención de los resultados correctos

Resultado positivo. Se devuelve la media de la variable `price` para cada categoría.

Prueba 8: Ejecutar `Lapply` distribuido y obtención de los resultados correctos

```
l<-list(1,2,3,1,2,3,7,8,1,3,6,9)
t<-function(data){return(data*2)}
token<-distLApply(l,FUN=t)
c<-getResults(token)
identical(c,lapply(l,FUN=t))
```

Source Code 7.7: Prueba 8: Ejecutar `lapply` distribuido y obtención de los resultados correctos

Resultado positivo. Se devuelve una lista con cada elemento multiplicado por 2.

Prueba 9: Ejecutar `sapply` distribuido y obtención de los resultados correctos

```
l<-list(1,2,3,1,2,3,7,8,1,3,6,9)
t<-function(data){return(data*2)}
token<-distSApply(l,FUN=t)
c<-getResults(token)
identical(c,sapply(l,FUN=t))
```

Source Code 7.8: Prueba 9: Ejecutar `sapply` distribuido y obtención de los resultados correctos

Resultado positivo. Se devuelve un vector con cada elemento de la lista multiplicado por 2.

Prueba 10: comprobación del estado *sleep* cuando se alcanza el máximo de procesos activos

En esta prueba se comprueba que el número de procesos activos en un nodo no excede el número máximo establecido. Para esto se ejecutan las siguientes instrucciones en R y se comprueba en la **ExecutionWeb** el número de procesos activos. Se utiliza la instrucción `Sys.sleep` para poder ver con más claridad qué está haciendo cada ejecución. En esta configuración cada nodo puede tener hasta 3 procesos activos.

```
token<-executeLines("Sys.sleep(10);a+b",objects)
token<-executeLines("Sys.sleep(10);a+b",objects)
token<-executeLines("Sys.sleep(10);a+b",objects)
token<-executeLines("Sys.sleep(10);a+b",objects)
```

Source Code 7.9: Prueba 10: comprobación del estado *sleep* cuando se alcanza el máximo de procesos activos.

Como se muestra en la Figura 7.1, se inician 3 ejecuciones. Una vez una de estas ejecuciones termina, se reanuda la que está pausada.

USER	TOKEN	STATUS
Usuario2	5c2cd676-94b7-480a-ab81-c93fe006e9db	processing
Usuario2	cccb6461-57cc-48eb-a1a3-c01977648ca7	processing
Usuario2	fdd5a544-c5f5-4948-8e01-23057d57bf65	processing
Usuario2	1cb6db8d-842a-4c7d-9d85-6c77720845ef	sleep

Figura 7.1: Prueba 10: comprobación del estado *sleep* cuando se alcanza el máximo de procesos activos

Prueba 11: Reanudación de los procesos en el caso de que caiga un nodo

En esta prueba se comprueba que en el caso de caída de un nodo, sus ejecuciones pendientes son reanudadas por los otros nodos y estas terminan con éxito. Para esta prueba se utilizan dos nodos y se detiene forzosamente uno de ellos. Cuando un nodo se detiene, en la tabla se debe ver que cambia de nodo. El orden de eventos para comprobar el funcionamiento es el siguiente:

1. Se arrancan los dos nodos, n2 y n3.
2. Se envían un total de 6 ejecuciones con demora mediante `Sys.sleep`.

3. Se comprueba que ambos nodos tienen 3 procesos procesando datos R.
4. Se detiene forzosamente el nodo n2.
5. Se comprueba que las ejecuciones del nodo n2 se transfieren al nodo n3, pero no arrancan debido a que ya existen 3 procesos activos en ese nodo.
6. Al finalizar los 3 procesos se arrancan los que pertenecían al nodo n2.

Como se muestra en la Figura 7.2, se inician 6 ejecuciones y cada nodo comienza a procesar 3 ejecuciones. Cuando se detiene forzosamente el nodo n2 (Figura 7.3), se observa cómo las ejecuciones pasan a pertenecer al nodo n3 y las 3 ejecuciones se pausan. A medida que n3 termina sus ejecuciones se arrancan aquellas que tiene pausadas, como se ha comprobado en la prueba anterior.

