



UNIVERSIDAD NACIONAL DE ITAPUA – U.N.I.
Creada por Ley N°:1.009/96 del 03/12/96
Facultad de Ingeniería



Ingeniería en Informática

Trabajo Final de Grado

**DESARROLLO DE UN SISTEMA PARA LA EJECUCIÓN
DISTRIBUIDA DE PRUEBAS UNITARIAS DE APLICACIONES
RUBY ON RAILS UTILIZANDO LA LIBRERÍA RSPEC**

Leslie Fabiana López Figueredo

Rodrigo Jacinto Fernández Ojeda

Trabajo Final de Grado
presentado a la Facultad de
Ingeniería de la Universidad
Nacional de Itapúa, cuyo tema
fue aprobado por Res. Dec.
049/2019.

Encarnación - Paraguay
2022



UNIVERSIDAD NACIONAL DE ITAPUA – U.N.I.
Creada por Ley N°:1.009/96 del 03/12/96
Facultad de Ingeniería



Ingeniería Informática

**DESARROLLO DE UN SISTEMA PARA LA EJECUCIÓN
DISTRIBUIDA DE PRUEBAS UNITARIAS DE APLICACIONES
RUBY ON RAILS UTILIZANDO LA LIBRERÍA RSPEC**

Leslie Fabiana López Figueredo

Rodrigo Jacinto Fernández Ojeda

Tutor: Ing. Aldo Miguel Medina Venialgo

**Encarnación - Paraguay
2022**



UNIVERSIDAD NACIONAL DE ITAPUA – U.N.I.
Creada por Ley N°:1.009/96 del 03/12/96
Facultad de Ingeniería



HOJA DE EVALUACIÓN DE TFG

TÍTULO: "*Desarrollo de un sistema para la ejecución distribuida de pruebas unitarias de aplicaciones Ruby on Rails utilizando la librería RSpec.*"

TUTOR: Prof. Ing. Aldo Medina

INTEGRANTES MESA EXAMINADORA

- MSc Ing. Oscar D. Trochez V. Decano

- Prof. Ing. Arnaldo Francisco Ocampo

- Prof. Ing. Fernando Esteban Fukuchi

- Prof. Dra. María Nieves Florentín

AUTOR: LESLIE FABIANA LÓPEZ FIGUERO

CALIFICACIÓN FINAL: 5 (cinco)

ACTA N°: 03/2022.-

FECHA: 26.08.2022.-



Ing. Elsa C. González T.
Secretaría General



Ing. Oscar D. Trochez V.
Decano

Encarnación - Paraguay
2022



UNIVERSIDAD NACIONAL DE ITAPUA – U.N.I.
Creada por Ley N°:1.009/96 del 03/12/96
Facultad de Ingeniería



HOJA DE EVALUACIÓN DE TFG

TÍTULO: "Desarrollo de un sistema para la ejecución distribuida de pruebas unitarias de aplicaciones Ruby on Rails utilizando la librería RSpec."

TUTOR: Prof. Ing. Aldo Medina

INTEGRANTES MESA EXAMINADORA

- MSc Ing. Oscar D. Trochez V., Decano

- Prof. Ing. Arnaldo Francisco Ocampo

- Prof. Ing. Fernando Esteban Fukuchi

- Prof. Dra. María Nieves Florentín

AUTOR: RODRIGO JACINTO FERNÁNDEZ OJEDA

CALIFICACIÓN FINAL: 5 (cinco)

ACTA N°: 03/2022.-

FECHA: 26.08.2022.-



Lic. Elsa C. González T.
Secretaria General



Ing. Oscar D. Trochez V.
Decano

**Encarnación - Paraguay
2022**

A mi familia y a mi esposo Carlos dedico éste trabajo de corazón, por toda la ayuda y apoyo incondicional que me brindaron en toda la carrera. Sin ellos no hubiera sido posible culminar ésta etapa de mi vida.

Leslie López

Che sy Juana ha che ru Silvino pe guarã péva ko tembiapo, penderehe'ỹ naguehẽ mo'ai kuri ko'apeve. Hasype arekota mbo'ehaovusu kuation che sy.

A mi esposa Maga por estar más emocionada que yo por la culminación de este trabajo.

Rodrigo Fernández

AGRADECIMIENTOS

Primeramente a Dios, por haberme guiado y fortalecido a lo largo de éste camino.

A mis padres Ceferino y Eva por todo el apoyo incondicional que me brindaron, así también por sus mensajes de aliento para poder culminar ésta etapa de mi vida.

A mi esposo por su apoyo moral y ayuda en todo este tiempo.

A la Dra. Maria Nieves Florentin por ayudar a enfocarnos y orientarnos a la culminación del trabajo.

A nuestro tutor Ing. Aldo Medina por su orientación y tiempo que nos brindó para poder culminar el TFG.

Y a todos los que de alguna manera colaboraron en la realización de este Trabajo Final de Grado.

Leslie López

A mis padres por su apoyo incondicional durante toda la carrera, sin su soporte no hubiera podido llegar hasta aquí.

Al Consejo Directivo de la Facultad de Ingeniería de la Universidad Nacional de Itapúa por concederme la oportunidad de terminar ésta flamante carrera.

A la Dra. Maria Nieves Florentin por brindarnos su constante ayuda para culminar este Trabajo Final de Grado.

A nuestro tutor, el Ing. Aldo Medina por su seguimiento, apoyo y ayuda en todo el proceso de desarrollo de este Trabajo Final de Grado.

Rodrigo Fernández

ÍNDICE

1 INTRODUCCIÓN.....	16
1.1 PROBLEMÁTICA.....	16
1.2 OBJETIVOS.....	21
1.2.1 General.....	21
1.2.2 Específicos.....	21
1.3 ALCANCE DEL PROYECTO.....	22
1.4 JUSTIFICACIÓN.....	24
2 MARCO TEÓRICO.....	25
2.1 CICLO DE VIDA DEL DESARROLLO DEL SOFTWARE.....	25
2.1.1 Modelos de proceso de software prescriptivo o tradicionales.....	27
2.2 PRUEBAS.....	31
2.2.1 Pruebas de software.....	31
2.2.2 Métodos de testeo.....	33
2.2.3 Tipos de pruebas.....	34
2.3 DESARROLLO DIRIGIDO.....	36
2.3.1 Desarrollo dirigido por pruebas de aceptación.....	36
2.3.2 Desarrollo dirigido por comportamiento.....	36
2.3.3 Desarrollo dirigido por pruebas.....	38
2.4 PRUEBAS UNITARIAS.....	40
2.4.1 Frameworks de pruebas unitarias en Ruby.....	40
2.5 PROTOCOLOS TCP/IP.....	43
2.5.1 Protocolo de Control de Transmisión.....	43
2.5.2 Protocolo de Datagramas de Usuario.....	47
2.5.3 Protocolo secure shell.....	48
2.6 SOCKETS.....	49
2.6.1 Sockets según su orientación.....	49
2.6.2 Arquitectura cliente – servidor.....	50
2.7 DIFUSIÓN DE DATOS.....	52
2.7.1 Unidifusión.....	52
2.7.2 Difusión.....	52
2.8 LINUX.....	54
2.8.1 Distribuciones Linux.....	54
2.8.2 shell.....	56
2.8.3 Script Shell Bash.....	56
2.9 LENGUAJES DE PROGRAMACIÓN.....	58
2.10 RUBY.....	59
2.10.1 Hilos.....	59
2.10.2 Logs de ejecución de programas.....	59
2.11 GEMA.....	61
2.11.1 Estructura de una Gema.....	61
2.11.2 Creación de una Gema.....	63
2.12 GESTIÓN DE VERSIONES.....	64
2.12.1 Git.....	65
2.12.2 GitHub.....	65
2.13 SCRUM.....	66
2.13.1 Eventos de Scrum.....	66

2.13.1 Scrum Team.....	68
2.13.2 Artefactos de Scrum.....	70
3 MARCO METODOLÓGICO.....	72
3.1 HERRAMIENTAS UTILIZADAS.....	72
3.2 METODOLOGÍA DE DESARROLLO.....	72
3.3 PROCESO DE DESARROLLO.....	75
4 SOLUCIÓN IMPLEMENTADA.....	77
4.1 ARQUITECTURA DEL SISTEMA.....	77
4.2 DISEÑO DEL ALGORITMO.....	77
4.2.1 Validación del proyecto Ruby on Rails.....	78
4.2.2 Inicio de configuraciones.....	79
4.2.3 Solicitud de credenciales.....	81
4.2.4 Compresión de código fuente.....	81
4.2.5 Obtención de pruebas unitarias.....	81
4.2.6 Agrupación de pruebas unitarias.....	81
4.2.7 Establecimiento de comunicación.....	82
4.2.8 Obtención y descompresión de código fuente.....	86
4.2.9 Coordinación de la ejecución de pruebas unitarias.....	87
4.2.10 Ejecución de pruebas unitarias.....	88
4.2.11 Envío de resultados de pruebas unitarias.....	88
4.2.12 Muestra de resultados de pruebas unitarias.....	90
4.3 IMPLEMENTACIÓN DEL SISTEMA.....	91
4.3.1 Creación de la gema Liri usando Bundler.....	91
4.3.2 Instalador de la Aplicación Agente.....	97
5 RESULTADOS.....	100
5.1 USO DEL SISTEMA.....	100
5.1.1 Uso de la Aplicación Coordinadora.....	100
5.1.2 Uso de la Aplicación Agente.....	106
5.2 EVALUACIÓN DEL SISTEMA IMPLEMENTADO.....	108
5.2.1 Entornos de pruebas del sistema implementado.....	108
5.2.2 Formato de registro de resultados.....	109
5.2.3 Ejecución de las pruebas unitarias con el método convencional.....	110
5.2.4 Ejecución de las pruebas unitarias utilizando el sistema implementado.....	111
5.2.5 Comparativa de tiempos entre el método convencional y el sistema implementado.....	115
5.2.1 Validación de resultados.....	118
6 CONCLUSIÓN.....	120
7 LÍNEAS DE INVESTIGACIÓN FUTURA.....	121

LISTA DE FIGURAS

Figura 1: <i>Prototipo de la arquitectura del sistema implementado.</i>	23
Figura 2: <i>Modelo en Cascada.</i>	28
Figura 3: <i>Modelo Incremental.</i>	29
Figura 4: <i>Modelo V.</i>	30
Figura 5: <i>Desarrollo dirigido por comportamiento.</i>	38
Figura 6: <i>Ranking de frameworks de pruebas unitarias en Ruby.</i>	41
Figura 7: <i>Cabecera de un segmento TCP.</i>	44
Figura 8: <i>Cabecera de un datagrama UDP.</i>	47
Figura 9: <i>Comunicación entre sockets.</i>	49
Figura 10: <i>Ejemplo básico de Cliente-Servidor.</i>	50
Figura 11: <i>Envío de paquete, utilizando la técnica de unidifusión.</i>	52
Figura 12: <i>Envío de paquete utilizando Difusión.</i>	53
Figura 13: <i>Línea del tiempo de varios lenguajes de programación.</i>	58
Figura 14: <i>Estructura de carpetas de una gema.</i>	62
Figura 15: <i>Tablero ZenHub.</i>	74
Figura 16: <i>Esquema de componentes del sistema Liri.</i>	77
Figura 17: <i>Interacción entre la Aplicación Coordinadora y la Aplicación Agente.</i>	78
Figura 18: <i>Estructura de carpetas de la Aplicación Coordinadora.</i>	79
Figura 19: <i>Estructura de carpetas de la Aplicación Agente.</i>	80
Figura 20: <i>Esquema de proceso de la Aplicación Coordinadora.</i>	83
Figura 21: <i>Esquema de proceso de la Aplicación Agente.</i>	85
Figura 22: <i>Ejecución del comando bundle gem liri.</i>	92
Figura 23: <i>Ejecución del comando liri help.</i>	97
Figura 24: <i>Estructura de carpetas del instalador del Agente Liri.</i>	98
Figura 25: <i>Ejecución de la Aplicación Coordinadora.</i>	101
Figura 26: <i>Proceso de compresión del código fuente.</i>	101
Figura 27: <i>Proceso de comunicación con las Aplicaciones Agentes.</i>	102
Figura 28: <i>Proceso de ejecución de las pruebas unitarias.</i>	102
Figura 29: <i>Resumen de ejecución del sistema Liri.</i>	104
Figura 30: <i>Ejecución de la Aplicación Agente.</i>	106
Figura 31: <i>Comparativa de tiempos de ejecución según conjunto de archivos y entornos de pruebas.</i>	112
Figura 32: <i>Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 30 archivos.</i>	114
Figura 33: <i>Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 50 archivos.</i>	114
Figura 34: <i>Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 100 archivos.</i>	115
Figura 35: <i>Porcentaje de optimización en la ejecución de pruebas utilizando el sistema Liri.</i>	117
Figura 36: <i>Instalación de la Aplicación Agente.</i>	133
Figura 37: <i>Estado de la Aplicación Agente.</i>	133
Figura 38: <i>Gestor de software de Manjaro.</i>	137
Figura 39: <i>Configurar software de terceros en Manjaro.</i>	137
Figura 40: <i>Instalación de Google Chrome.</i>	138

Figura 41: <i>Comparativa de tiempos de ejecución del sistema Liri en cada entorno de prueba sobre la aplicación Consul</i>	148
---	-----

LISTA DE TABLAS

Tabla 1: <i>Librerías de testeo unitario existentes para ciertos lenguajes.</i>	18
Tabla 2: <i>Comparativa de herramientas que soportan la ejecución distribuida de pruebas unitarias.</i>	19
Tabla 3: <i>Niveles de prueba y sus métodos de testeo.</i>	35
Tabla 4: <i>Distribuciones Linux.</i>	55
Tabla 5: <i>Especificaciones de las computadoras utilizadas en las pruebas del sistema Liri.</i>	108
Tabla 6: <i>Entornos utilizados para las pruebas del sistema Liri.</i>	109
Tabla 7: <i>Variables consideradas en las pruebas del sistema Liri.</i>	109
Tabla 8: <i>Planilla detallada para el registro de resultados de la ejecución del sistema Liri.</i>	110
Tabla 9: <i>Planilla resumida para el registro de resultados de la ejecución del sistema Liri.</i>	110
Tabla 10: <i>Tiempo de ejecución de las pruebas por computadora sin Liri.</i>	110
Tabla 11: <i>Tiempos de ejecución según conjunto de archivos y entornos de pruebas.</i>	111
Tabla 12: <i>Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 30 archivos.</i>	113
Tabla 13: <i>Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 50 archivos.</i>	113
Tabla 14: <i>Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 100 archivos.</i>	113
Tabla 15: <i>Tiempo de ejecución de pruebas unitarias utilizando el método convencional y Liri, con sus diferentes entornos y tamaño de conjunto de archivos.</i>	115
Tabla 16: <i>Porcentaje de optimización en la ejecución de pruebas utilizando el sistema Liri.</i>	116
Tabla 17: <i>Planilla de validación del sistema Liri.</i>	119
Tabla 18: <i>Registro de resultados de pruebas de ejecución del sistema Liri en el entorno A con un tamaño de conjunto de 30 archivos.</i>	139
Tabla 19: <i>Registro de resultados de pruebas de ejecución del sistema Liri en el entorno B con un tamaño de conjunto de 30 archivos.</i>	139
Tabla 20: <i>Registro de resultados de pruebas de ejecución del sistema Liri en el entorno C con un tamaño de conjunto de 30 archivos.</i>	140
Tabla 21: <i>Registro de resultados de pruebas de ejecución del sistema Liri en el entorno D con un tamaño de conjunto de 30 archivos.</i>	140
Tabla 22: <i>Registro de resultados de pruebas de ejecución del sistema Liri en el entorno A con un tamaño de conjunto de 50 archivos.</i>	141
Tabla 23: <i>Registro de resultados de pruebas de ejecución del sistema Liri en el entorno B con un tamaño de conjunto de 50 archivos.</i>	141
Tabla 24: <i>Registro de resultados de pruebas de ejecución del sistema Liri en el entorno C con un tamaño de conjunto de 50 archivos.</i>	141
Tabla 25: <i>Registro de resultados de pruebas de ejecución del sistema Liri en el entorno D con un tamaño de conjunto de 50 archivos.</i>	142
Tabla 26: <i>Registro de resultados de pruebas de ejecución del sistema Liri en el entorno A con un tamaño de conjunto de 100 archivos.</i>	143
Tabla 27: <i>Registro de resultados de pruebas de ejecución del sistema Liri en el entorno B con un tamaño de conjunto de 100 archivos.</i>	143

Tabla 28: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno C con un tamaño de conjunto de 100 archivos.....	143
Tabla 29: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno D con un tamaño de conjunto de 100 archivos.....	144
Tabla 30: Registro de resultados de pruebas de la ejecución del sistema Liri.....	145
Tabla 31: Entornos utilizados para las pruebas del sistema Liri con la aplicación Consul.....	146
Tabla 32: Tiempo de ejecución por computadora de pruebas unitarias de la aplicación Consul sin usar Liri.....	146
Tabla 33: Registro de resultados de pruebas de la ejecución del sistema Liri sobre la aplicación Consul.....	147
Tabla 34: Tiempo de ejecución del sistema Liri en cada entorno de prueba sobre la aplicación Consul.....	147
Tabla 35: Aplicación de la fórmula para (A.1) a los entornos de prueba del Sistema Liri, excepto el entorno A.....	149

LISTA DE ABREVIATURAS

Almac.	Almacenamiento
ATDD	Desarrollo Orientado a Pruebas de Aceptación (Acceptance Test Driven Development)
BDD	Desarrollo Guiado por el Comportamiento (Behavior Driven Development)
Cant.	Cantidad
DNS	Sistema de Nombres de Dominio (Domain Name Service)
FTP	Protocolo de Transmisión de Archivos (File Transfer Protocol)
IDE	Entorno de Desarrollo Integrado (Integrated Development Environment)
IP	Protocolo de Internet (Internet Protocol)
Mem.	Memoria
OO	Orientado a Objetos (Object Oriented)
Optim.	Optimización
Proces.	Procesador
PU	Pruebas Unitarias
PC	Computadora Personal (Personal Computer)
TCP	Protocolo de Control de Transmisión (Transmission Control Protocol)
TDD	Desarrollo Guiado por Pruebas (Test Driven Development)
UDP	Protocolo de Datagramas de Usuario (User Datagram Protocol)
SCP	Protocolo de Copia Segura (Secure Copy Protocol)
SDLC	Modelo de Ciclo de Vida de Desarrollo de Software (Software Development Life Cycle Model)
SO	Sistema Operativo
SSH	Secure Shell
TFG	Trabajo Final de Grado
TI	Tecnologías de la Información (Information Technology)

RESUMEN

En la actualidad, las pruebas unitarias son una herramienta fundamental para el aseguramiento de la calidad del software. Durante el desarrollo de un proyecto de software, la cantidad y complejidad de las pruebas unitarias van aumentando, esto incide negativamente en el tiempo de ejecución del conjunto de pruebas, ya que la computadora que las ejecuta suele permanecer con los mismos recursos que al inicio del proyecto. Para paliar este problema, se implementó un sistema que se enfoca en la ejecución de pruebas unitarias de aplicaciones desarrolladas con Ruby on Rails, utilizando RSpec como librería de pruebas. Esta ejecución de pruebas se realiza a través de una arquitectura distribuida que descentraliza la carga de trabajo, reduciendo así, el tiempo requerido para la obtención de resultados.