USER	TOKEN	STATUS	START TIME	END TIME	NODE
Usuario2	726dd55d-2f36-4491-af7b-7e500e6e906f	processing	2023-07-09 18:24:29Z		n2@192.168.1.32
Usuario2	e069a3c6-3578-4b23-9657-19ac3930ba43	processing	2023-07-09 18:24:30Z		n3@192.168.1.31
Usuario2	9555a22e-e07b-4a6f-bd07-5aea1a2421a2	processing	2023-07-09 18:24:31Z		n3@192.168.1.31
Usuario2	e350a6aa-5454-4673-b9fd-935ffc442e87	processing	2023-07-09 18:24:32Z		n2@192.168.1.32
Usuario2	c2a15d7e-8520-4536-b27a-14d245d681d2	processing	2023-07-09 18:24:33Z		n2@192.168.1.32
Usuario2	62aee05f-dcf8-4c8a-9b0a-6aae40a66285	processing	2023-07-09 18:24:34Z		n3@192.168.1.31

Figura 7.2: Prueba 11: Arranque de ejecuciones en 2 nodos.

USER	TOKEN	STATUS	START TIME	END TIME	NODE
Usuario2	726dd55d-2f36-4491-af7b-7e500e6e906f	sleep	2023-07-09 18:24:29Z		n3@192.168.1.31
Usuario2	e069a3c6-3578-4b23-9657-19ac3930ba43	processing	2023-07-09 18:24:30Z		n3@192.168.1.31
Usuario2	9555a22e-e07b-4a6f-bd07-5aea1a2421a2	processing	2023-07-09 18:24:31Z		n3@192.168.1.31
Usuario2	e350a6aa-5454-4673-b9fd-935ffc442e87	sleep	2023-07-09 18:24:32Z		n3@192.168.1.31
Usuario2	c2a15d7e-8520-4536-b27a-14d245d681d2	sleep	2023-07-09 18:24:33Z		n3@192.168.1.31
Usuario2	62aee05f-dcf8-4c8a-9b0a-6aae40a66285	processing	2023-07-09 18:24:34Z		n3@192.168.1.31

Figura 7.3: Prueba 11: Ejecuciones del nodo n2 se reinician en el nodo n3.

Prueba 12: Conexión automática de los nodos

En esta prueba se comprueba que la conexión de nuevos nodos al *pool* de nodos se realiza de manera automática cuando se arranca un nuevo nodo. Para esta prueba se forma el *pool* de nodos con el nodo

n2 y n3 y se arranca el nodo n4. En la Figura 7.4 se observa en el *Dashboard* como el nodo n4 pasa automáticamente a formar parte del *pool*.

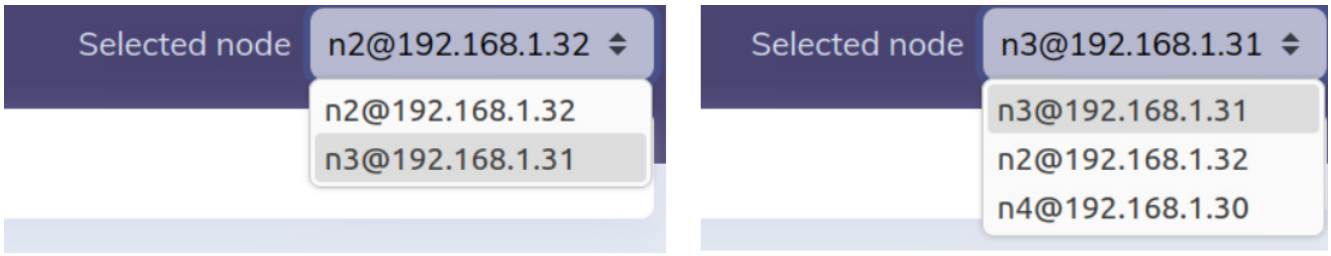

Figura 7.4: Prueba 12: Conexión automática de los nodos.

7.2. Pruebas del rendimiento del sistema

En esta batería de pruebas se busca estudiar si existe una mejora de rendimiento al usar el sistema que se ha desarrollado en este trabajo. Para todas estas pruebas se utilizan 3 nodos con las características indicadas en la introducción del capítulo. Se van a realizar tres tipos de pruebas:

- Pruebas del coste de envío de datos y procesamiento de las peticiones.
- Pruebas de rendimiento de 3 máquinas en comparación con 1 máquina de las mismas especificaciones.
- Pruebas de rendimiento de máquinas en comparación con 1 máquina con mayores prestaciones. Las máquinas utilizadas como nodos no son un indicador de la potencia media de una máquina actual, por lo que se compara con una máquina actual con mayores prestaciones.

7.2.1. Pruebas del coste de envío de datos y procesamiento de las peticiones

En esta sección se mide el coste de traspaso de los datos y procesamiento de las peticiones. Para esto, se definen varias tareas artificiales con un coste computacional en segundos que se modifica para estudiar el comportamiento. El tamaño de los datos también se va aumentando a medida que aumenta el tiempo. El objetivo es mostrar como la proporción de tiempo que se consume en traspasar los datos y procesar las peticiones disminuye a medida que se aumenta el tiempo de computación. La conexión entre el ordenador con la instancia R y el nodo tiene lugar a través de un *switch Gigabit*. Todas las instrucciones se ejecutan en un único nodo, es decir no hay ningún grado de paralelismo o de división de las operaciones para aislar los tiempos de procesamiento.

El tiempo que tarda la ejecución se define mediante la operación `Sys.sleep`, es decir, el nodo lo único que hace es simular que ejecuta un código que le lleva este tiempo y se queda esperando. En la Tabla 7.2 se muestran los escenarios que se proponen para una combinación de tiempos y tamaño de datos:

	1	2	3	4	5	6	7	8	9
tiempo de ejecución (s)	0	10	20	30	40	50	60	70	80
tamaño (Mb)	4	8	12	16	20	24	28	32	36

Tabla 7.2: Escenarios de prueba diseñados para comprobar el rendimiento del sistema.

Cada escenario se repite 10 veces y se promedian los tiempos entre cada repetición. En la Tabla 7.3 se muestran los valores medios de tiempo total y en la Figura 7.5 se muestra un gráfico con el aumento del tiempo total en función del tiempo de ejecución. Como se puede observar en el gráfico y la tabla, al aumentar el tiempo de ejecución, este representa un porcentaje más alto del tiempo total, como es de esperar. Esto quiere decir que a medida que la carga de trabajo aumenta, mejor es el rendimiento del sistema distribuido y menos presente está el tiempo de traspaso de datos y procesamiento de las peticiones en el sistema.

	1	2	3	4	5	6	7	8	9
tiempo de ejecución (s)	0	10	20	30	40	50	60	70	80
tiempo total (s)	6.72	17.39	28.12	38.91	50.41	61,40	72.25	83.26	94.37

Tabla 7.3: Resultados medios con 10 repeticiones para los escenarios propuestos

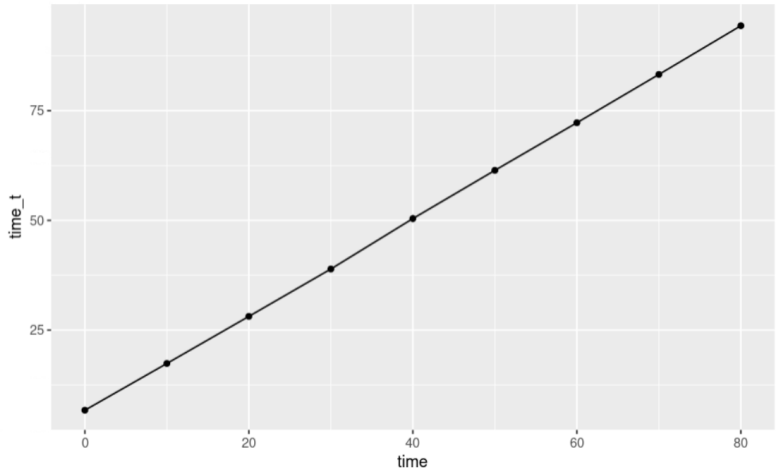

Figura 7.5: Comparación del rendimiento en los diez escenarios distintos con 10 repeticiones. El gráfico dispone de lineas de error pero son inapreciables por las dimensiones del gráfico.