El sistema está compuesto por una Aplicación Coordinadora y una Aplicación Agente. La Aplicación Coordinadora se encarga de coordinar el proceso de ejecución del conjunto de pruebas unitarias y del procesamiento de los resultados. La Aplicación Agente es un servicio que se encarga de ejecutar las pruebas unitarias y retornar los resultados a la Aplicación Coordinadora.

Luego de las pruebas llevadas a cabo sobre el sistema implementado, se logró disminuir el tiempo de ejecución de pruebas unitarias a partir de la utilización de dos Aplicaciones Agentes ejecutándose en diferentes computadoras.

Palabras claves: Pruebas Unitarias, RSpec, Arquitectura Distribuida, Aplicación Coordinadora, Aplicación Agente.

ABSTRACT

Nowadays, unit tests are a fundamental tool for software quality assurance. During the development of a software project, the number and complexity of unit tests increase, which has a negative impact on the execution time of the test suite, since the computer that executes them usually remains with the same resources as at the beginning of the project. To alleviate this problem, a system was implemented that focuses on the execution of unit tests of applications developed with Ruby on Rails, using RSpec as a test library. This test execution is performed through a distributed architecture that decentralizes the workload, thus reducing the time required to obtain results.

The system is composed of a Coordinator Application and an Agent Application. The Coordinator Application is responsible for coordinating the process of executing the set of unit tests and processing the results. The Agent Application is a service that executes the unit tests and returns the results to the Coordinator Application.

After the tests carried out on the implemented system, it was possible to reduce the unit test execution time by using two Agent Applications running on different computers.

Keywords: Unit Tests, RSpec, Distributed Architecture, Coordinator Application, Agent Application.

1 INTRODUCCIÓN

1.1 PROBLEMÁTICA

El software se ha convertido en un elemento ubicuo en el actual mundo digital. Esto quiere decir que está presente en todos los aspectos de la vida humana. Desde el punto de vista de la sociedad, el software provee flexibilidad, inteligencia y seguridad a todos los sistemas complejos y equipos que soportan y controlan las diferentes infraestructuras claves de nuestra sociedad como son los: transportes, comunicaciones, energía, industria, negocios, gobierno, salud, entretenimiento, etc. (Tavarez, 2014)

El proceso de desarrollo en la industria del software debe ser más dinámico y adaptable, para hacer frente a las complejidades de la actualidad. Es por eso que desde los años 60 se han propuesto y aplicado varios modelos de Ciclo de Vida de Desarrollo de Software (SDLC, del inglés Software Development Life Cycle) para lograr un mejor desarrollo y éxito económico. El SDLC representa toda la vida del proceso en función de la especificación, el diseño, la validación y la evolución del software (Chowdhury, Bhowmik, Hasan y Rahim, 2017).

Según Chowdhury, Bhowmik, Hasan y Rahim (2017), existen varios modelos de SDLC y un modelo típico de proceso de desarrollo de software tiene generalmente las siguientes etapas:

- Planificación y recopilación de información.
- Análisis y definición de requisitos.
- Diseño y definición de la arquitectura.
- Desarrollo.
- Pruebas según los requisitos y la perspectiva técnica.
- Despliegue y mantenimiento.

Berzal (2004) menciona que las etapas de desarrollo son:

Un reflejo del proceso que se sigue a la hora de resolver cualquier tipo de problema. Básicamente, resolver un problema requiere: Comprender el

problema (análisis). Plantear una posible solución, considerando soluciones alternativas (diseño). Llevar a cabo la solución planteada (implementación).

Comprobar que el resultado obtenido es correcto (pruebas).

Las etapas adicionales, como la planificación, despliegue y mantenimiento son necesarias para el desarrollo de un software porque conlleva unos costos asociados, por lo que se hace necesaria la planificación y el posterior despliegue y mantenimiento, ya que, lo que se pretende una vez construido el software, es que éste debería poder utilizarse (Berzal, 2004).

Es importante resaltar que durante el proceso de desarrollo de software, continuamente se realizan pruebas con el objetivo de verificar el correcto funcionamiento del software. Según Gómez, Jústiz y Delgado (2013) las pruebas contribuyen a la calidad del software y estas pruebas deben comenzar desde el momento en el que el desarrollador inicia la implementación del software y deben finalizar antes del despliegue del mismo.

Los principales tipos de pruebas que se pueden efectuar en cualquier tipo de software son: pruebas de integración, pruebas de regresión, pruebas de humo, pruebas del sistema y pruebas unitarias (Pauta Ayabaca y Moscoso Bernal, 2017).

En las pruebas de integración se combinan las diferentes unidades o componentes probados para formar un subsistema que funcione. A pesar de que una unidad ha superado con éxito una prueba unitaria, aún puede comportarse de manera impredecible al interactuar con otros componentes del sistema. Por lo tanto, el objetivo de las pruebas de integración es garantizar la correcta interacción e interconexión entre las unidades en un sistema de software, tal y como se define en las especificaciones de diseño detalladas (Oladimeji, 2007).

Las pruebas de regresión se llevan a cabo cada vez que se modifica el sistema, ya sea agregando nuevos componentes o corrigiendo errores. Su objetivo es determinar si la modificación del sistema ha introducido nuevos errores en el sistema (Oladimeji, 2007).

El objetivo de las pruebas de humo es determinar si la nueva versión de software es estable o no, de modo que el equipo de control de calidad pueda usarla para realizar pruebas detalladas y el equipo de desarrollo pueda seguir trabajando. Este tipo de prueba

se lleva a cabo para garantizar que las funciones más cruciales de un programa funcionen, pero sin preocuparse por los detalles más finos (Chauhan, 2014).

Las pruebas del sistema comienzan después de completar las pruebas de integración. Las mismas se realizan para demostrar que la implementación del sistema no cumple con la especificación de requisitos del sistema (Oladimeji, 2007). Estos tipos de pruebas están enfocadas directamente en los requisitos de negocio, verificando el ingreso, procesamiento, recuperación de los datos y la implementación propiamente dicha (Pauta Ayabaca y Moscoso Bernal, 2017).

La prueba unitaria enfoca los esfuerzos de verificación en la unidad más pequeña del diseño de software: el componente o módulo de software (Pressman, 2010).

Las pruebas unitarias normalmente son el primer nivel de prueba de un sistema de software y están motivadas por el hecho de que el costo de encontrar y corregir errores en el momento de la prueba de la unidad, es menor que encontrar y corregir errores que se encuentran durante los otros tipos de pruebas mencionados anteriormente o después del despliegue. Las pruebas unitarias facilitan las pruebas de regresión cuando el software cambia porque permiten a los desarrolladores verificar que no hayan dañado la funcionalidad existente (Ganesan et al., 2013).

La popularidad de las pruebas unitarias ha ido aumentando a medida que fueron surgiendo más librerías de testeo, para los distintos lenguajes de programación existentes (Athanasiou, Nugroho, Visser y Zaidman, 2014).

Lenguaje	Librerías de Testeo
Java	Junit5, TestNG
.Net	NUnit
Ruby	UnitTest, Rspec, RubyUnit, MiniTest
Phyton	Unitest, PyUnit, doctest
PHP	SimpleTest, PHPUnit
Javascript	Qunit, Mocha, Jasmine, Chai

Tabla 1: *Librerías de testeo unitario existentes para ciertos lenguajes.*

Fuente: Elaboración propia.

En la Tabla 1 se presentan algunas librerías que permiten implementar pruebas unitarias para ciertos lenguajes de programación, a estas librerías también se le denominan

frameworks. Cabe aclarar que utilizaremos las palabras prueba y test como sinónimos, así también la frase librería de testeo es lo mismo que librería de prueba.

En la Tabla 2 se presenta una comparativa de algunas herramientas de integración continua que mejoran el tiempo de ejecución de pruebas unitarias. Dichas herramientas consisten en la fusión del trabajo de los desarrolladores, permitiendo mejorar la calidad del software (Fitzgerald y Stol, 2017, citado por Shahin, Ali Babar, y Zhu, 2017). Esta práctica incluye la ejecución automatizada de las pruebas de software (Leppänen et al., 2015, citado por Shahin et al., 2017).

Herramienta	Interfaz	Lenguajes Soportados	Instalación	Licencia
Jenkins	Web	Varios	Linux, Windows, OS X, Nube	MIT, Código Abierto
TeamCity	Web	Varios	Linux, Windows, OS X, Nube	Comercial, Gratuita.
Travis CI	Web	Varios	Nube	Comercial, Gratuita.
Circle CI	Web	Varios	Linux, Windows, OS X, Nube	Comercial, Gratuita
CodeShip	Web	Varios	Nube	Comercial
Dist Test	Web	C++, Java	Linux	Apache License 2.0

Tabla 2: Comparativa de herramientas que soportan la ejecución distribuida de pruebas unitarias.
Fuente: Elaboración propia.

Como se mencionó anteriormente la integración continua ayuda mejorar el tiempo de ejecución de las pruebas unitarias, y muchas veces ésto se logra distribuyendo las mismas a través de varias computadoras interconectadas, de esta manera comparten la carga de trabajo y reducen el tiempo de ejecución. Estas herramientas dependen de un repositorio para la obtención del código fuente sobre el cual ejecutar las pruebas, por lo que no soportan la ejecución de pruebas sobre el código del desarrollador, es decir, el código sobre el cual está trabajando el desarrollador en su computadora local.

Según Wang (2016), cuando la ejecución de las pruebas unitarias toma mucho tiempo se generan los siguientes problemas:

- La productividad del desarrollador se ve afectada porque tiene que esperar la ejecución de todas las pruebas antes de continuar su trabajo.
- El desarrollador se vuelve reacio a agregar más pruebas porque incrementa el tiempo de espera para la ejecución de todas las pruebas.
- La calidad del software baja porque se dejan de escribir pruebas.

- El desarrollador pierde tiempo actualizando las pruebas tratando de disminuir su tiempo de ejecución.
- El desarrollador termina considerando a las pruebas más como una carga que una ventaja adicional para mejorar la calidad.

En 2016, Wang desarrolló un framework denominado Dist Test, orientado al desarrollador, que permite la ejecución distribuida de pruebas unitarias para software desarrollado en el lenguaje Java (Wang, 2016).

Para contrarrestar los problemas citados anteriormente, se propuso el desarrollo de un sistema similar al mencionado por Wang que dé soporte a software desarrollado en lenguaje Ruby, enfocándose específicamente en la librería de testeo RSpec para Ruby on Rails.

1.2 OBJETIVOS

1.2.1 General

Desarrollar un sistema que permita reducir el tiempo de ejecución de pruebas unitarias de aplicaciones desarrolladas con Ruby on Rails utilizando la librería RSpec.

1.2.2 Específicos

- Analizar y definir los requerimientos del sistema.
- Implementar el algoritmo de distribución de pruebas unitarias.
- Implementar el algoritmo de distribución de código fuente.
- Configurar el entorno de prueba para el sistema.
- Implementar las pruebas unitarias que se usarán para evaluar el sistema.
- Evaluar los resultados obtenidos de la ejecución del sistema.

1.3 ALCANCE DEL PROYECTO

Se implementó un sistema, que permite al desarrollador ejecutar pruebas unitarias de aplicaciones desarrolladas con Ruby on Rails utilizando Rspec como librería de pruebas unitarias. La ejecución de pruebas unitarias se realiza utilizando una arquitectura distribuida, es decir, la carga de trabajo que conlleva la ejecución de estas pruebas será distribuida entre computadoras interconectadas en una red de área local.

El sistema está compuesto por dos aplicaciones. La primera aplicación a la que denominamos Aplicación Coordinadora, se encarga de coordinar el proceso de ejecución de las pruebas, y mostrar los resultados. La segunda aplicación a la que denominamos Aplicación Agente, es un programa que se ejecuta como servicio en segundo plano y se encarga de la ejecución de las pruebas bajo las órdenes de la Aplicación Coordinadora. Para que un desarrollador pueda iniciar la ejecución de sus pruebas, debe tener instalado la Aplicación Coordinadora.

El sistema puede ser usado desde una línea de comandos. Una vez iniciado el proceso de ejecución de pruebas unitarias sobre el código del desarrollador, se muestran los resultados en la interfaz de línea de comandos, indicando las pruebas ejecutadas con éxito o que hayan fallado.

En la Figura 1 se presenta una arquitectura compuesta por cuatro computadoras, la Aplicación Coordinadora (PC1) coordina la ejecución de las pruebas, comunicándose con las Aplicaciones Agente las cuales estarán ejecutándose como un servicio dentro de todas las computadoras de la red (PC2, PC3, PC4), eventualmente, las computadoras que componen la red pueden tener instalada una o ambas aplicaciones.

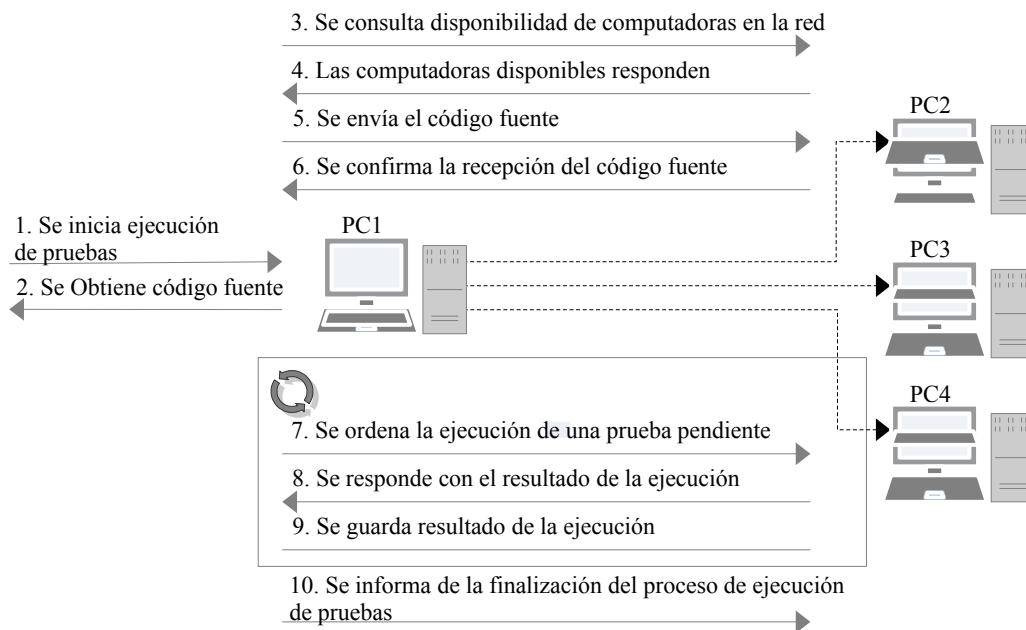


Figura 1: Prototipo de la arquitectura del sistema implementado.

Fuente: Elaboración propia.

1.4 JUSTIFICACIÓN

En grandes proyectos, la ejecución de todas las pruebas después de cada cambio puede convertirse en una operación costosa que requiere varias horas. (Blondeau et al., 2017).

Según Blondeau et al. (2017), en una importante empresa de TI, se encontraron proyectos con un entorno tan complejo y con tantas pruebas que se necesitaban horas para ejecutarlas, por este motivo, no se alentaba a los desarrolladores a ejecutar pruebas periódicas de cada modificación si esto implicaba obtener la respuesta horas después.

A pesar de que los desarrolladores deberían lanzar las pruebas localmente para evitar cometer posibles errores y propagarlos a sus colegas, tienden a delegar esta validación a la integración continua. En consecuencia, se realizan menos pruebas a nivel local y se envían los posibles errores a los demás desarrolladores del equipo (Blondeau et al., 2017).

En nuestra experiencia, a medida que se avanza en el desarrollo de un software, la cantidad y complejidad de las pruebas unitarias aumenta y la potencia de la computadora del desarrollador permanece constante, incidiendo negativamente en el tiempo de ejecución de las pruebas, haciendo que el desarrollador se vuelva reacio a ejecutar las pruebas localmente y delegando la ejecución de las mismas a la herramienta de integración continua.

Con el objetivo de reducir el tiempo de ejecución de las pruebas unitarias de código y facilitar la ejecución local de las pruebas unitarias, el sistema desarrollado en el marco de este TFG tendrá como objetivos:

- Disminuir el tiempo de ejecución de pruebas unitarias dividiendo la carga de trabajo entre varias computadoras interconectadas, de este modo el tiempo que toma la ejecución de las pruebas unitarias de código no estará limitado por la potencia de la computadora del desarrollador.
- Permitir la ejecución de pruebas unitarias sobre el código fuente del desarrollador utilizando la arquitectura distribuida mencionada en el punto anterior.
- La cantidad de pruebas unitarias a ejecutar en cada computadora involucrada será configurable de manera manual.

2 MARCO TEÓRICO

2.1 CICLO DE VIDA DEL DESARROLLO DEL SOFTWARE

Un ciclo de vida abarca todas las etapas del software, desde su inicio, con la definición de requisitos, hasta su puesta en marcha y mantenimiento (Ruparelia, 2010).

El Ciclo de Vida de Desarrollo de Software (SDLC, del inglés Software Development Life Cycle), es un marco conceptual o un proceso que considera la estructura de las etapas que intervienen en el desarrollo de una aplicación, desde su estudio inicial de viabilidad hasta su despliegue y su mantenimiento. Existen varios modelos de SDLC que describen diversos enfoques del proceso SDLC. Un modelo SDLC se utiliza generalmente para describir los pasos que se siguen dentro del marco del ciclo de vida (Ruparelia, 2010).

Según Berzal (2004), un ciclo de vida comprende una serie de etapas entre las que se encuentran las siguientes:

- Planificación
- Análisis
- Diseño
- Implementación
- Pruebas
- Instalación o despliegue
- Uso y mantenimiento

Antes de arrancar con un proyecto de desarrollo de software es necesario realizar una serie de tareas previas que influirán decisivamente para la finalización exitosa del proyecto, esta serie de tareas se realiza durante la etapa de **planificación** (Berzal, 2004).