7.2.2. Pruebas de rendimiento de 3 máquinas frente a una equivalente

Para estas pruebas se genera una tarea artificial que se divide en tres partes. Además, se fija el número máximo de ejecuciones simultáneas en cada nodo a 1, ya que se quiere comparar el rendimiento de tres nodos con un *minipc* equivalente. Ni el *minipc* ni cada nodo pueden realizar varias tareas simultáneas. Para valorar esto, se define una tarea cuyo coste computacional es de 60 segundos y el tamaño de los datos es de 64Mb. De esta manera se dividen los datos en 3 particiones de 21 Mb y el tiempo de ejecución de cada partición se fija a 20 segundos. Al igual que en el caso anterior, se repite el escenario 10 veces. En la

Figura 7.6 se muestra un gráfico de cajas y bigotes para el rendimiento de los 3 nodos dividiendo la tarea. Como se puede observar una ejecución que tardaría 60 segundos, queda reducida a 37 segundos cuando se divide la tarea en 3 partes y estas se reparten. Además se observa como hay muy poca variabilidad entre los tiempos totales, lo que quiere decir que el comportamiento del sistema en cuanto al traspaso de datos y manejo de ejecuciones es temporalmente consistente, es decir, suele tardar lo mismo para una tamaño de datos constante.

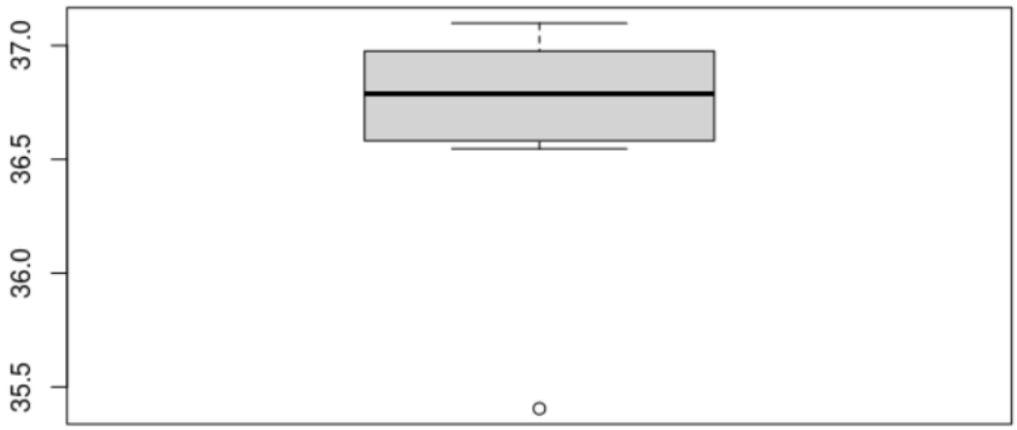

Figura 7.6: Rendimiento de los nodos en una tarea dividida de 60 segundos.

7.2.3. Pruebas de rendimiento de 3 máquinas frente a 1 máquina con mayores prestaciones

En esta prueba de rendimiento se compara el *pool* de nodos con 3 tareas simultáneas frente a una máquina con las siguientes prestaciones:

- CPU: AMD ryzen 7 4800hs @2.9GHz $\times$ 16
- RAM: 16Gb

Como se puede ver este ordenador es varios órdenes de magnitud más potente que los *minipcs*, por lo que es de interés medir las diferencias del *pool* de nodos frente a una máquina con mayores prestaciones. Se plantean tres escenarios para la comparación:

- 3 Nodos con 3 ejecuciones simultáneas por nodo.
- Ordenador realizando la ejecución en serie.
- Ordenador realizando la ejecución en paralelo. Para esto se usa el paquete R *parallel*.

La ejecución que realiza la máquina con mayores prestaciones se muestra en el siguiente código:

```
c<-c(1:4000000)
#serie
r1<-sapply(c,mean)
#paralelo
n.cores <- detectCores()
clust <- makeCluster(n.cores)
r2<-parSapply(clust, c, mean)
```

Source Code 7.10: Fragmentos del código de prueba de rendimiento en serie y en paralelo.

Se repite el escenario 10 veces y se muestra a continuación en la Tabla 7.4 los tiempos promedio para cada escenario:

	Pool nodos	Ordenador en serie	Ordenador en Paralelo
tiempo (s)	39.21	22.93	4.70

Tabla 7.4: Tiempos medios de ejecución en 10 repeticiones para los tres escenarios.

Como se puede ver, ante un ordenador con muchas mayores prestaciones, el sistema no es capaz de igualar la velocidad de este. Cabe remarcar que la diferencia producida está especialmente presente en el cálculo de las ejecuciones, no en el traspaso de los datos ni en el procesamiento de las ejecuciones.

7.3. Pruebas de usabilidad del sistema

Por último se ha realizado, con ayuda de varios usuarios habituales de R, unas pruebas de la usabilidad del paquete R. En estas pruebas se les ha proporcionado la documentación y pedido que usaran el sistema y crearan varias ejecuciones. Los participantes han conseguido utilizar la aplicación a partir de la documentación sin mayores problemas. Varios participantes han remarcado que les gustaría que hubiera alguna explicación más en la documentación, por lo que se han modificado algunas entradas de las funciones para que queden más claras siguiendo sus recomendaciones.

Capítulo 8

Conclusiones y discusión de los resultados

En este capítulo se desarrollan las conclusiones y se discuten los resultados obtenidos en la realización de este proyecto. También se proponen varias lineas de trabajo futuro.

8.1. Conclusiones y discusión de los resultados

Se ha conseguido desarrollar un sistema que permita realizar cálculos en un sistema distribuido. Este sistema permite de manera transparente al usuario la ejecución de sus propias lineas de código en el sistema y se han creado varias operaciones predefinidas que permiten mayor comodidad para distribuir una computación.

En primer lugar se ha creado un paquete R «`distR`» que ofrece la funcionalidad necesaria para definir las operaciones, dividir los datos y las operaciones, enviarlos al componente relevante y, posteriormente cuando haya finalizado el cálculo, reagrupar los datos y presentárselos al usuario de una manera esperada para un usuario de R.

En segundo lugar, se ha desarrollado una aplicación Elixir «`DistR`» que puede ser ejecutada en varios nodos. Estos nodos forman un `pool` de nodos que se comunican y gestionan unos a otros en un sistema en el que todo nodo es **supervisado** por el resto de los nodos. El resultado final es una configuración de supervisión de todos a todos. De esta manera, los componentes del *pool* de nodos ejecutan la misma aplicación, lo que facilita el despliegue de nuevos nodos en el sistema. Además, este sistema se vuelve resiliente a los posibles fallos que puedan existir en la comunicación entre varios componentes que están distribuidos en lugares distintos de una red de nodos. Se ha comprobado como en el evento de fallo de uno de los componentes las ejecuciones y operaciones del usuario son retomadas de manera transparente por el sistema sin perjuicio alguno en los resultados finales. Esta aplicación además ofrece dos vistas que permiten estudiar el comportamiento del sistema y mostrar la información que el usuario y administrador puedan necesitar.

Además, se ha realizado un despliegue del sistema en una red local y se ha comprobado su rendimiento. A partir de estas pruebas se ha visto como la distribución de tareas supone una mejora en cuanto al

rendimiento de un único componente. Además, se ha identificado que el coste computacional asociado a las transferencias y peticiones en el sistema supone un porcentaje menor del tiempo total de cómputo a medida que las tareas que debe realizar el sistema aumentan en complejidad.

Por último, este proyecto ha resultado exitoso en proporcionar nuevos conocimientos que pueden ser de gran utilidad en el mundo laboral y ha proporcionado una nueva perspectiva de cómo afrontar problemas en los que las entidades se pueden formular como actores.