La etapa de **análisis** en el ciclo de vida del software corresponde al proceso mediante el cual se intenta descubrir qué es lo que realmente se necesita y se llega a una comprensión adecuada de los requerimientos del sistema (las características que el software debe poseer). (Berzal, 2004, p. 9)

Mientras que en la etapa de análisis, los requisitos del usuario se representan desde distintos puntos de vista (el qué), en la etapa de **diseño** se representan las características del sistema que nos permitirán implementarlo de forma efectiva (el cómo). (Berzal, 2004)

En la etapa de diseño se estudian las posibles alternativas de implementación del software que se desarrollará y se decide la estructura general que tendrá el software (su diseño arquitectónico) (Berzal, 2004).

En la etapa de **implementación** se comienza a codificar, pero antes de comenzar a escribir alguna línea de código, es fundamental haber comprendido bien el problema que se quiere resolver y haber aplicado principios básicos de diseño que permitan construir un software de calidad. Para la etapa de **implementación** se seleccionan las herramientas adecuadas, un entorno de desarrollo que facilite el trabajo y un lenguaje de programación apropiado para el tipo de software a desarrollar (Berzal, 2004).

La etapa de **pruebas** tiene como objetivo detectar los errores que se pudieron haber cometido en etapas anteriores del proyecto, y eventualmente corregirlos. Tiene como fin descubrir los posibles errores antes de que el usuario final del sistema los sufra. De hecho, una prueba es exitosa cuando se detecta un error y no al revés (Berzal, 2004).

Luego de finalizar las etapas anteriores se procede a la **instalación o despliegue** del sistema. Según Berzal (2004), en esta etapa se planifica:

El entorno en el que el sistema debe funcionar, tanto hardware como software: equipos necesarios y su configuración física, redes de interconexión entre los equipos y de acceso a sistemas externos, sistemas operativos, librerías y componentes suministrados por terceras partes, etc. (p.21)

Dada la naturaleza del software, que no se rompe, ni se desgasta con el uso, su **mantenimiento** incluye tres facetas diferentes:

- Eliminar los defectos que se detecten durante su vida útil (mantenimiento correctivo).
- Adaptarlo a nuevas necesidades (mantenimiento adaptativo), cuando el sistema ha de funcionar sobre una nueva versión del sistema operativo o en un entorno hardware diferente, por ejemplo.
- Añadirle nueva funcionalidad (mantenimiento perfectivo), cuando se proponen características deseables que supondrían una mejora del sistema ya existente. (Berzal, 2004, p. 21)

2.1.1 Modelos de proceso de software prescriptivo o tradicionales

Los modelos de proceso prescriptivo fueron propuestos originalmente para poner orden en el caos del desarrollo de software. La historia indica que estos modelos tradicionales han dado cierta estructura útil al trabajo de ingeniería de software y que constituyen un mapa razonablemente eficaz para los equipos de software. (Pressman, 2010, p. 33)

En estos modelos se prescriben un conjunto de elementos del proceso: actividades estructurales, acciones, tareas, productos del trabajo, aseguramiento de la calidad y mecanismos de control del cambio para cada proyecto. Cada modelo también prescribe un flujo del proceso (de trabajo), es decir, la manera en la que los elementos del proceso se relacionan entre sí (Pressman, 2010).

El proceso que se adapta mejor al proyecto en el que se está trabajando depende del software que se está elaborando. Un proceso puede ser apropiado para crear software destinado a un sistema de control electrónico de un aeroplano, mientras que para la creación de un sitio web, el proceso seleccionado podría ser distinto (Pressman, 2010).

A continuación se seleccionaron algunos modelos para ver cómo se componen y funcionan sus diferentes flujos de trabajo, con esto podemos identificar que las pruebas son parte de prácticamente todos los modelos existentes. Tener en cuenta que, la cantidad y forma de presentación de las etapas para cada modelo puede variar, dependiendo de la bibliografía consultada.

Modelo en Cascada

El modelo en cascada, sigue un enfoque sistemático y secuencial para el desarrollo del software, (Pressman, 2010). Como se puede observar en la Figura 2, el proceso de desarrollo posee la siguiente secuencia de fases: análisis, diseño, código y prueba.

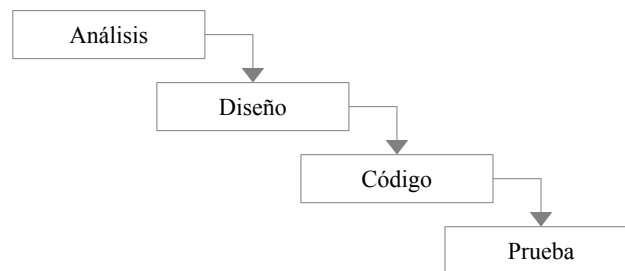


Figura 2: *Modelo en Cascada.*

Fuente: Modelo lineal secuencial (p. 20), por Roger S. Pressman, 2002, McGraw-Hill Interamericana.

Cataldi, Lage, Pessacq y García Martínez (1999) menciona las siguientes características del modelo en cascada:

- Cada fase empieza cuando se ha terminado la anterior.
- Para pasar a la fase siguiente es necesario haber logrado los objetivos de la fase previa.
- Es útil para un control de fechas de entregas.
- Al final de cada fase las personas involucradas (como los usuarios y encargados técnicos) tienen la oportunidad de revisar el progreso del proyecto. (p. 187)

Modelo Incremental

Según Pressman (2002), el modelo incremental combina elementos del modelo en cascada (aplicados repetidamente) en conjunto con la construcción de prototipos. Como se puede observar en la Figura 3, en cada incremento se pasa por las mismas etapas indicadas en el modelo en cascada.

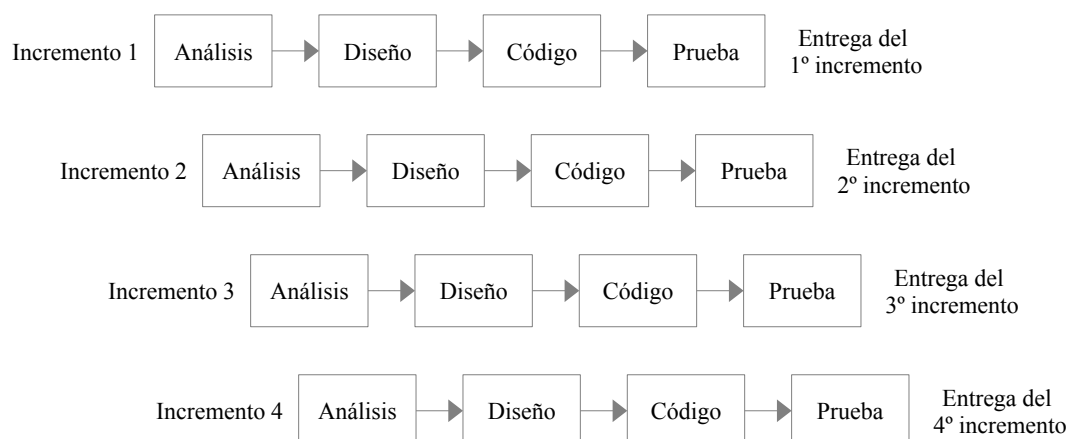


Figura 3: *Modelo Incremental.*

Fuente: Modelo incremental (p. 24), por Roger S. Pressman, 2002, McGraw-Hill Interamericana.

En el modelo incremental, el primer incremento suele ser un producto esencial, por lo que se cubren solo los requisitos básicos. Luego de una evaluación por parte del cliente, se desarrolla un plan para el siguiente incremento. En cada incremento se trata de cubrir mejor las necesidades del cliente agregando características adicionales. Este proceso se repite en cada incremento hasta lograr un producto terminado (Pressman, 2002).

El modelo incremental se enfoca en entregar en cada incremento versiones incompletas, pero funcionales del producto final, de este modo el cliente puede ir evaluando el funcionamiento del producto (Pressman, 2002).

Modelo V

El modelo V es una variante del modelo en cascada. Como se puede observar en la Figura 4, a medida que el desarrollo avanza hacia abajo desde el lado izquierdo de la V, se van generando representaciones más detalladas y técnicas del problema a resolver y su solución. Una vez generado el código, se procede a ejecutar pruebas subiendo por el lado derecho de la V, para validar cada modelo creado cuando se bajó por el lado izquierdo de la V (Pressman, 2010).

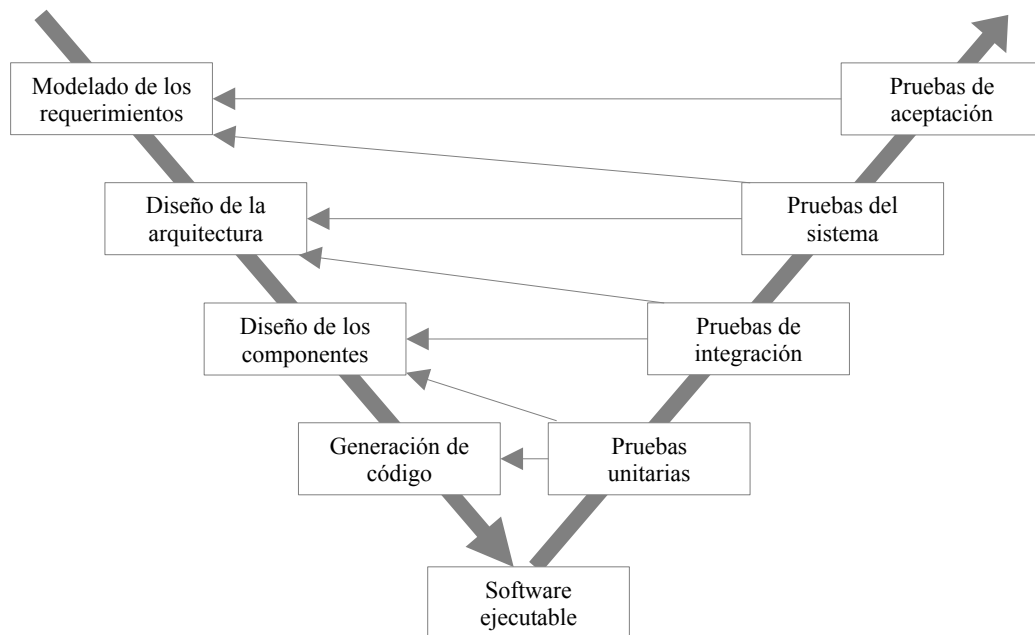


Figura 4: *Modelo V.*

Fuente: Modelo en V (p. 35), por Roger S. Pressman, 2010, McGraw-Hill Interamericana.

2.2 PRUEBAS

Las pruebas forman parte de una de las fases del ciclo de desarrollo de un software, y es la que estaremos profundizando, ya que estaremos tratando sobre el mismo en el desarrollo del TFG.

Bertolino (2001) afirma que las pruebas son una parte importante y obligatoria del desarrollo de software; es una técnica para evaluar la calidad del producto y también para mejorarla indirectamente, identificando defectos y problemas.

Las pruebas son una actividad esencial en la ingeniería de software y, en términos más sencillos, consisten en observar la ejecución de un sistema de software para validar si se comporta según lo previsto e identificar posibles fallos de funcionamiento. Las pruebas se utilizan ampliamente en la industria para garantizar la calidad: en efecto, al examinar directamente el software en ejecución, proporciona una información realista de su comportamiento y, como tal, sigue siendo el complemento ineludible de otras técnicas de análisis (Bertolino, 2007).

2.2.1 Pruebas de software

La prueba es el proceso de ejecutar un programa con la intención de encontrar errores (Myers, Badgett, Thomas y Sandler, 2004).

Bertolino (2001) menciona que las pruebas de software consisten en la verificación dinámica del comportamiento de un programa en un conjunto finito de casos de prueba, adecuadamente seleccionados del dominio de ejecuciones generalmente infinito, contra el comportamiento esperado especificado.

Según ISTQB (2018), a lo largo de los últimos 50 años se han sugerido una serie de principios de pruebas que ofrecen unas directrices generales comunes a todas las pruebas:

- **Principio 1. Las pruebas demuestran la presencia de defectos, no su ausencia:**
Las pruebas reducen la probabilidad de que queden defectos sin descubrir en el software, pero, aunque no se encuentren defectos, no significa que no los haya.
- **Principio 2. Las pruebas exhaustivas son imposibles:** Probarlo todo (todas las combinaciones de entradas y precondiciones) no es factible, salvo en casos triviales.

En lugar de intentar hacer pruebas exhaustivas, hay que utilizar el análisis de riesgos, las técnicas de prueba y las prioridades para centrar los esfuerzos de prueba.

- **Principio 3. Las pruebas tempranas ahorran tiempo y dinero:** Para encontrar los defectos con antelación, las actividades de prueba tanto estáticas como dinámicas deben iniciarse lo antes posible en el ciclo de vida del desarrollo de software. La realización de pruebas en una fase temprana del ciclo de vida de desarrollo del software ayuda a reducir o eliminar cambios costosos.
- **Principio 4. Los defectos se agrupan:** Un pequeño número de módulos suele contener la mayor parte de los defectos descubiertos durante las pruebas previas al lanzamiento, o es responsable de la mayoría de los fallos operativos. Las agrupaciones de defectos previstas y las observadas realmente en las pruebas o en el funcionamiento son una aportación importante al análisis de riesgos utilizado para centrar el esfuerzo de las pruebas.
- **Principio 5. Cuidado con la paradoja de los pesticidas:** Si se repiten las mismas pruebas una y otra vez, con el tiempo estas pruebas dejan de encontrar nuevos defectos. Para detectar nuevos defectos, es posible que haya que cambiar las pruebas y los datos de prueba existentes, y que haya que escribir nuevas pruebas. (Las pruebas dejan de ser eficaces para encontrar defectos, igual que los pesticidas dejan de ser eficaces para matar insectos después de un tiempo). En algunos casos, como las pruebas de regresión automatizadas, la paradoja del pesticida tiene un resultado beneficioso, que es el número relativamente bajo de defectos de regresión.
- **Principio 6. Las pruebas dependen del contexto:** Las pruebas se realizan de forma diferente en distintos contextos. Por ejemplo, el software de control industrial de seguridad crítica se prueba de forma diferente a una aplicación móvil de comercio electrónico. Otro ejemplo: las pruebas en un proyecto ágil se realizan de forma diferente a las pruebas en un proyecto de ciclo de vida de desarrollo de software secuencial.
- **Principio 7. La ausencia de errores es una falacia:** Algunas organizaciones esperan que los testers puedan realizar todas las pruebas posibles y encontrar todos

los defectos posibles, pero los principios 1 y 2, respectivamente, nos dicen que esto es imposible. Además, es una falacia (es decir, una creencia errónea) esperar que el mero hecho de encontrar y corregir un gran número de defectos garantice el éxito de un sistema. Por ejemplo, si se comprueban a fondo todos los requisitos especificados y se corrigen todos los defectos encontrados, el resultado puede ser un sistema difícil de usar, que no satisface las necesidades y expectativas de los usuarios o que es inferior en comparación con otros sistemas.

2.2.2 Métodos de testeo

- **Caja Negra:** Es la técnica de prueba sin tener ningún conocimiento del funcionamiento interno de la aplicación. El tester ignora la arquitectura del sistema y no tiene acceso al código fuente. Por lo general, al realizar una prueba de caja negra, el tester interactuará con la interfaz de usuario del sistema al proporcionar entradas y examinar las salidas sin saber cómo y dónde se trabaja con las entradas (tutorialspoint.com, 2011).
- **Caja Gris:** La prueba de caja gris es una técnica para probar la aplicación con un conocimiento limitado del funcionamiento interno de una aplicación. Dominar el dominio de un sistema siempre le da al tester una ventaja sobre alguien con conocimiento limitado del dominio. A diferencia de las pruebas de caja negra, en las pruebas de caja gris, el tester tiene acceso a los documentos de diseño y la base de datos. Con este conocimiento, el mismo puede preparar mejor los datos de prueba y los escenarios de prueba al hacer el plan de prueba (tutorialspoint.com, 2011).
- **Caja Blanca:** Es la investigación detallada de la lógica interna y la estructura del código. La prueba de caja blanca también se llama prueba de vidrio o prueba de caja abierta. Para realizar pruebas de caja blanca en una aplicación, el tester debe tener conocimiento del funcionamiento interno del código, debido a que el mismo debe verificar el código fuente y descubrir qué unidad/fragmento del código se está comportando de manera inapropiada (tutorialspoint.com, 2011).

2.2.3 Tipos de pruebas

Pauta Ayabaca y Moscoso Bernal (2017) mencionan los siguientes tipos de pruebas de software:

- **Pruebas unitarias:** Comprueba si el código funciona y cumple con las especificaciones necesarias para su desempeño óptimo, se focaliza en verificar cada módulo con lo que mejora el manejo de la integración de lo más básico a los componentes mayores.
- **Pruebas de integración:** Comprueba si los componentes funcionan correctamente luego de integrarlos, identificando los errores producidos por la combinación, definiendo si las interfaces entre los usuarios y las aplicaciones funcionan de una manera adecuada.
- **Pruebas de regresión:** Comprueba si los cambios efectuados en una parte del programa afectan a otras partes de la aplicación.
- **Pruebas de humo:** Comprueba los errores tempranamente revisando el sistema constantemente, lo cual permite disminuir la dificultad en el momento de la integración reduciendo así los riesgos.
- **Pruebas del sistema:** Están enfocadas directamente a los requisitos tomados de los casos de uso según el funcionamiento del negocio, verificando el ingreso, procesamiento, recuperación de los datos y la implementación propiamente dicha. El principal objetivo es que la aplicación cubra las necesidades del negocio, entre las cuales tenemos: prueba de funcionalidad, usabilidad, performance, documentación, procedimientos, seguridad, controles, etc. (pp. 30-31)

En la Tabla 3 se puede observar los niveles de pruebas y sus respectivos métodos de testeo.

Nivel de Prueba	Método de Testeo
Pruebas unitarias	Caja blanca
Pruebas de integración	Caja blanca y negra
Pruebas de regresión	Caja blanca y negra
Pruebas de humo	Caja negra
Pruebas del sistema	Caja negra

Tabla 3: *Niveles de prueba y sus métodos de testeo.*

Fuente: Guía de Testing de Software: Principales tipos de Pruebas de Software (p.1), por A. C. Rocha, 2010.

2.3 DESARROLLO DIRIGIDO

2.3.1 Desarrollo dirigido por pruebas de aceptación

El desarrollo dirigido por pruebas de aceptación (ATDD, del inglés Acceptance Test Driven Development), es una práctica en la que todo el equipo analiza en colaboración los criterios de aceptación, con ejemplos y luego los proyecta en un conjunto de pruebas de aceptación concretas, antes de que comience el desarrollo. Es la mejor manera para asegurar que todos tengan la misma comprensión compartida de lo que realmente están construyendo (Hendricksons, 2008).

Al realizar un desarrollo dirigido por pruebas de aceptación, como parte de una práctica ágil, las pruebas se crean de manera iterativa, comenzando antes y continuando durante la implementación de una historia de usuario.

Las historias de usuario deben incluir criterios de aceptación y, a su vez, estos pueden convertirse en (borradores) pruebas de aceptación. Todo esto sucede a través de representantes comerciales, desarrolladores y testers que trabajan juntos en una reunión de especificación. En la reunión de especificación no se trata solo de crear pruebas de aceptación. El objetivo real es comprender en colaboración lo que el software debe y no debe hacer (Black et al., 2017).

2.3.2 Desarrollo dirigido por comportamiento

El desarrollo dirigido por comportamiento (BDD, del inglés Behaviour-Driven Development) comienza desde el punto de vista de que los comportamientos del software son más fáciles de entender para las partes interesadas cuando participan en la creación de pruebas. La idea es, junto con todos los miembros del equipo, representantes comerciales e incluso clientes, definir los comportamientos del software. Esta definición puede ocurrir en una reunión de especificación, luego, las pruebas se basan en los comportamientos esperados y el desarrollador ejecuta estas pruebas todo el tiempo mientras desarrolla el código (Black et al., 2017)

BDD se basa en el uso de un vocabulario muy específico (y reducido) para minimizar la falta de comunicación y garantizar que todos estén en la misma página y que también

utilicen las mismas palabras, minimizando así los obstáculos entre ellos y permitiendo la entrega incremental del software (Duarte y Amílcar, 2011).

El núcleo de BDD es "conseguir las palabras adecuadas", es decir, producir un vocabulario que sea preciso, accesible, descriptivo y coherente. Concentrarse en encontrar las palabras adecuadas nos lleva (a los programadores) a comprender mejor la estrecha relación que existe entre las historias que utilizamos para especificar el comportamiento y las especificaciones que implementamos (Duarte y Amílcar, 2011).

BDD amplía la idea de TDD con historias de usuario escritas en lenguaje natural o pruebas de aceptación, que se denominan escenarios. Estos se agrupan por medio de características y cada escenario se describe como una secuencia de oraciones. Para tener un texto agradablemente legible, el flujo BDD sugiere usar las palabras clave Dado, Cuándo y Entonces, que se refieren al código de prueba que contiene suposiciones, condiciones y afirmaciones, respectivamente (Diepenbeck y Drechsler, 2015).

North (2006), menciona que un escenario dado/cuando/entonces debe describir lo siguiente:

- **Dado (Given):** un contexto inicial (los hechos),
- **Cuando (When):** ocurre un evento,
- **Entonces (Then):** asegura algunos resultados.

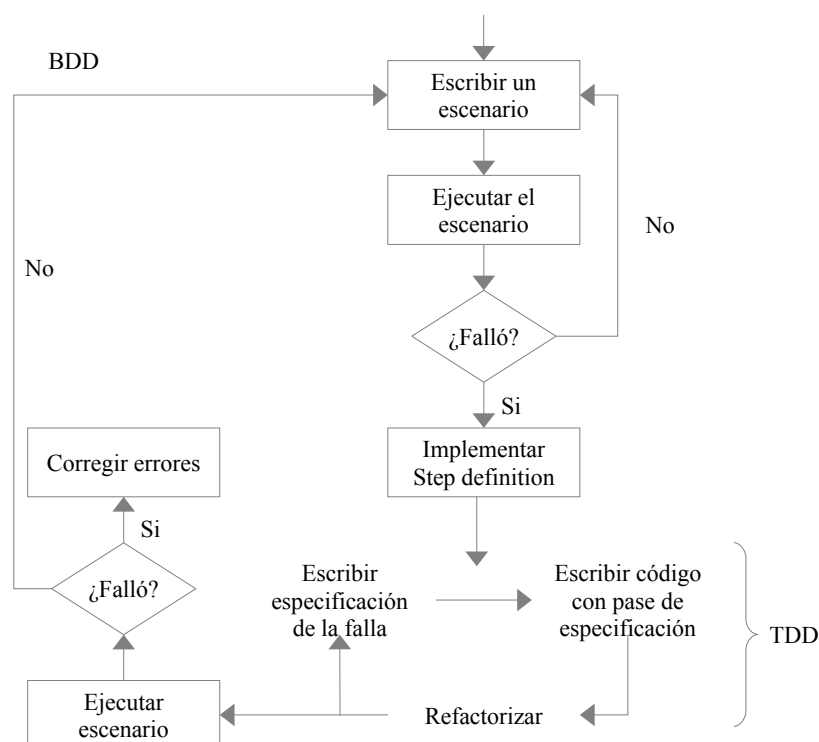


Figura 5: *Desarrollo dirigido por comportamiento.*

Fuente: Estructura del Desarrollo Guiado por Comportamiento (p. 196), por Aldo Emanuel Soralus Soralus, Miguel Ángel Valles Coral y Danny Lévano Rodríguez, 2021, Ingeniería y Desarrollo.

Como se puede observar en la Figura 5, BDD se basa en un análisis a través de escenarios dada la característica o funcionalidad del sistema, que cuenta con tres componentes principales a definir: dado, cuando y entonces, que prácticamente es la especificación en detalle de una acción dada en un momento específico y el resultado que se espera, el cual representa cierta funcionalidad del sistema.

2.3.3 Desarrollo dirigido por pruebas

El desarrollo dirigido por pruebas (TDD, del inglés Test-Driven Development), es un método mediante el cual se crean pruebas unitarias, en pequeños pasos incrementales, y en los mismos, se crea el código para cumplir con esas pruebas (Black et al., 2017).

Estas pruebas unitarias permiten a los desarrolladores verificar si su código se comporta de acuerdo con su diseño, tanto mientras desarrollan la unidad, como después de realizar cualquier cambio en la unidad. Esto proporciona un nivel de confianza que lleva a muchos

desarrolladores a quedarse con TDD una vez que se han acostumbrado al proceso (Black et al., 2017).

TDD implica primero escribir una prueba de la funcionalidad de bajo nivel esperada y luego escribir el código que ejecutará esa prueba. Cuando pasa la prueba, es hora de pasar a la siguiente pieza de funcionalidad de bajo nivel. A medida que aumenta la cantidad de pruebas y la base de código de forma incremental, también se necesita refactorizar el código con frecuencia. La fortaleza de TDD es que se desarrolla el código mínimo que se requiere para pasar las pruebas unitarias existentes, antes de pasar a la siguiente prueba y código (Black et al., 2017).

2.4 PRUEBAS UNITARIAS

Una prueba unitaria procura una “unidad” de código de forma aislada y compara los resultados reales con los esperados. Las pruebas unitarias ejecutan uno o más fragmentos de código para producir resultados observables que se verifican automáticamente (Olan, 2003).

Según ISTQB (2018), las pruebas unitarias suele ser realizadas por los desarrolladores que han escrito el código.

Osherove (2014) menciona que las pruebas unitarias deben tener las siguientes propiedades:

- Automatizado y repetible.
- Fácil de implementar.
- Pertinente.
- Cualquiera debería poder ejecutarlo.
- Ejecutarse rápidamente.
- Coherente en sus resultados (siempre devuelve el mismo resultado si no cambia nada entre ejecuciones).
- Tener el control total de la unidad bajo prueba.
- Estar completamente aislado (se ejecuta independientemente de otras pruebas).
- Cuando falla, debería ser fácil detectar lo que se esperaba y determinar cómo identificar el problema.

2.4.1 Frameworks de pruebas unitarias en Ruby

En Ruby tenemos una amplia gama de frameworks para pruebas unitarias. En la Figura 6 podemos ver un ranking de las librerías más utilizadas actualmente.

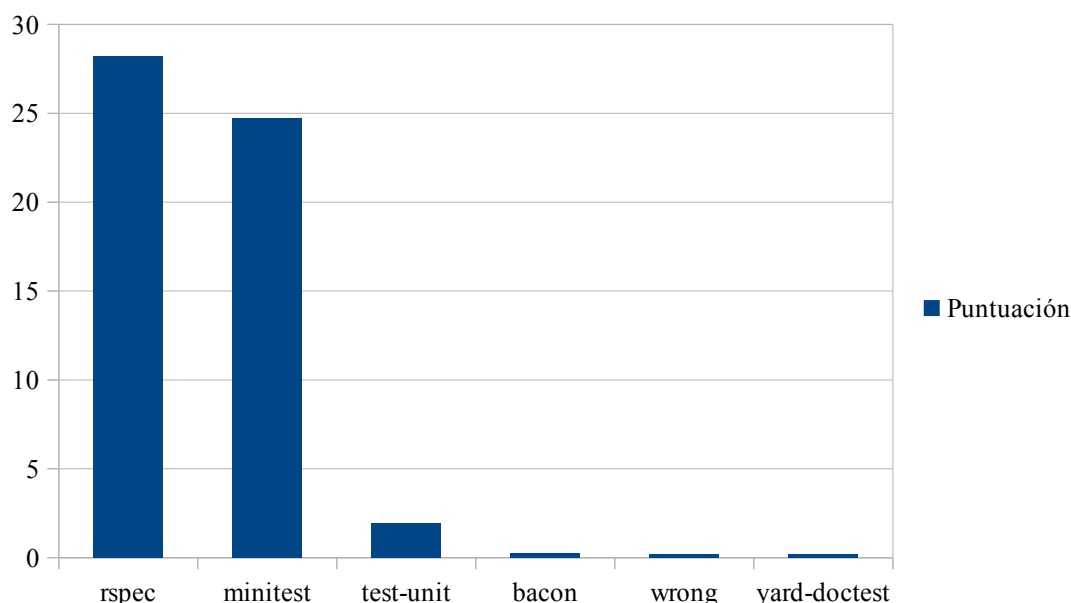


Figura 6: *Ranking de frameworks de pruebas unitarias en Ruby.*

Fuente: Datos expresados de acuerdo a la puntuación del proyecto. Test frameworks, 2022.
https://www.ruby-toolbox.com/categories/testing_frameworks?display=table&order=score

Como se puede observar en la Figura 6, en los 2 primeros puestos del ranking se encuentran RSpec y Minitest respectivamente. Tanto RSpec, como Minitest, pueden ser usados para implementar pruebas unitarias de aplicaciones escritas con Ruby on Rails.

RSpec

Es una herramienta de desarrollo dirigida por el comportamiento para programadores de Ruby («RSpec», 2022).

Hourquebie (2018) menciona que:

RSpec es, a juzgar por las gemas y proyectos que hemos visto, el framework preferido por los desarrolladores Ruby. Es una herramienta muy robusta y preparada para testear cualquier solución que le haga frente.

La estructura de los tests escritos con RSpec es más flexible que la de Minitest, permite utilizar diferentes descripciones y contextos para clarificar las incumbencias de aquello que se está probando.

MiniTest

Es una herramienta de prueba para Ruby que proporciona un conjunto completo de facilidades de prueba que soportan TDD, BDD y otros (Davis Ryan, 2022).

Según Hourquebie (2018):

Minitest es muy simple de utilizar, tiene una sintaxis sencilla y es de rápida implementación con tests de baja a mediana complejidad. Su utilización consiste básicamente en definir clases que se encarguen de testear a otra clase Ruby, o bien a una integración de ellas, dependiendo del tipo de test que se quiere realizar. Cada clase se compone básicamente de tres partes:

- **Test:** Operaciones básicas que se ejecutan para verificar algún requerimiento en particular. Su definición requiere de un string que define aquello que se está probando.
- **Setup:** Bloque de código que se ejecuta **antes de correr cada test**, una vez por cada uno de ellos, para configurar un entorno común.
- **Teardown:** Bloque de código que se ejecuta **luego de correr cada test**, una vez por cada uno de ellos. Se utiliza para realizar una limpieza post-test.

2.5 PROTOCOLOS TCP/IP

TCP/IP es un conjunto robusto de protocolos diseñados para proporcionar comunicaciones entre sistemas con diferentes sistemas operativos y hardware (Petersen, 2008).

El conjunto de protocolos TCP/IP consta en realidad de diferentes protocolos, cada uno de ellos diseñado para una tarea específica en una red TCP/IP. Los tres protocolos básicos son el Protocolo de Control de Transmisión (TCP, del inglés Transmission Control Protocol), que se encarga de recibir y enviar paquetes; el Protocolo de Internet (IP, del inglés Internet Protocol), es la base que utilizan todos los demás protocolos y se encarga de las transmisiones reales; y el Protocolo de Datagramas de Usuario (UDP, del inglés User Datagram Protocol), que también se encarga de recibir y enviar paquetes (Petersen, 2008).

2.5.1 Protocolo de Control de Transmisión

El protocolo de control de transmisión está pensado para ser utilizado como un protocolo host a host muy fiable entre miembros de redes de comunicación de computadoras por intercambio de paquetes y en un sistema interconectado de tales redes (IETF, 1981).

Según (Verdejo Alvarez, 2003):

TCP utiliza los servicios de IP para su transporte por Internet, es un protocolo orientado a conexión. Esto significa que las dos aplicaciones envueltas en la comunicación (usualmente un cliente y un servidor), deben establecer previamente una comunicación antes de poder intercambiar datos. (p. 21)

“TCP conecta un encabezado a los datos transmitidos. Este encabezado contiene múltiples parámetros que ayudan a los procesos del sistema transmisor a conectarse a sus procesos correspondientes en el sistema receptor” (ORACLE, 2010). TCP asegura que cualquier información enviada a través de la red sea recibida por el otro sistema con el cual se estableció la comunicación, lo que lo hace muy confiable en su entrega (Dominguez, 2002).

En la Figura 7 se representa los datos enviados en una cabecera TCP.

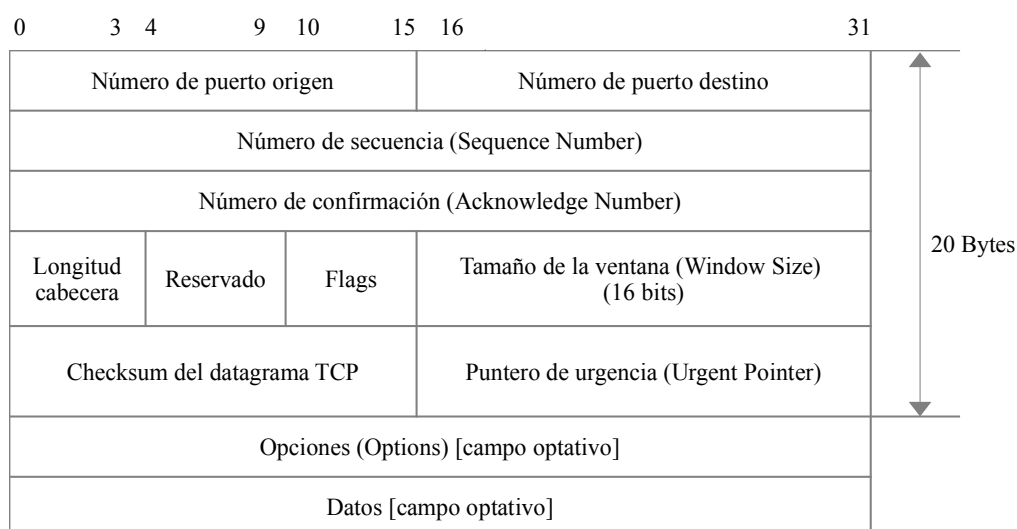


Figura 7: Cabecera de un segmento TCP.

Fuente: Cabecera de un segmento TCP (p. 23), por Gabriel Verdejo Alvarez, 2003, Universitat Autònoma de Barcelona.

Verdejo Alvarez (2003) explica de la siguiente manera cada campo de la cabecera de un segmento TCP:

El **número de puerto origen y número de puerto destino**, sirven para diferenciar una comunicación en un ordenador de las demás. La tupla formada por la dirección IP y el número de puerto se denomina socket.

El **número de secuencia (Sequence Number)**, identifica el byte concreto del flujo de datos que actualmente se envía del emisor al receptor. De esta forma, TCP numera los bytes de la comunicación de una forma consecutiva a partir del número de secuencia inicial. Cuando se establece una comunicación, emisor y receptor eligen un número de secuencia común, lo que permite implementar mecanismos de control como asegurar que los datos lleguen en el orden adecuado. Es un número de 32 bits, con lo que podemos enviar $2^{32}-1$ bytes antes de que reinicie el ciclo.

El **número de confirmación (Acknowledge Number)**, es el número de secuencia más uno. De este modo se especifica al emisor que los datos enviados hasta este número de secuencia menos uno son correctos.

La **longitud de la cabecera (header Length)**, especifica en palabras de 32 bits (4 bytes) el tamaño de la cabecera del segmento TCP incluyendo las posibles opciones.

Las **banderas (Flags)**, son las encargadas de especificar los diferentes estados de la comunicación. Así mismo, también validan los valores de los distintos campos de la cabecera de control. Puede haber simultáneamente varios flags activados.

El **tamaño de la ventana (Window Size)**, es el número de bytes desde el número especificado en el campo de confirmación, que el receptor está dispuesto a aceptar. El tamaño máximo es de (2^{16}) 65535 bytes. De esta forma, el protocolo TCP permite la regulación del flujo de datos entre el emisor y el receptor.

El **checksum** del segmento TCP, al igual que el del UDP o IP, tiene la función de controlar los posibles errores que se produzcan en la transmisión. Este checksum engloba la cabecera TCP y los datos. En caso de error, el segmento queda descartado y el propio protocolo es el encargado de asegurar la retransmisión de los segmentos erróneos y/o perdidos.

El **puntero de urgencia (Urgent Pointer)**, es válido solo si el flag de URG se encuentra activado. Consiste en un valor positivo que se debe sumar al número de secuencia especificando una posición adelantada dónde podemos enviar datos urgentes.

Las **opciones (Options)**, nos permiten especificar de forma opcional características extras a la comunicación.

Los **datos (Data)** son opcionales. Esto significa que podemos enviar simplemente cabeceras TCP con diferentes opciones. Esta característica se utiliza por ejemplo al iniciar la comunicación o en el envío de confirmaciones.

(pp. 23-25)

Según Verdejo Alvarez (2003):

TCP es también un protocolo fiable. La fiabilidad proporcionada por este protocolo viene dada principalmente por los siguientes aspectos:

- Los datos a enviar son reagrupados por el protocolo en porciones denominadas segmentos. El tamaño de estos segmentos lo asigna el propio protocolo.
- Cuando en una conexión TCP se recibe un segmento completo, el receptor envía una respuesta de confirmación (Acknowledge) al emisor confirmando el número de bytes correctos recibidos. De esta forma, el emisor da por correctos los bytes enviados y puede seguir enviando nuevos bytes.
- Cuando se envía un segmento se inicializa un temporizador (timer). De esta forma, si en un determinado plazo de tiempo no se recibe una confirmación (Acknowledge) de los datos enviados, estos se retransmiten.
- TCP incorpora un checksum para comprobar la validez de los datos recibidos. Si se recibe un segmento erróneo (fallo de checksum por ejemplo), no se envía una confirmación. De esta forma, el emisor retransmite los datos (bytes) otra vez.
- Como IP no garantiza el orden de llegada de los paquetes, el protocolo TCP utiliza unos números de secuencia para asegurar la recepción en orden, evitando cambios de orden y/o duplicidades de los bytes recibidos.
- TCP implementa un control de flujo de datos. De esta forma, en el envío de datos se puede ajustar la cantidad de datos enviada en cada segmento, evitando colapsar al receptor. Este colapso sería posible si el emisor enviara datos sin esperar la confirmación de los bytes ya enviados. (pp. 21-22)

2.5.2 Protocolo de Datagramas de Usuario

El protocolo UDP “proporciona un servicio no orientado a conexión para los procedimientos de la capa de aplicación. Según esto último, es un servicio no seguro, lo cual implica que la entrega y la protección frente a duplicados no están garantizadas” (Marrero, 2000, p. 43).