8.2. Modificaciones de la planificación inicial

Debido a una carga inesperada de trabajo, especialmente debido a la realización de dos Trabajos Fin de Grado de manera simultánea, se ha retrasado la entrega del proyecto a la segunda convocatoria. Esto se ve reflejado en un retraso en el entregable 4 y la finalización del proyecto en 16 días lectivos más.

8.3. Lineas Futuras

Este sistema ha sido desarrollado con un propósito funcional, y por ello no está preparado para un despliegue como aplicación profesional, ya que existen vulnerabilidades que deberían ser tratadas. Por tanto, se proponen las siguientes mejoras al sistema para asemejarse más a una aplicación profesional.

- Se propone incluir en el sistema un mecanismo de inicio de sesión, diferente para el usuario y el administrador, de tal manera que cada usuario tenga acceso exclusivamente a la información de sus ejecuciones.
- Si se quiere mantener un `Dashboard` se debe ocultar información al usuario del funcionamiento interno de cada componente, ya que esta información solo debería estar disponible para el administrador.
- En el despliegue actual del sistema, el administrador debe conectarse de manera externa a cada uno de los nodos para administrarlos. Se propone diseñar varias herramientas de administración para el administrador que le permitan realizar las tareas más comunes sin tener que acceder a cada nodo de manera individual.

Además, en este sistema se dividen las tareas entre los nodos en función de las peticiones HTTP que recibe cada nodo. En el futuro se podrían implementar otros mecanismos para la división de tareas como uno que haga una estimación del coste computacional de la tarea en función del código enviado.

Bibliografía

[1] Louie Andre. 53 important statistics about how much data is created every day. https://financesonline.com/how-much-data-is-created-every-day/, May 2023.

[2] Ananth Grama. *Introduction to parallel computing*. Pearson Education, 2003.

[3] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[4] Ericsson. Erlang. https://www.erlang.org/, 2023. [Online; accessed 2-July-2023].

[5] The Elixir Team. Elixir. https://elixir-lang.org/, 2023. [Online; accessed 2-July-2023].

[6] Sean Callan. Elixir school. https://elixirschool.com/en, 2023. [Online; accessed 2-July-2023].

[7] The Elixir Team. Introduction to elixir. https://elixir-lang.org/getting-started/introduction.html, 2023. [Online; accessed 2-July-2023].

[8] Fred Hebert. *Learn you some Erlang for great good!: a beginner's guide*. No Starch Press, 2013.

[9] A Guide. Project management body of knowledge (pmbok® guide). In *Project Management Institute*, volume 11, pages 7–8, 2001.

[10] Glassdoor. Sueldos para el puesto de ingeniero de software junior en españa. https://www.glassdoor.es/Sueldos/ingeniero-de-software-junior-sueldo-SRCH_KO0,28.htm, 2023. [Online; accessed 2-July-2023].

[11] "The Editors of Encyclopaedia Britannica". Category theory. https://www.britannica.com/science/foundations-of-mathematics/Category-theory, 2023. [Online; accessed 2-July-2023].

[12] R-core. Parallel. https://www.rdocumentation.org/packages/parallel/versions/3.6.2, 2023. [Online; accessed 2-July-2023].

[13] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 31–40, 2016.

[14] Paul Schoenfelder. Libcluster. https://hexdocs.pm/libcluster/, 2023. [Online; accessed 2-July-2023].

[15] Derek Kraan. Horde. https://hexdocs.pm/horde/, 2023. [Online; accessed 2-July-2023].

[16] Chris McCord. Phoenix. https://hexdocs.pm/phoenix, 2023. [Online; accessed 2-July-2023].

[17] Lenny Delligatti. *SysML distilled: A brief guide to the systems modeling language*. Addison-Wesley, 2013.

[18] James F Kurose and Keith W Ross. Computer networking: A top-down approach edition. *Addision Wesley*, 2007.