UDP por lo general se suele utilizar para realizar streaming o comunicaciones que necesiten el envío de mucha información en el que no sea importante que falle en llegar un mensaje, porque, como se mencionó anteriormente la entrega de la información no está garantizada. UDP es mucho más simple que TCP, pero mantiene el encabezado, aunque, mucho más corto (Pineda y Méndez, 2018).

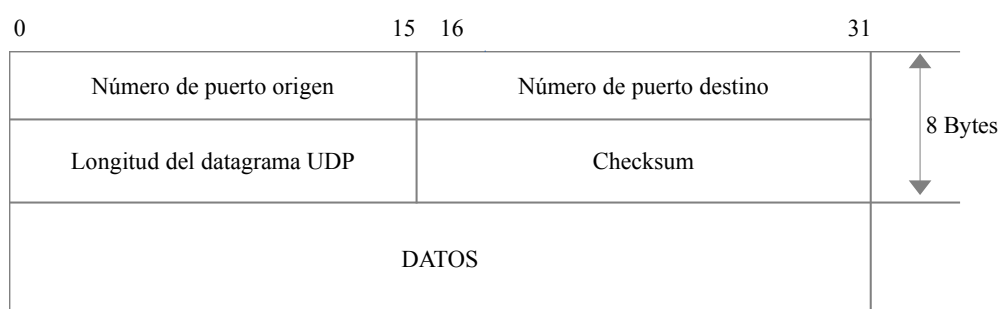


Figura 8: *Cabecera de un datagrama UDP.*

Fuente: Cabecera de un datagrama UDP (p. 20), por Gabriel Verdejo Alvarez , 2003, Universitat Autònoma de Barcelona.

En la Figura 8 se puede observar cómo se compone la cabecera de un datagrama UDP.

“El número de puerto (Port) se utiliza en la comunicación entre dos ordenadores para diferenciar las diferentes conexiones existentes. Si tenemos varias comunicaciones desde nuestro ordenador, al recibir un datagrama IP debemos saber a cuál de las conexiones pertenece” (Verdejo Alvarez, 2003, pp. 19-20)

“La longitud del datagrama UDP hace referencia al tamaño del datagrama en bytes, y engloba la cabecera más los datos que transporta” (Verdejo Alvarez, 2003, p. 20).

El campo de checksum, sirve como método de control de los datos, verificando que no han sido alterados. Este checksum cubre tanto la cabecera UDP como

los datos enviados. Si se detecta un error en el checksum, el datagrama es descartado sin ningún tipo de aviso (Verdejo Alvarez, 2003, p. 20).

2.5.3 Protocolo secure shell

El protocolo SSH (también conocido como Secure Shell) es un método para el inicio de sesión remoto seguro de una computadora a otra. Proporciona varias opciones alternativas para una autenticación fuerte y protege la seguridad e integridad de las comunicaciones con un cifrado fuerte (SSH.COM, s. f.).

En SSH.COM (s. f.) mencionan que el protocolo se utiliza en redes corporativas para:

- Proporcionar acceso seguro para usuarios y procesos automatizados.
- Transferencias de archivos interactivas y automatizadas.
- Emisión de comandos remotos.
- Administrar la infraestructura de red y otros componentes del sistema de misión crítica.

El protocolo funciona en el modelo cliente - servidor, lo que significa que la conexión la establece el cliente SSH que se conecta al servidor SSH. El cliente SSH dirige el proceso de configuración de la conexión y utiliza criptografía de clave pública para verificar la identidad del servidor SSH. Después de la fase de configuración, el protocolo SSH utiliza un cifrado simétrico fuerte y algoritmos hash para garantizar la privacidad y la integridad de los datos que se intercambian entre el cliente y el servidor (SSH.COM, s. f.).

Protocolo de copia segura

Protocolo de copia segura (SCP, del inglés Secure Copy Protocol). Es una utilidad de línea de comandos que permite al usuario copiar archivos y directorios de forma segura entre dos ubicaciones, generalmente entre sistemas Unix o Linux (Nyakundi, 2021).

El protocolo garantiza que la transmisión de archivos esté encriptada para evitar que otras personas puedan tener acceso al mismo. También es importante tener en cuenta que SCP utiliza cifrado a través de una conexión SSH (Secure Shell), lo que garantiza que los datos que se transfieren estén protegidos contra ataques sospechosos (Nyakundi, 2021).

2.6 SOCKETS

Fúquene Ardila (2011) define al socket (Ver Figura 9) como:

Un método para la comunicación entre un programa del cliente y un programa del servidor en una red. Un socket se define como el punto final en una conexión. Los sockets se crean y se utilizan con un sistema de peticiones o de llamadas de función, a veces llamados interfaz de programación de aplicación de sockets.

Un socket es también una dirección de Internet combinando una dirección IP y un número de puerto. (p. 49)

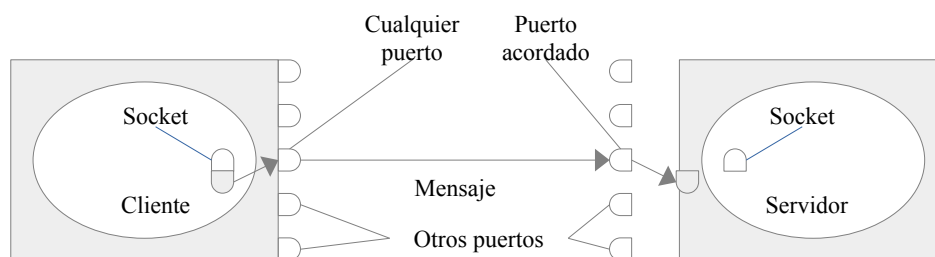


Figura 9: Comunicación entre sockets.

Fuente: Sockets y puertos (p. 120), por George Coulouris, Jean Dollimore y Tim Kindberg, 2001, Pearson Education.

2.6.1 Sockets según su orientación

- **Orientado a conexión:** Establece un camino virtual entre servidor y cliente, fiable sin pérdidas de información ni duplicados, la información llega en el mismo orden que se envía. El cliente abre una sesión en el servidor y este guarda un estado del cliente. (Fúquene Ardila, 2011, p. 49)
- **Orientado a no conexión:** Envío de datagramas de tamaño fijo. No es fiable, puede haber pérdidas de información y también duplicados, la información puede llegar en distinto orden del que se envía. No se guarda ningún estado del cliente en el servidor, por ello, es más tolerante a fallos del sistema. (Fúquene Ardila, 2011, p. 50)

2.6.2 Arquitectura cliente – servidor

Según Fúquene Ardila (2011) la arquitectura cliente - servidor:

Es un modelo de aplicación distribuida en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. Un programa cliente realiza peticiones a otro programa, el servidor, que le da respuesta. (p. 50)

Servidor corresponde a la instancia en que un proceso crea un socket. Cliente corresponde a la instancia en que un proceso se conecta a un socket que se ha creado en otro proceso. Una vez que el socket servidor está habilitado, escucha conexiones y las acepta según su capacidad definida al crearse (Pineda y Méndez, 2019, p. 3).

Como se puede observar en la Figura 10, el servidor es el encargado de responder las peticiones de uno o más clientes

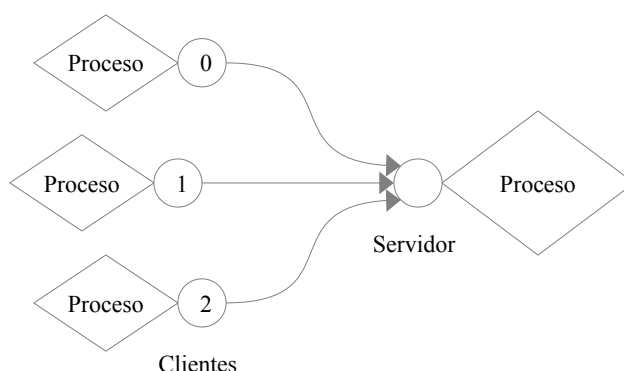


Figura 10: Ejemplo básico de Cliente-Servidor.

Fuente: Ejemplo de Cliente Servidor (p. 3), por David Pineda y Daniel Méndez , 2018, Universidad de Chile.

Según Fúquene Ardila (2011):

El mecanismo de comunicación vía sockets tiene los siguientes pasos:

- El proceso servidor crea un socket con nombre y espera la conexión.
- El proceso cliente crea un socket sin nombre.
- El proceso cliente realiza una petición de conexión al socket servidor.

- El cliente realiza la conexión a través de su socket mientras el proceso servidor mantiene el socket servidor original con nombre. (p. 50)

2.7 DIFUSIÓN DE DATOS

2.7.1 Unidifusión

El proceso de unidifusión (del inglés Unicast), consiste en transmitir paquetes de datos al destino. En el momento de reenviar el paquete de datos, este método de transmisión de mensajes usa la dirección de destino en el paquete de datos para buscarlo en la tabla de enrutamiento (Kaur, Singh y Singh, 2016).

Usando la técnica de unidifusión, existe solo un emisor y un receptor. Si algún emisor necesita o debe enviar un mensaje a varios destinos, tendrá que enviar varios mensajes de unidifusión, cada uno de esos mensajes debe ser dirigido a un destino. Como podemos ver en la Figura 11, los mensajes de unidifusión se enviarán a dispositivos específicos utilizando la dirección IP específica del dispositivo, como dirección de destino en el paquete (Cicnavi, 2011).

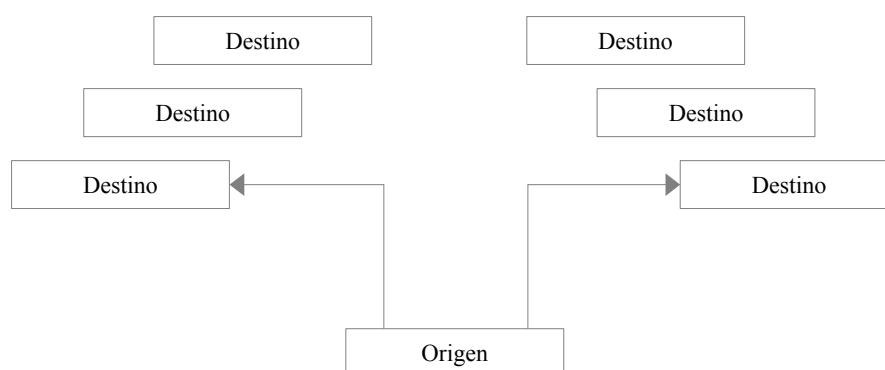


Figura 11: Envío de paquete, utilizando la técnica de unidifusión.

Fuente: Unicast message Transmission in MANETs (p. 16), por Shivraj Kaur, Kulwinder Singh y Yadvinder Singh, 2016, International Journal of Computer Applications.

2.7.2 Difusión

Difusión (del inglés Broadcast) “se utiliza para enviar paquetes a todos los hosts en la red usando la dirección de broadcast para la red” (Cisco, 2022).

Como se puede visualizar en la Figura 12, si el paquete tiene una dirección broadcast, todos los dispositivos que reciban ese mensaje lo procesarán. Entonces, todos los dispositivos en el mismo segmento de red verán el mismo mensaje (Cicnavi, 2011).

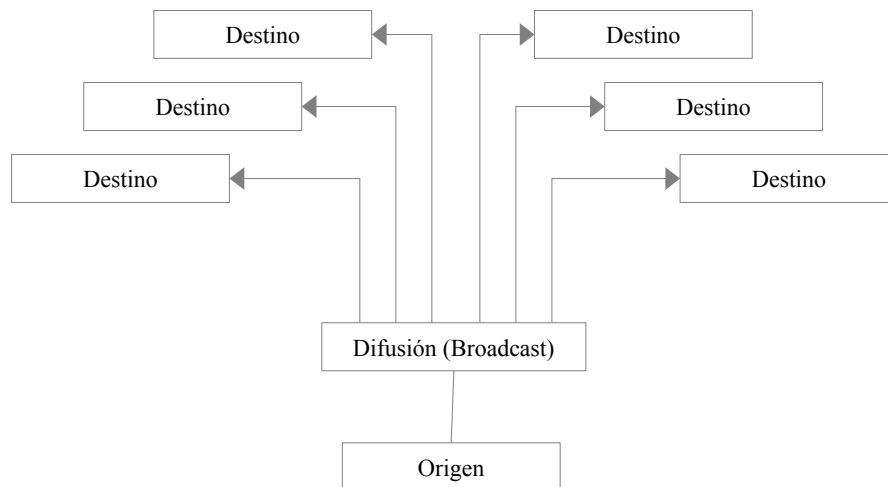


Figura 12: *Envío de paquete utilizando Difusión.*

Fuente: Broadcast message Transmission in MANETs (p. 17), por Shivraj Kaur, Kulwinder Singh y Yadvinder Singh , 2016, International Journal of Computer Applications.

2.8 LINUX

Según Petersen (2008):

Linux es un sistema operativo de código abierto rápido y estable para computadoras personales (PC) y estaciones de trabajo que cuentan con servicios de Internet de nivel profesional, amplias herramientas de desarrollo, interfaces gráficas de usuario completamente funcionales y una gran cantidad de aplicaciones que van desde suites de oficina hasta aplicaciones multimedias. Junto con las capacidades del sistema operativo de Linux, vienen potentes funciones de red, que incluyen soporte para Internet, intranets y redes de Windows. Como norma, las distribuciones de Linux incluyen servidores de Internet rápidos, eficientes y estables, como la web, el Protocolo de transferencia de archivos (FTP) y los servidores DNS, junto con servidores proxy, de noticias y de correo. En otras palabras, Linux tiene todo lo que necesita para configurar, admitir y mantener una red completamente funcional. (p.3).

2.8.1 Distribuciones Linux

Aunque sólo existe una versión estándar de Linux, en realidad hay varias distribuciones diferentes. Diferentes empresas y grupos han empaquetado Linux y el software de Linux de formas ligeramente diferentes. Algunas de las distribuciones más populares son Red Hat, Ubuntu, Mepis, SUSE, Fedora y Debian. El núcleo de Linux se distribuye de forma centralizada a través de kernel.org. Todas las distribuciones utilizan este mismo núcleo, aunque puede estar configurado de forma diferente (Petersen, 2008).

Las distribuciones líderes por lo general han existido por un tiempo y están bien establecidas. También suelen admitir varias arquitecturas y se traducen a varios idiomas. Algunos provienen de empresas que proveen contratos de servicio y soporte para sus productos, otros son proyectos comunitarios. («The LWN.net Linux Distribution List», 2021)

En la Tabla 4 se nombran algunas distribuciones líderes de Linux según LWN.net.

Sistema Operativo	URL	Año de Anuncio o Lanzamiento
Android	http://www.android.com/	Android se anunció por primera vez el 5 de noviembre de 2007.
Debian GNU/Linux	http://www.debian.org/	El Proyecto Debian es una de las distribuciones más antiguas, data de 1993
Fedora	http://getfedora.org/	El primer lanzamiento de Fedora Core data del 5 de noviembre de 2003.
OpenSUSE	https://www.opensuse.org/	openSUSE se abrió para el desarrollo de la comunidad con el lanzamiento de SUSE Linux 10.0, con fecha del 6 de octubre de 2005.
Red Hat Enterprise Linux	http://www.redhat.com/	En 2003, la empresa anunció su decisión de abandonar su popular Red Hat Linux para concentrarse en la línea Red Hat Enterprise Linux.
Slackware Linux	http://www.slackware.com/	Es la distribución de Linux activa más antigua con el primer lanzamiento fechado el 16 de julio de 1993.
SUSE Linux	http://www.suse.com/	SUSE Linux existe desde 1994, lo que la convierte en una de las distribuciones de Linux más antiguas.
Ubuntu	http://www.ubuntu.com/	La primera versión de Ubuntu fue 4.10 Preview "Warty Warthog", lanzada el 15 de septiembre de 2004.

Tabla 4: *Distribuciones Linux.*

Fuente: The LWN.net Linux Distribution List, 2021.

A menudo existe una fina línea entre las distribuciones "líderes" y las "conocidas". Algunos podrían considerar que algunas de ellas son distribuciones "líderes". Entre las distribuciones conocidas tenemos a Arch Linux, CentOS, Gentoo Linux, Linux Mint, etc. Luego están las distribuciones diseñadas para ser fáciles de usar y de instalar, por ejemplo: Chakra, Elementary OS, Linux Lite, etc. También se tienen distribuciones adecuadas para aplicaciones de escritorio y de servidor como ser: Bodhi, ArchBang, Manjaro Linux, etc. («The LWN.net Linux Distribution List», 2021).

2.8.2 Shell

Según Blum y Bresnahan (2015) El shell de Linux es una utilidad interactiva especial. Proporciona una manera para que los usuarios inicien programas, administren archivos (en el sistema de archivos) y administren procesos que se ejecutan en el sistema Linux.

El núcleo del shell es el símbolo de sistema (del inglés command prompt). El símbolo de sistema es la parte interactiva del shell que permite ingresar comandos de texto, y luego interpreta los comandos y los ejecuta en el kernel (Blum y Bresnahan, 2015).

El shell contiene un conjunto de comandos internos que utiliza para controlar cosas como copiar archivos, mover archivos, renombrar archivos, mostrar los programas que se ejecutan actualmente en el sistema y detener los programas que se ejecutan en el sistema. Además de los comandos internos, el shell también le permite ingresar el nombre de un programa en el símbolo del sistema, luego le pasa el nombre del programa al Kernel para iniciarlo (Blum y Bresnahan, 2015).

También puede agrupar comandos de shell en archivos para ejecutarlos como un programa. Esos archivos se denominan scripts de shell. Cualquier comando que pueda ejecutar desde la línea de comandos puede colocarse en un script de shell y ejecutarse como un grupo de comandos. Esto proporciona una gran flexibilidad en la creación de utilidades para comandos que se ejecutan con frecuencia o procesos que requieren varios comandos agrupados. En el mundo UNIX/Linux existen varios tipos de shells que se diferencian entre sí, básicamente, en la sintaxis de sus órdenes y en la interacción con el usuario (Blum y Bresnahan, 2015).

2.8.3 Script Shell Bash

El shell predeterminado que se usa en todas las distribuciones de Linux es el bash shell. Dicho shell fue desarrollado por el proyecto GNU como reemplazo del shell estándar de Unix, llamado Bourne Shell en honor a su creador (Blum y Bresnahan, 2015).

Bash es un shell compatible con sh que incorpora características útiles del Korn shell (ksh) y el C shell (csh). Está diseñado para cumplir con el estándar IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools. Ofrece mejoras funcionales sobre sh tanto para programación como para uso interactivo; estos incluyen edición de línea de comandos,

historial de comandos de tamaño ilimitado, control de trabajos, funciones de shell y alias, matrices indexadas de tamaño ilimitado y aritmética de enteros en cualquier base de dos a sesenta y cuatro. Bash puede ejecutar la mayoría de los scripts sh sin modificaciones (Garrels, 2008).

Un archivo de script, básicamente es una secuencia ordenada de comandos que debe ser ejecutada por un intérprete de comandos correspondiente. La forma en que un intérprete lee un archivo de script varía y hay diferentes formas de hacerlo en una sesión del shell Bash, pero el intérprete por defecto de un archivo de script se indica en la primera línea del archivo script, justo después de los caracteres `#!` (a esto se lo conoce como shebang). En un script con instrucciones para el bash, la primera línea debe ser `#!/bin/bash`, al iniciar esa línea, el intérprete de todas las instrucciones del archivo será `/bin/bash` (Linux Professional Institute, s.f.).

2.9 LENGUAJES DE PROGRAMACIÓN

Un programa informático se define como un conjunto de instrucciones que, una vez ejecutado, realiza una o varias tareas en una computadora. De esta forma, sin programas, una computadora no puede realizar las actividades para las que fue diseñada y creada.

Un programa se escribe con instrucciones en un **lenguaje de programación**, el cual, a su vez, está definido por su sintaxis, que establece e indica las reglas de escritura (la gramática), y por la semántica de los tipos de datos, instrucciones, definiciones, y todos los otros elementos que constituyen un programa.

(Juganaru Mathieu, 2014, p. 5)

A lo largo del tiempo fueron surgiendo varios lenguajes de programación, en la Figura 13 se puede observar una lista de lenguajes de programación, ordenados cronológicamente.

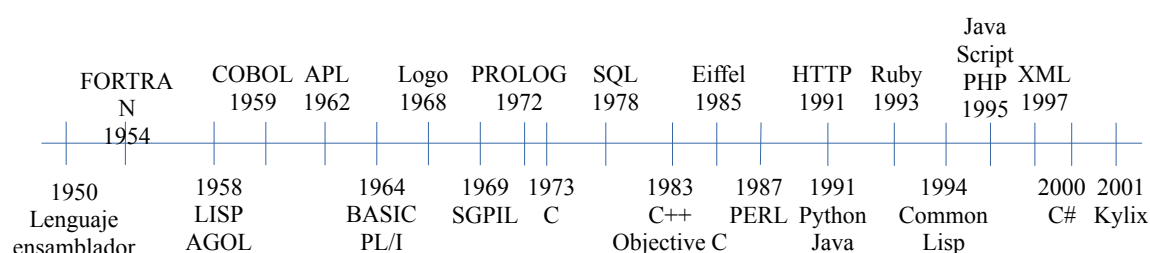


Figura 13: Línea del tiempo de varios lenguajes de programación.

Fuente: Línea de tiempo de los lenguajes de programación (p. 8), por Mihaela Juganaru Mathieu, 2014, Grupo Editorial Patria.

Como se puede observar en la Figura 13, en 1993 surgió el lenguaje de programación Ruby cuyo creador Yukihiro Matsumoto, mezcló partes de sus lenguajes favoritos como: Perl, Smalltalk, Eiffel, Ada y Lisp para formar uno nuevo que incorporara tanto la programación funcional como la imperativa («Acerca de Ruby», s. f.).

2.10 RUBY

Ruby es un lenguaje de programación orientado a objetos (OO) simple y poderoso, la misma se puede utilizar para escribir servidores, experimentar con prototipos y para tareas de programación diarias (Maeda, 2022).

Desde su liberación pública en 1995, Ruby ha atraído devotos desarrolladores de todo el mundo. En el 2006, Ruby alcanzó reconocimiento masivo, formándose grupos de usuarios activos en las ciudades más importantes del mundo y llenando las capacidades de las conferencias relacionadas a Ruby.

El índice TIOBE, que mide el crecimiento de los lenguajes de programación, ubica a Ruby entre los diez mejores del ranking mundial. Gran parte del crecimiento se atribuye a la popularidad del software escrito en Ruby, particularmente el framework Ruby on Rails. («Acerca de Ruby», s. f.)

2.10.1 Hilos

Los programas tradicionales tienen un único hilo de ejecución: las sentencias o instrucciones que componen el programa se ejecutan secuencialmente hasta que el programa termine (Flanagan y Matsumoto, 2008).

Un programa multihilo tiene más de un hilo de ejecución. Dentro de cada hilo, las sentencias se ejecutan secuencialmente, pero los propios hilos pueden ejecutarse en paralelo en una CPU multinúcleo. A menudo, en una máquina con una sola CPU, los hilos múltiples no se ejecutan realmente en paralelo, sino que el paralelismo se simula intercalando la ejecución de los hilos. Ruby facilita la escritura de programas multihilo con la clase Thread. Los hilos de Ruby son una forma ligera y eficiente de lograr la concurrencia en tu código. (Flanagan y Matsumoto, 2008)

2.10.2 Logs de ejecución de programas

El registro es una parte importante del ciclo de vida de cada software. Tener un buen sistema de registro se convierte en una característica clave que ayuda a los desarrolladores, administradores de sistemas y equipos de soporte a comprender y resolver los problemas que aparecen. Cada mensaje de registro tiene un nivel de registro asociado y esos niveles ayudan a comprender la gravedad y la urgencia del mensaje. Por lo general, cada nivel de

registro tiene un número entero asignado que representa la gravedad del mensaje (“How to Start Logging With Ruby on Rails”, 2022).

Un ejemplo para el registro de Log en Ruby es la clase `Logger` que proporciona una utilidad de registro simple pero sofisticada que se puede usar para generar mensajes. Los mensajes tienen asociados niveles, los cuales son:

- UNKNOWN: Un mensaje desconocido que siempre debe registrarse.
- FATAL: Un error inmanejable que provoca un bloqueo del programa.
- ERROR: Una condición de error manejable.
- WARN: Una advertencia.
- INFO: Información genérica (útil) sobre el funcionamiento del sistema.
- DEBUG: Información de bajo nivel para desarrolladores (“Logger,” s. f.).

2.11 GEMA

Los programas en Ruby suelen distribuirse en paquetes de software llamados gemas. Una gema contiene una aplicación o biblioteca Ruby empaquetada. Las gemas pueden utilizarse para ampliar o modificar la funcionalidad de las aplicaciones Ruby. Comúnmente se usan para distribuir funcionalidad reutilizable que se comparte con otros programadores Ruby, para que las usen en sus aplicaciones y bibliotecas. Algunas gemas proporcionan utilidades de línea de comandos para ayudar a automatizar tareas y acelerar el trabajo («Guides - RubyGems Guides», 2022).

Existe una comunidad Ruby llamada RubyGems.org que ofrece un servicio de alojamiento de gemas («RubyGems.org | your community gem host», 2022). Esta comunidad ofrece un software que permite descargar, instalar y utilizar fácilmente paquetes de software empaquetados como gemas («Guides - RubyGems Guides», 2022).

El comando `gem` permite interactuar con RubyGems («RubyGems Basics - RubyGems Guides», 2022). El comando **`gem push`** se encarga de publicar la gema en RubyGems.org, esto hace que la gema sea accesible para que cualquier persona pueda instalar la gema a través del comando **`gem install`** para posteriormente usarla («Publishing your gem - RubyGems Guides», 2022).

2.11.1 Estructura de una Gema

Dentro de las gemas se encuentran los siguientes componentes: Código (incluyendo pruebas y otras utilidades), Documentación y el `gemspec` («What is a gem? - RubyGems Guides», 2022).

Según «What is a gem? - RubyGems Guides» (2022) todas las gemas siguen una estructura estándar para organizar el código como se representa en la Figura 14.

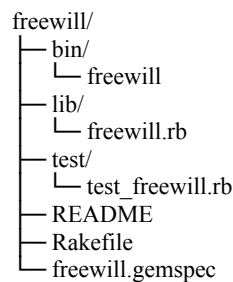


Figura 14: Estructura de carpetas de una gema.

Fuente: What is a gem? - RubyGems Guides, 2022.
<https://guides.rubygems.org/what-is-a-gem/>

De acuerdo a “What is a gem? - RubyGems Guides” (2022), los componentes más importantes de una gema son:

- La carpeta **lib** que contiene el código de la gema.
- Las carpetas **test** o **spec** que contienen las pruebas, dependiendo del framework de pruebas que el desarrollador utilice.
- Una gema usualmente tiene un **Rakefile**, que junto al programa rake (que también es una gema) se usa para automatizar pruebas, generar código, y realizar otras tareas.
- La gema también incluye un archivo ejecutable en la carpeta **bin**, que será cargado en el entorno donde el usuario instaló la gema.
- La documentación usualmente se incluye en el archivo **README** y dentro del código mismo. Cuando se instala una gema, la documentación es generada automáticamente. Muchas gemas incluyen RDoc y YARD las cuales son gemas para generar documentación.
- La pieza final es el **gemspec**, que contiene información sobre la gema, como por ejemplo: los archivos de la gema, la información de las pruebas, la plataforma, el número de versión, el correo electrónico y nombre del autor/es.

2.11.2 Creación de una Gema

Para implementar una gema se pueden seguir las guías paso a paso de la página RubyGems.org, pero tener que crear manualmente la estructura necesaria para la gema sería un poco tedioso, afortunadamente, existe una gema llamada Bundler que facilita este proceso.

Bundler es una herramienta creada para manejar las dependencias en las librerías Ruby y también para facilitar la creación de gemas («Bundler: How to create a Ruby gem with Bundler», 2022).

Bundler es una gema Ruby, por lo tanto, requiere tener Ruby instalado en el sistema operativo. La mejor forma de instalar Ruby es a través de un gestor de versiones. Los dos gestores más populares según («Ruby Version Management - The Ruby Toolbox», 2022) son Rbenv y RVM (Ruby Version Manager). Para el presente TFG se utilizó RVM.

RVM es una herramienta de línea de comandos que permite instalar, gestionar y trabajar fácilmente con múltiples versiones de ruby, desde intérpretes hasta conjuntos de gemas («RVM: Ruby Version Manager - Documentation», 2022).

En el Anexo A se procede a detallar los pasos para instalar RVM, Ruby y Bundler.

2.12 GESTIÓN DE VERSIONES

Atlassian Corporation (2021) menciona:

El control de versiones, también conocido como "control de código fuente", es la práctica de rastrear y gestionar los cambios en el código de software. Los sistemas de control de versiones son herramientas de software que ayudan a los equipos de software a gestionar los cambios en el código fuente a lo largo del tiempo. A medida que los entornos de desarrollo se aceleran, los sistemas de control de versiones ayudan a los equipos de software a trabajar de forma más rápida e inteligente.

El sistema de control de versiones realiza un seguimiento de todas las modificaciones en el código en un tipo especial de base de datos. Si se comete un error, los desarrolladores pueden ir hacia atrás en el tiempo y comparar las versiones anteriores del código para ayudar a resolver el error, al tiempo que se minimizan las interrupciones para todos los miembros del equipo.

Los desarrolladores de software que trabajan en equipos están escribiendo continuamente nuevo código fuente y cambiando el que ya existe. El código de un proyecto, una aplicación o un componente de software normalmente se organiza en una estructura de carpetas o "árbol de archivos". Un desarrollador del equipo podría estar trabajando en una nueva función mientras otro desarrollador soluciona un error no relacionado cambiando código. Cada desarrollador podría hacer sus cambios en varias partes del árbol de archivos. El control de versiones ayuda a los equipos a resolver este tipo de problemas al realizar un seguimiento de todos los cambios individuales de cada colaborador y al contribuir a evitar que el trabajo concurrente entre en conflicto.

Existen varios sistemas de control de versiones. Anwer (2020) menciona que algunos de los sistemas más populares del 2021 fueron: Git, Subversión (también conocido como SVN), GitLab, Mercurial y Bazaar.

2.12.1 Git

Hoy en día Git es, con diferencia, el sistema de control de versiones moderno más utilizado del mundo. Git es un proyecto de código abierto maduro y con un mantenimiento activo que desarrolló originalmente Linus Torvalds, en 2005. Una gran cantidad de proyectos de software dependen de Git para el control de versiones, incluidos proyectos comerciales y de código abierto. Este sistema funciona a la perfección en una amplia variedad de sistemas operativos y entornos de desarrollo integrados (IDE, del inglés Integrated Development Environment). (Atlassian, 2022)

Git, presenta una arquitectura distribuida. En lugar de tener un único espacio para todo el historial de versiones del software, como sucede de manera habitual en los sistemas de control de versiones antaño populares, como CVS o SVN, en Git, la copia de trabajo del código de cada desarrollador es también un repositorio que puede albergar el historial completo de todos los cambios. (Atlassian, 2022)

2.12.2 GitHub

GitHub es un servicio basado en la nube que aloja un sistema de control de versiones llamado Git. Este permite a los desarrolladores colaborar y realizar cambios en proyectos compartidos, a la vez se puede mantener un seguimiento detallado de los progresos (Bustos, 2021).

GitHub aloja más de 100 millones de repositorios, la mayoría de los cuales son proyectos de código abierto. Esta estadística revela que GitHub se encuentra entre los clientes Git GUI más populares y es utilizado por varios profesionales y grandes empresas.

Además, la interfaz de usuario de GitHub es más fácil de usar que la de Git, lo que la hace accesible para personas con pocos o ningún conocimiento técnico. Esto significa que se puede incluir a más miembros del equipo en el progreso y la gestión de un proyecto, haciendo que el proceso de desarrollo sea más fluido. (Bustos, 2021)

2.13 SCRUM

Schwaber y Sutherland (2020) lo definen de la siguiente manera “Scrum es un marco de trabajo liviano que ayuda a las personas, equipos y organizaciones a generar valor a través de soluciones adaptativas para problemas complejos” (p. 3).

Scrum se basa en la inteligencia colectiva de las personas que lo utilizan. En lugar de proporcionar a las personas instrucciones detalladas, las reglas de Scrum guían sus relaciones e interacciones.

En este marco de trabajo pueden emplearse varios procesos, técnicas y métodos. Scrum envuelve las prácticas existentes o las hace innecesarias. Scrum hace visible la eficacia relativa de las técnicas actuales de gestión, entorno y trabajo, de modo que se puedan realizar mejoras. (Schwaber y Sutherland, 2020, p. 3)

“El marco de trabajo Scrum consiste de Equipos Scrum (Scrum Teams), roles, eventos, artefactos y reglas asociadas. Cada componente dentro del marco de trabajo sirve a un propósito específico y es esencial para el éxito de Scrum y para su uso” (Schwaber y Sutherland, 2013, p. 4).

2.13.1 Eventos de Scrum

“El Sprint es un contenedor para todos los demás eventos. Cada evento en Scrum es una oportunidad formal para inspeccionar y adaptar los artefactos Scrum. Estos eventos están diseñados específicamente para habilitar la transparencia requerida” (Schwaber y Sutherland, 2020).

El Sprint

Los Sprints son el corazón de Scrum, donde las ideas se convierten en valor. Son eventos de duración fija de un mes o menos para crear consistencia. Un nuevo Sprint comienza inmediatamente después de la conclusión del Sprint anterior.

Todo el trabajo necesario para lograr el Objetivo del Producto, incluido el Sprint Planning, Daily Scrums, Sprint Review y Sprint Retrospective, ocurre dentro de los Sprints. (Schwaber y Sutherland, 2020, p. 7)

Durante el Sprint:

- No se realizan cambios que pongan en peligro el Objetivo del Sprint;
- La calidad no disminuye;
- El Product Backlog se refina según sea necesario; y,
- El alcance se puede aclarar y renegociar con el Product Owner a medida que se aprende más. (p. 8)

Sprint Planning

El Sprint Planning inicia al establecer el trabajo que se realizará para el Sprint. El Scrum Team crea este plan resultante mediante trabajo colaborativo. El Product Owner se asegura de que los asistentes estén preparados para discutir los elementos más importantes del Product Backlog y cómo se relacionan con el Objetivo del Producto. (Schwaber y Sutherland, 2020, p. 8)

Schwaber y Sutherland (2020) menciona que en el Sprint Planning se abordan temas como las siguientes:

- ¿Por qué es valioso este Sprint?
- ¿Qué se puede hacer en este Sprint?
- ¿Cómo se realizará el trabajo elegido?.

Con estos temas se pretende definir el objetivo del sprint con el Scrum Team, también seleccionar los elementos del Product Backlog para incluirlos en el Sprint actual para finalmente descomponerlos en tareas más pequeñas que sean realizables en un día o menos y así introducirlos al Sprint Backlog.

Daily Scrum

“Daily Scrum es un evento de 15 minutos para los Developers del Scrum Team. Para reducir la complejidad, se lleva a cabo a la misma hora y en el mismo lugar todos los días hábiles del Sprint” (Schwaber y Sutherland, 2020, p. 9).

Sprint Review

El propósito de la Sprint Review es inspeccionar el resultado del Sprint y determinar futuras adaptaciones.

Durante el evento, el Scrum Team y los interesados revisan lo que se logró en el Sprint y lo que ha cambiado en su entorno. Con base en esta información, los asistentes colaboran sobre qué hacer a continuación. (Schwaber y Sutherland, 2020, p. 10)

Sprint Retrospective

El propósito de la Sprint Retrospective es planificar formas de aumentar la calidad y la efectividad.

El Scrum Team inspecciona cómo fue el último Sprint con respecto a las personas, las interacciones, los procesos, las herramientas y su definición de terminado. Se identifican los supuestos que los llevaron por mal camino y se exploran sus orígenes. El Scrum Team analiza qué salió bien durante el Sprint, qué problemas encontró y cómo se resolvieron (o no) esos problemas. (Schwaber y Sutherland, 2020, p. 10)

2.13.1 Scrum Team

El Scrum Team consta de un Scrum Master, un Product Owner y Developers. Dentro de un Scrum Team, no hay subequipos ni jerarquías. Es una unidad cohesionada de profesionales enfocados en un objetivo a la vez, el Objetivo del Producto

El Scrum Team es responsable de todas las actividades relacionadas con el producto, desde la colaboración de los interesados, la verificación, el mantenimiento, la operación, la experimentación, la investigación y el desarrollo, y cualquier otra cosa que pueda ser necesaria. Están estructurados y empoderados por la organización para gestionar su propio trabajo. (Schwaber y Sutherland, 2020, p. 5)

Developers

“Las personas del Scrum Team que se comprometen a crear cualquier aspecto de un Incremento utilizable en cada Sprint son Developers” (Schwaber y Sutherland, 2020, p. 5).