[19] Hadley Wickham, Jim Hester, Winston Chang, Jennifer Bryan. devtools. https://www.r-project.org/nosvn/pandoc/devtools.html, 2023. [Online; accessed 2-July-2023].

[20] Hadley Wickham, Jim Hester, Winston Chang, Jennifer Bryan. Roxygen2. https://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html, 2023. [Online; accessed 2-July-2023].

[21] Cran. https://cran.r-project.org/, 2023. [Online; accessed 2-July-2023].

[22] Simon Urbanek. uuid. https://cran.r-project.org/web/packages/uuid/index.html, 2023. [Online; accessed 2-July-2023].

[23] Hadley Wickham. httr. https://cran.r-project.org/web/packages/httr/index.html, 2023. [Online; accessed 2-July-2023].

[24] Jeroen Ooms . Rcurl. https://cran.r-project.org/web/packages/uuid/index.html, 2023. [Online; accessed 2-July-2023].

[25] Daniel Stenberg. curl. https://curl.se/, 2023. [Online; accessed 2-July-2023].

[26] Tony Plate, Richard Heiberger. abind. https://cran.r-project.org/web/packages/abind/index.html, 2023. [Online; accessed 2-July-2023].

[27] Igor Sysoev. Nginx. http://nginx.org/, 2023. [Online; accessed 2-July-2023].

[28] Six Colors AB. Hex. https://hex.pm/, 2023. [Online; accessed 2-July-2023].

[29] Anthony Grimes. Mix. https://hexdocs.pm/mix/1.14/Mix.html, 2023. [Online; accessed 2-July-2023].

[30] The Elixir Team. Application. https://hexdocs.pm/elixir/1.14.5/Application.html, 2023. [Online; accessed 2-July-2023].

[31] Carlos Andres Bolaños. Nebulex. `https://hexdocs.pm/nebulex/Nebulex.html`, 2023. [Online; accessed 2-July-2023].

[32] Dirk Eddelbuettel. Rcpp. `https://cran.r-project.org/web/packages/Rcpp/index.html`, 2023. [Online; accessed 2-July-2023].

[33] Dirk Eddelbuettel. Rinside. `https://cran.r-project.org/web/packages/RInside/index.html`, 2023. [Online; accessed 2-July-2023].

Anexo

A.1. Repositorio para el código

El código de este proyecto se encuentra en `https://github.com/samuel-bassols/Comunicacion-distribuida-de-procesos`. Se ponen a disposición dos proyectos, el proyecto R del paquete distR y el proyecto de la aplicación *Elixir* DistR. Cada uno se encuentra en la carpeta llamada Elixir y R respectivamente.

Los códigos fuente del paquete R se encuentran en la carpeta `/R/distR/R` y los de la aplicación *Elixir* en `/Elixir/dist_r/lib`

En la guía que se presenta en el Anexo A.2 se muestra cómo instalar y poner en funcionamiento el sistema.

A.2. Manual de instalación

En este Anexo se describen los pasos necesarios para poner en marcha el sistema implementado en este proyecto. Esta instalación se ha probado en una máquina con el sistema operativo Ubuntu 22.04.2 LTS por lo que la instalación de algunas partes puede variar si se utiliza una distribución diferente de Linux.

A.2.1. Instalación de Elixir

En esta sección se muestra como instalar la versión 1.14.5 de *Elixir*. Para el manejo de las instalaciones se usa el *version manager* `asdf`.

1. Instalar git para poder descargar repositorios.

   `sudo apt install curl git`

2. Descargar `asdf`.

   `git clone https://github.com/asdf-vm/asdf.git ~/.asdf`.

3. Añadir el comando `asdf` al archivo .bashrc. Este archivo se encuentra en el directorio HOME y se le debe añadir la siguiente linea:

   `. $HOME/.asdf/asdf.sh`.

4. Instalar el plugin de *Erlang*.

   `asdf plugin add erlang`.

5. Instalar *Erlang* 25.3.2.3. Es posible que en el sistema falten algunas dependencias para compilar los archivos de *Erlang*. En este caso, se muestran los paquetes que es necesario instalar en la terminal.