Schwaber y Sutherland (2020) mencionan que los Developers son responsables de:

- Crear un plan para el Sprint, el Sprint Backlog.

- Inculcar calidad al adherirse a una definición de Terminado.
- Adaptar su plan cada día hacia el Objetivo del Sprint.
- Responsabilizarse mutuamente como profesionales. (pp. 5–6)

Product Owner

Según Schwaber y Sutherland (2020) “El Product Owner es responsable de maximizar el valor del producto resultante del trabajo del Scrum Team. La forma en que esto se hace puede variar ampliamente entre organizaciones, Scrum Teams e individuos” (p. 6).

El Product Owner también es responsable de la gestión efectiva del Product Backlog, lo que incluye:

- Desarrollar y comunicar explícitamente el Objetivo del Producto.
- Crear y comunicar claramente los elementos del Product Backlog.
- Ordenar los elementos del Product Backlog.
- Asegurarse de que el Product Backlog sea transparente, visible y se entienda.

El Product Owner puede realizar el trabajo anterior o puede delegar la responsabilidad en otros. Independientemente de ello, el Product Owner sigue siendo el responsable de que el trabajo se realice. (Schwaber y Sutherland, 2020, p. 6)

Scrum Master

Schwaber y Sutherland (2020) mencionan que:

El Scrum Master es responsable de establecer Scrum como se define en la Guía de la misma. Lo hace ayudando a todos a comprender la teoría y la práctica de Scrum, tanto dentro del Scrum Team como de la organización.

El Scrum Master sirve al Scrum Team de varias maneras, que incluyen:

- Guiar a los miembros del equipo en ser autogestionados y multifuncionales.
- Ayudar al Scrum Team a enfocarse en crear incrementos de alto valor que cumplan con la Definición de Terminado.

- Procurar la eliminación de impedimentos para el progreso del Scrum Team.
- Asegurarse de que todos los eventos de Scrum se lleven a cabo y sean positivos, productivos y se mantengan dentro de los límites de tiempo recomendados.

El Scrum Master sirve al Product Owner de varias maneras, que incluyen:

- Ayudar a encontrar técnicas para una definición efectiva de Objetivos del Producto y la gestión del Product Backlog;
 - Ayudar al Scrum Team a comprender la necesidad de tener elementos del Product Backlog claros y concisos;
 - Ayudar a establecer una planificación empírica de productos para un entorno complejo; y,
 - Facilitar la colaboración de los interesados según se solicite o necesite.
- (pp. 6–7)

2.13.2 Artefactos de Scrum

Schwaber y Sutherland (2020) mencionan que “Los artefactos de Scrum representan trabajo o valor y que están diseñados para maximizar la transparencia de la información clave” (p. 10).

Product Backlog

El Product Backlog es una lista ordenada de todo lo que podría ser necesario en el producto, y es la única fuente de requisitos para cualquier cambio a realizarse en el producto. El Product Owner es el responsable de la lista, incluyendo su contenido, disponibilidad y ordenación.

El Product Backlog es dinámico; cambia constantemente para identificar lo que el producto necesita para ser adecuado, competitivo y útil. Mientras el producto exista, el Product Backlog también.

El Product Backlog enumera todas las características, funcionalidades, requisitos, mejoras y correcciones que constituyen cambios a ser hechos sobre el producto para entregas futuras. (Schwaber y Sutherland, 2013, p. 15).

Sprint Backlog

El Sprint Backlog se compone del Objetivo del Sprint (por qué), el conjunto de elementos del Product Backlog seleccionados para el Sprint (qué), así como un plan de acción para entregar el Increment (cómo).

El Sprint Backlog es un plan realizado por y para los Developers. Es una imagen muy visible y en tiempo real del trabajo que los Developers planean realizar durante el Sprint para lograr el Objetivo del Sprint. En consecuencia, el Sprint Backlog se actualiza a lo largo del Sprint a medida que se aprende más. Debe tener suficientes detalles para que puedan inspeccionar su progreso en la Daily Scrum. (Schwaber y Sutherland, 2020, p. 11)

Incremento

El Incremento es la suma de todos los elementos del Product Backlog completados durante un Sprint y el valor de los incrementos de todos los Sprints anteriores. Al final de un Sprint, el nuevo Incremento debe estar “Terminado”, lo cual significa que está en condiciones de ser utilizado y que cumple la definición de “Terminado” del Equipo Scrum. (Schwaber y Sutherland, 2013, p. 17)

3 MARCO METODOLÓGICO

3.1 HERRAMIENTAS UTILIZADAS

- **Git:** Se utilizó para el versionamiento del código fuente del sistema, de manera que el equipo pueda trabajar de forma individual, con la creación de ramas y fusión de las mismas si así lo requiriera.
- **GitHub:** Se utilizó como servidor del repositorio Git y para creación/edición de tareas a realizar.
- **ZenHub:** Se utilizó para ver las tareas creadas en GitHub en un tablero dinámico (Ver Figura 15), con estados como columnas para que se pueda ver mejor el panorama del proyecto.
- **Ruby:** Lenguaje de programación seleccionado para el desarrollo del sistema.
- **RubyMine:** Entorno de desarrollo integrado para la elaboración del código fuente. Se utilizó para la escritura, depuración de código, pruebas y el despliegue de la aplicación. Se utilizó esta herramienta con una licencia para estudiantes.
- **VirtualBox:** Para la creación de varias máquinas virtuales con sistemas operativos Linux y con el sistema funcionando en cada una de ellas, para realizar pruebas sin necesidad de tener muchas máquinas físicas funcionando.

3.2 METODOLOGÍA DE DESARROLLO

El desarrollo del sistema se basó en el marco de trabajo Scrum, por la practicidad y dinamismo que ofrece, como también por las experiencias previas de los integrantes del TFG con el marco de trabajo.

Cabe mencionar que no se utilizaron todos los roles y actividades descritas en las prácticas del marco de trabajo. A continuación se procede a listar las características de Scrum utilizadas y/o adaptadas:

Scrum Team

- **Product Owner:** El cual fue ejercido por el Tutor del TFG.
- **Development Team:** Conformado por los integrantes de este TFG.

Eventos

- **Sprint Planning:** En este evento se definieron la lista de actividades a realizar en el Sprint y que luego fueron registradas en la herramienta Zenhub. En la Figura 15 se puede observar un ejemplo de cómo se ven estas actividades en dicha herramienta.
- **Daily Scrum:** Este evento fue realizado casi semanalmente de forma online o presencial. En cada reunión se respondieron las siguientes preguntas: ¿Qué hice? ¿Qué haré? ¿Qué me detiene?.
- **Sprint Review:** se realizaron reuniones con el Product Owner, de inspección y adaptación al lograr una versión del sistema. También se recolectaron sugerencias del mismo para una siguiente versión.

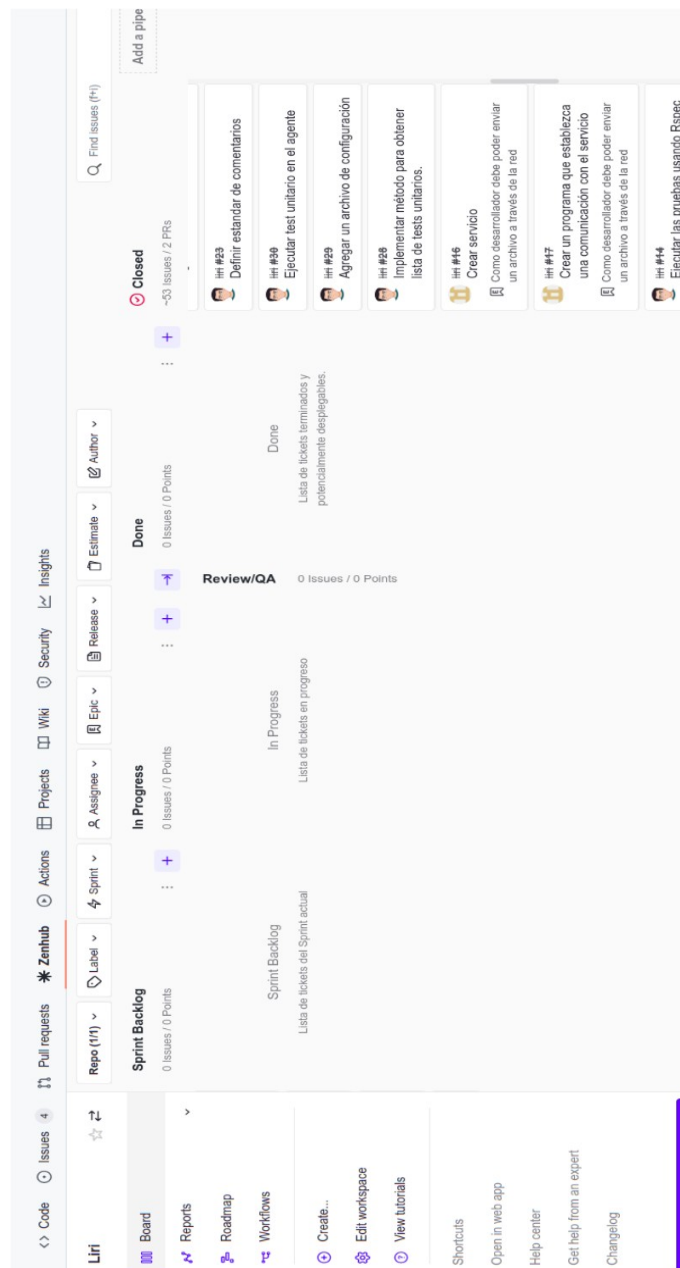


Figura 15: Tablero ZenHub.

Fuente: Elaboración propia.

3.3 PROCESO DE DESARROLLO

Primeramente se comenzó diseñando un **prototipo** de la arquitectura del sistema a implementar, este prototipo define el proceso de comunicación de una arquitectura distribuida entre varias computadoras a fin de coordinar el proceso de ejecución de pruebas unitarias, distribuyendo el esfuerzo de ejecución de las mismas a fin de reducir el tiempo de ejecución.

Seguidamente, se investigó de qué manera lograr el proceso de comunicación descrito en el prototipo de la arquitectura diseñada. Se encontró que a través de un **broadcast UDP** es posible sondear la red y por este medio establecer una comunicación inicial con las computadoras encargadas de ejecutar las pruebas, para luego a través de una conexión **TCP** establecer una comunicación más sólida para coordinar la ejecución de pruebas entre las diferentes computadoras.

Para la distribución del código fuente, primero se comprime la carpeta contenedora del mismo, de este modo se reduce su tamaño, lo cual acelera el proceso de distribución del código fuente entre las máquinas detectadas. Se utilizó el protocolo **SCP** para distribuir el código fuente entre las computadoras identificadas en el proceso de broadcast. Esta distribución de código se realiza en el contexto de la conexión TCP.

El sistema se desarrolló como una **gema** de Ruby para facilitar el proceso de pruebas del sistema, lo que permitió generar, instalar y probar nuevas versiones de la gema a medida que se iba desarrollando el sistema. En la gema, se empaquetaron dos aplicaciones, la Aplicación Coordinadora, que se encarga de iniciar, coordinar y mostrar el proceso de ejecución de pruebas, y, la Aplicación Agente, que se encarga de ejecutar las pruebas y devolver los resultados a la Aplicación Coordinadora.

Para la ejecución de pruebas unitarias, inicialmente se implementó un algoritmo que recorría los archivos de pruebas unitarias e identificaba las líneas correspondientes a las pruebas unitarias utilizando la palabra clave *it* (esta palabra clave se usa en Rspec para definir una prueba unitaria), de este modo se le especificaba a cada Aplicación Agente que pruebas ejecutar. Posteriormente se identificó que la palabra clave *it*, no era la única forma de definir una prueba, también existían otras más. Entonces, como identificar cada prueba

en cada archivo se volvió complejo, se decidió identificar cada archivo de prueba e indicarle a las Aplicaciones Agentes que archivos de prueba ejecutar.

4 SOLUCIÓN IMPLEMENTADA

4.1 ARQUITECTURA DEL SISTEMA

Antes de comenzar a explicar la arquitectura, es importante mencionar que el nombre del sistema es LIRI. Este nombre surgió como una combinación de los nombres de los integrantes del presente TFG, de Les-lie y Ro-dri-go resultó Liri.

Como se mencionó anteriormente, el sistema Liri está compuesto por dos aplicaciones, una Aplicación Coordinadora y una Aplicación Agente.

La Aplicación Coordinadora se comunica con uno o más Aplicaciones Agentes para lograr su cometido. En la Figura 16 se puede observar el diagrama de componentes del sistema Liri representando una Aplicación Coordinadora comunicándose con N Aplicaciones Agentes.

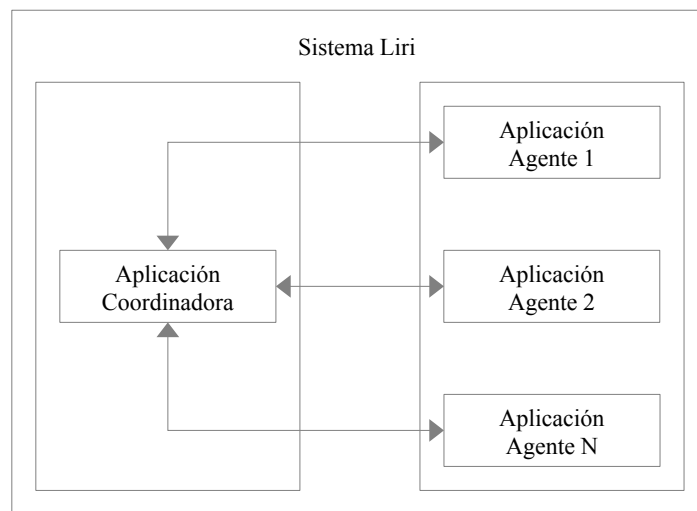


Figura 16: Esquema de componentes del sistema Liri.

Fuente: Elaboración propia.

4.2 DISEÑO DEL ALGORITMO

En la Figura 17 se presenta la secuencia de operaciones realizada por la Aplicación Coordinadora y la Aplicación Agente, así como la interacción entre los mismos como parte del sistema Liri.

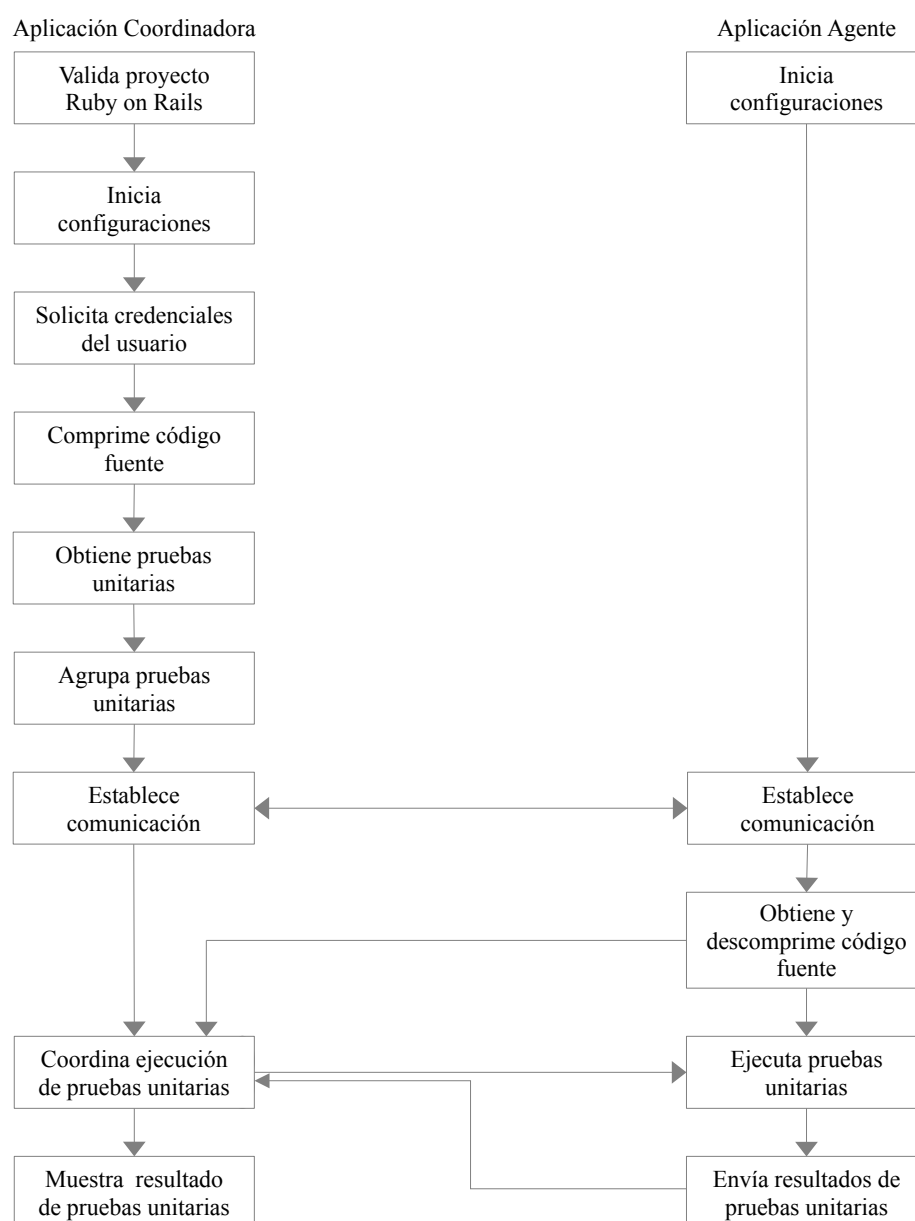


Figura 17: Interacción entre la Aplicación Coordinadora y la Aplicación Agente.

Fuente: Elaboración propia.

A continuación se procede a explicar cada operación realizada por el sistema Liri.

4.2.1 Validación del proyecto Ruby on Rails

La Aplicación Coordinadora valida que su ejecución se realice dentro de un proyecto Ruby on Rails. Para lograr esto, verifica la existencia del archivo Gemfile el cual está presente en cualquier proyecto Ruby on Rails.

4.2.2 Inicio de configuraciones

Al iniciar la Aplicación Coordinadora y la Aplicación Agente, se crea (si es que no existe) la estructura de carpetas que se utilizarán para organizar el proceso de ejecución del sistema.

En la Figura 18 se presenta la estructura de carpetas creada durante la ejecución de la Aplicación Coordinadora.