   `asdf install erlang 25.3.2.3`.

6. Instalar el plugin de *Elixir*.

   `asdf plugin add elixir`.

7. Instalar *Elixir* 1.14.5-opt-25.

   `asdf install elixir 1.14.5-otp-25`.

8. Establecer las versiones instaladas como globales.

   `asdf global elixir 1.14.5-otp-25`.

   `asdf global erlang 25.3.2.3`.

9. Comprobar que las versiones de Elixir y Erlang coinciden. La versión de OTP debería de ser la misma cuando se ejecuta la siguiente linea.

   `elixir -v`.

A.2.2. Instalación y configuración de NGINX y el servidor sftp

En esta sección se muestra como instalar y configurar el servidor web NGINX. Para instalar NGINX se utiliza el siguiente comando: `sudo apt install nginx`. Una vez instalado se debe configurar el *LoadBalancer*. El archivo de configuración se encuentra en `etc/nginx/sites-available/conf`. En este archivo se debe copiar el texto mostrado en el fragmento de código 6.7. Para añadir nuevos nodos basta con incluir nuevos servidores en el *Loadbalancer*.
Para el servidor sftp se puede usar cualquier configuración de servidor que se desee, mientras permita conexiones desde las máquinas que se vayan a usar para el sistema.

A.2.3. Instalación de R

Para instalar R en los nodos de cómputo basta con ejecutar el siguiente comando
`sudo apt -y install r-base`. Una vez se ha instalado R hay que instalar las dependencias. Primero se debe abrir una terminal R mediante el comando `sudo R` y ejecutar las siguientes lineas:

```
install.packages("httr")
install.packages("RCurl")
install.packages("Rcpp")
install.packages("RInside")
```

A.2.4. Puesta en marcha del sistema

Una vez se han instalado todos los componentes explicados anteriormente se puede poner en funcionamiento el sistema. Para ello se debe abrir una terminal y colocarse en el directorio raíz del proyecto *Elixir* y ejecutar la siguiente linea para descargar las dependencias del proyecto.

```
mix deps.get
```

Y ya, para poner en marcha el sistema se ejecuta la siguiente linea:

```
PORT=PORT MAXP=MAXP elixir --name NAME@ADRESS
--cookie COOKIE  -S mix phx.server
```

Se deben rellenar los campos como se indica a continuación:

- PORT: Puerto por el que el nodo recibe peticiones.
- MAXP: Número máximo de ejecuciones simultáneas que se permiten en el nodo.
- NAME: Escoger un nombre del nodo.
- ADRESS: Dirección IP del nodo.
- COOKIE: Cookie que debe ser igual en todos los nodos para que se puedan comunicar.

Para indicar los miembros del *pool* de nodos, se deben añadir manualmente al archivo `./lib/dist_r/application` en la función de topología como se muestra en el fragmento de código 6.9. El nombre y dirección de los nodos debe coincidir con el que se declara en la linea de arranque de la aplicación.

Se debe además indicar la configuración correcta del servidor sftp en el archivo `./execute_command` y cuando este se haya modificado, se debe ejecutar el comando `make execute_command`. En este mismo directorio también se encuentra un archivo `r_libraries.R` donde se especifican los paquetes que son solicitados por los usuarios del sistema. Se deben previamente instalar estos paquetes antes de incluirlos.

Una vez se han establecido las distintas configuraciones, se pueden arrancar los nodos definidos en cualquier orden y momento y la conexión entre ellos es automática.

A.2.5. Paquete R DistR

Debido a que el paquete se encuentra en estado de desarrollo, este debe ser cargado como un proyecto en RStudio abriendo desde este programa el archivo `distR.Rproj`. Una vez se ha abierto el proyecto, se ejecuta las linea `devtools::load_all()` en un programa R y queda cargado el paquete DistR. Con el paquete cargado se puede consultar la ayuda de las funciones del paquete de manera idéntica a la que se utiliza con otras funciones. Se incluye además un archivo de configuración llamado `configuration.txt` en el que se puede ver un ejemplo de la configuración del entorno del paquete.