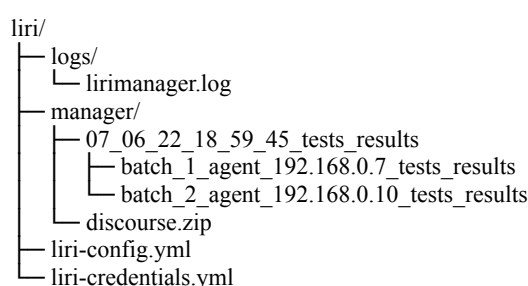


Figura 18: Estructura de carpetas de la Aplicación Coordinadora.

Fuente: Elaboración propia.

En la carpeta **logs**, se guardan los logs de ejecución de la Aplicación Coordinadora. Estos logs se registran usando la librería logger de Ruby.

En la carpeta **manager** se guarda el código fuente comprimido del proyecto Ruby on Rails cuyas pruebas se quiere ejecutar. La compresión del código se realiza usando la gema `rubyzip`. La Aplicación Agente obtendrá el código fuente comprimido usando el protocolo SCP a través de la gema `net-scp`. Dentro de la misma carpeta se crea una carpeta cuyo nombre lleva la fecha y hora en que se inicia la Aplicación Coordinadora. La Aplicación Agente utilizará el protocolo SCP para copiar los archivos de resultados de la ejecución de las pruebas unitarias. Cada vez que se inicia la Aplicación Coordinadora, se crea una nueva carpeta para contener los resultados de las pruebas.

En el archivo **liri-config.yml** se definen algunas configuraciones que determinarán el comportamiento de la Aplicación Coordinadora. Para más detalles sobre el archivo de configuración, ver el Anexo B.

En el archivo **liri-credentials.yml** se guarda la contraseña que se pide al inicio de la ejecución de la Aplicación Coordinadora, de este modo, la contraseña solo se pide en la

primera ejecución. Esta contraseña será utilizada por la Aplicación Agente para conectarse a la computadora donde está ejecutándose la Aplicación Coordinadora y así poder obtener el código fuente comprimido. Cabe aclarar que esta contraseña se guarda como texto plano y sin encriptar en el archivo indicado.

En la Figura 19 se muestra la estructura de carpetas creada durante la ejecución de la Aplicación Agente.

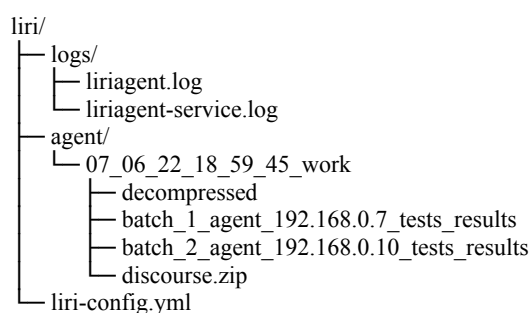


Figura 19: Estructura de carpetas de la Aplicación Agente.

Fuente: Elaboración propia.

La explicación de la estructura de carpetas es la siguiente:

- En la carpeta **logs** se guardan 2 logs referentes a la ejecución de la Aplicación Agente. En **liriagent.log** se registran los logs de la Aplicación Agente usando la librería logger de Ruby y en **liriagent-service.log** se incluye los mismos datos que **liriagent.log** junto a los logs del sistema operativo.
- En la carpeta **agent** se crean carpetas cuyos nombres incluyen la fecha y hora en que una Aplicación Coordinadora se conecta a la Aplicación Agente para iniciar el proceso de ejecución de pruebas unitarias. La Aplicación Agente guarda en esta carpeta el archivo comprimido de código fuente obtenido desde la computadora donde se ejecuta la Aplicación Coordinadora, en esta misma carpeta se descomprime el código fuente y se van guardando los archivos de resultados de la ejecución de las pruebas unitarias. Cada vez que una Aplicación Coordinadora se conecta, se crea una nueva carpeta para contener los resultados de las pruebas.

- En el archivo **liri-config.yml** se definen algunas configuraciones que determinarán el comportamiento de la Aplicación Agente. Para más detalles sobre el archivo de configuración, ver el Anexo B.

4.2.3 Solicitud de credenciales

En la primera ejecución de la Aplicación Coordinadora, se solicita la contraseña del usuario actual del sistema operativo donde se está ejecutando la Aplicación Coordinadora, y, se guarda en el archivo **liri-credentials.yml** para no volver a solicitar esta información en las siguientes ejecuciones.

4.2.4 Compresión de código fuente

La Aplicación Coordinadora comprime el código fuente del proyecto Ruby on Rails donde se está ejecutando y lo guarda dentro de la carpeta manager.

4.2.5 Obtención de pruebas unitarias

La Aplicación Coordinadora utiliza una estructura de datos de tipo diccionario para almacenar los archivos de pruebas unitarias en el siguiente formato:

```
{
  1=>"spec/test_file_one_spec.rb",
  2=>"spec/test_file_two_spec .rb",
  3=>"spec/test_file_three_spec .rb"
}
```

4.2.6 Agrupación de pruebas unitarias

La Aplicación Coordinadora construye otro diccionario que organiza las pruebas unitarias en varios conjuntos. Cada elemento del diccionario contiene cierta cantidad de archivos de pruebas unitarias. Esta cantidad está definida en la configuración *tests_files_by_runner* del archivo **liri-config.yml**. Al momento de construir este diccionario se seleccionan elementos aleatorios del diccionario creado según lo indicado en la sección 4.1.5 Obtención de pruebas unitarias. Por ejemplo, si *tests_files_by_runner* = 2, el diccionario tendrá el siguiente formato:

```
{
  1=>{
    :batch_num=>1,
    :tests_batch_keys=>[3,1],
    :files_count=>2,
    :status=>"pending",
    :files_status=>"2 pending",
    :agent_ip_address=>"",
    :examples=>0,
    :passed=>0,
    :failures=>0,
    :pending=>0,
    :failed_files=>"",
    :files_load=>0,
    :finish_in=>0,
    :batch_run=>0,
    :share_source_code=>0,
    :tests_runtime=>0,
    :hardware_specs=>""
  },
  2=>{
    :batch_num=>2,
    :tests_batch_keys=>[2],
    :files_count=>1,
    :status=>"pending",
    :files_status=>"1 pending",
    :agent_ip_address=>"",
    :examples=>0,
    :passed=>0,
    :failures=>0,
    :pending=>0,
    :failed_files=>"",
    :files_load=>0,
    :finish_in=>0,
    :batch_run=>0,
    :share_source_code=>0,
    :tests_runtime=>0,
    :hardware_specs=>""
  }
}
```

Más adelante se irá aclarando el significado de cada elemento del diccionario.

4.2.7 Establecimiento de comunicación

La Aplicación Coordinadora inicia el proceso de comunicación con uno o más Aplicaciones Agentes, para ello se utiliza un esquema de hilos, así se puede tratar con cada una de las Aplicaciones Agentes de forma paralela.

En la Figura 20 se puede observar el esquema de hilos que componen el proceso realizado por la Aplicación Coordinadora.

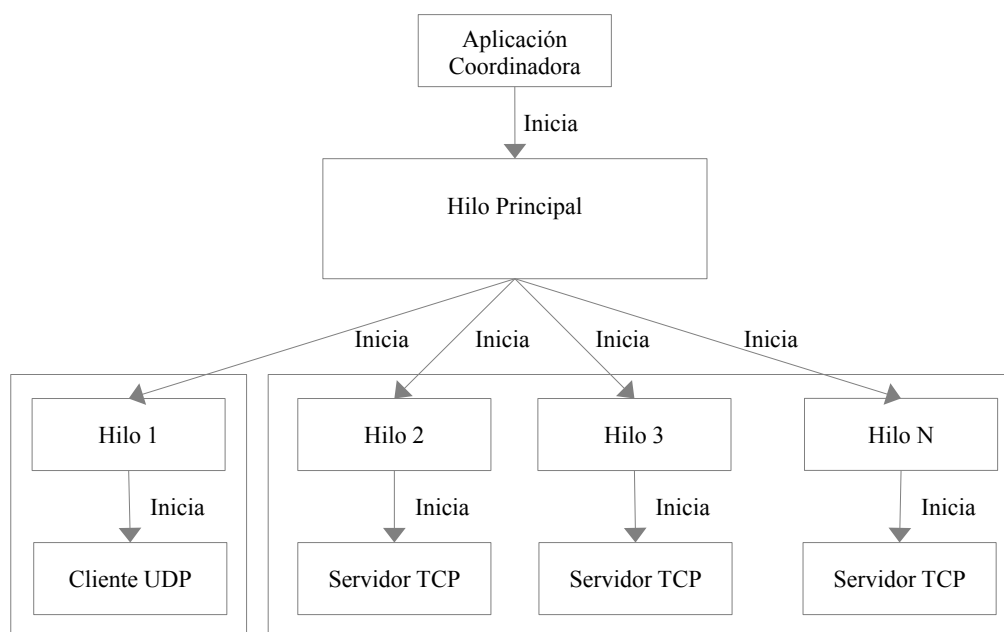


Figura 20: Esquema de proceso de la Aplicación Coordinadora.

Fuente: Elaboración propia.

Dentro del hilo principal, la Aplicación Coordinadora realiza las siguientes acciones:

1. Se inicia el hilo 1, el cual a su vez inicia un socket UDP de tipo cliente. Este cliente emite una petición broadcast a la red cada cierto tiempo con el objetivo de monitorear constantemente la red para conectar con más Aplicaciones Agentes, esto es necesario en caso de que alguna Aplicación Agente se inicie tiempo después de haber iniciado el proceso de ejecución de pruebas. Esta cantidad de tiempo se especifica en la configuración `udp_request_delay` del archivo **liri-config.yml**. En la petición se envía un mensaje con el siguiente formato:

```

{
  "tests_results_folder_path"=>"path/12_06_22_10_14_16_tests_results",
  "compressed_file_path"=>"path/discourse.zip",
  "user"=>"username",
  "password"=>"password"
}

```

En `tests_results_folder_path`, se indica la ruta de la carpeta de la computadora donde se ejecuta la Aplicación Coordinadora en donde se guardarán los archivos de resultados de la ejecución de las pruebas. En `compressed_file_path`, se indica la ruta del archivo de código fuente comprimido en la computadora donde se ejecuta la

Aplicación Coordinadora. En *user*, se indica el usuario del sistema operativo donde se está ejecutando la Aplicación Coordinadora. En *password*, se indica la contraseña del usuario del sistema operativo.

El cliente UDP, al ejecutarse dentro de un hilo, permite que la Aplicación Coordinadora realice otras operaciones a la par en otros hilos.

2. Se inicia el hilo 2, el cual a su vez inicia un socket TCP de tipo servidor. Este servidor se mantiene en espera a que algún cliente TCP se comuniquen desde alguna Aplicación Agente. Por cada Aplicación Agente que se conecte a la Aplicación Coordinadora, se genera un nuevo hilo que se encarga de la comunicación entre la Aplicación Coordinadora y la Aplicación Agente, en la Figura 20 se puede observar la representación de varios servidores TCP ejecutándose en varios hilos hasta el hilo N. A través de esta conexión TCP se coordina la ejecución del conjunto de pruebas unitarias.

Por su parte, la Aplicación Agente tiene un esquema de hilos representado en la Figura 21.

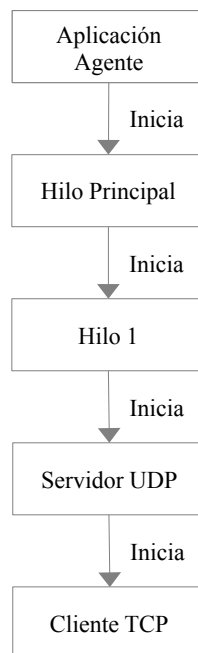


Figura 21: Esquema de proceso de la Aplicación Agente.

Fuente: Elaboración propia.

La Aplicación Agente realiza las siguientes acciones dentro de un hilo principal:

1. Se inicia el hilo 1, el cual a su vez inicia un socket UDP de tipo servidor. Este servidor UDP se mantiene en espera de que alguna Aplicación Coordinadora se conecte. Una vez que llegue la petición broadcast de la Aplicación Coordinadora, se procesa el contenido de la petición, que contiene los datos indicados anteriormente, por ejemplo, usuario, contraseña, ubicación del archivo de código fuente comprimido, etc.
2. Luego de obtener los datos de la petición UDP, se guarda la IP de la Aplicación Coordinadora para posteriormente poder identificarla y de este modo ignorar las peticiones UDP que la Aplicación Coordinadora realiza constantemente. Luego se inicia un socket TCP de tipo cliente y se envía un mensaje con el siguiente formato:

```
{  
  "msg"=>"get_source_code",  
  "hardware_specs"=>"AMD Ryzen 7 3700X 32GB"  
}
```

A través de la frase *get_source_code* se le indica a la Aplicación Coordinadora que se está listo para obtener el código fuente, y de este modo se da por establecida la comunicación entre las Aplicaciones Coordinadora y Agente.

4.2.8 Obtención y descompresión de código fuente

Una vez que la Aplicación Agente envía la frase *get_source_code*, la Aplicación Coordinadora registra la dirección IP de la Aplicación Agente, de este modo se evita que una Aplicación Agente intente conectarse más de una vez, seguidamente se responde a la Aplicación Agente con un mensaje con el siguiente formato:

```
{
  :msg=>"proceed_get_source_code"
}
```

Con este mensaje, la Aplicación Coordinadora le informa a la Aplicación Agente, que puede proceder con la obtención del código fuente comprimido. La Aplicación Agente utiliza las credenciales recibidas en el broadcast UDP y se conecta a través del protocolo SCP a la computadora donde se está ejecutando la Aplicación Coordinadora para obtener el código fuente. Es importante mencionar que la Aplicación Coordinadora no interviene en el proceso de obtención del código fuente comprimido.

Una vez obtenido el código fuente, se procede a descomprimirlo y luego se envía un mensaje a la Aplicación Coordinadora con el siguiente formato:

```
{
  :msg=>"get_tests_files"
}
```

De este modo se le está diciendo a la Aplicación Coordinadora que indique las pruebas a ejecutar.

La Aplicación Coordinadora también puede responder a la frase *get_source_code* con las frases *already_connected*, *no_exist_tests*, *finish_agent*; *already_connected* indica que la Aplicación Agente ya está conectada; *no_exist_tests*, indica que ya no hay pruebas que ejecutar; *finish_agent* le indica a la Aplicación Agente que debe terminar la conexión con la Aplicación Coordinadora. Para las tres frases indicadas anteriormente se procede a terminar la conexión.

4.2.9 Coordinación de la ejecución de pruebas unitarias

Una vez que la Aplicación Coordinadora recibe la frase *get_tests_files*, procede a recorrer el diccionario de pruebas unitarias de la sección 4.1.6 Agrupación de pruebas unitarias y obtiene el primer elemento con estado (*status*) *pending*.

Si no existe ningún elemento en estado *pending*, entonces, se obtiene el primer elemento con estado *sent*, con esto se logra acelerar el proceso de ejecución de pruebas, reenviando pruebas ya enviadas y cuyos resultados todavía no se recibieron. Reenviar pruebas es útil cuando una Aplicación Agente demora mucho en responder o nunca responde y otras Aplicaciones Agentes ya están libres. La demora puede darse por los siguientes motivos: La Aplicación Agente fue detenida, la computadora en donde se está ejecutando la Aplicación Agente fue apagada o la computadora no tiene suficiente capacidad de procesamiento. Eventualmente dos Aplicaciones Agentes pueden terminar ejecutando el mismo conjunto de pruebas, en este caso se procesarán los resultados de la primera Aplicación Agente que envíe sus resultados.

Una vez obtenido el elemento del diccionario se procede a:

- Cambiar el estado del elemento a *sent* si era *pending* o a *resent* si era *sent*.
- Establecer la IP de la Aplicación Agente en *agent_ip_address*.
- Establecer las especificaciones del hardware de la Aplicación Agente en *hardware_specs*. La información del hardware se imprimirá al terminar la ejecución de las pruebas y sirve para identificar las computadoras involucradas en el proceso, así también da una idea de la potencia de las computadoras indicando información sobre el procesador y la memoria.
- Establecer el tiempo que llevó el proceso de compartir el código fuente en *share_source_code*.

Una vez actualizado los datos del diccionario, se envía a la Aplicación Agente, un mensaje con el siguiente formato:

```
{
  :batch_num=>1,
  :tests_batch_keys=>[1, 2],
  :msg=>"process_tests"
}
```

En donde, *batch_num* es el número del conjunto enviado, *tests_batch_keys* es un arreglo con las claves del diccionario indicado en la sección 4.1.5 Obtención de pruebas unitarias. Cabe aclarar que no se le indica explícitamente a la Aplicación Agente que pruebas ejecutar, sólo se envían las claves.

4.2.10 Ejecución de pruebas unitarias

Cuando la Aplicación Agente recibe el mensaje con la frase *process_tests*, procede a ejecutar los archivos de pruebas indicados y guardar los resultados en un archivo.

Cuando la Aplicación Agente recibe el código fuente, procede a realizar el mismo proceso indicado en la sección 4.1.5 Obtención de pruebas unitarias, es decir, construye un diccionario enumerando todos los archivos del código fuente recibido, y como la Aplicación Coordinadora y la Aplicación Agente utilizan el mismo algoritmo para generar el diccionario con todas las pruebas, entonces, a través de las claves enviadas por la Aplicación Coordinadora se puede determinar cuál de los archivos de pruebas debe ejecutar.

Cuando ya no hay pruebas que ejecutar, se le envía a la Aplicación Agente la frase *no_exist_tests*, entonces la Aplicación Agente cierra la conexión.

4.2.11 Envío de resultados de pruebas unitarias

Una vez guardado los resultados de las pruebas en un archivo, a través del protocolo SCP se copia este archivo de resultados en la computadora en donde se está ejecutando la Aplicación Coordinadora, específicamente en la ruta indicada en *tests_results_folder_path*, el cual se recibió en la petición broadcast inicial.

Una vez terminada la copia de resultados, se envía a la Aplicación Coordinadora un mensaje con el siguiente formato:

```
{
  :msg=>"processed_tests",
  :batch_num=>1,
  :tests_result_file_name=>"batch_1_agent_192.168.0.7_tests_results"
}
```

De este modo se le informa a la Aplicación Coordinadora el nombre del archivo de resultados a procesar.

La Aplicación Coordinadora lee el archivo de resultados y comienza a actualizar el diccionario de pruebas unitarias de la sección 4.1.6 Agrupación de pruebas unitarias, para ello, identifica el elemento según el *batch_num* recibido y procede a actualizar:

- El estado a *processed*.
- El estado de los archivos a *processed* en *files_status*.
- La IP de la Aplicación Agente en *agent_ip_address*, esto es importante para las pruebas reenviadas, de este modo se sabe cuál Aplicación Agente ejecutó las pruebas.
- La cantidad de pruebas ejecutadas en *examples*.
- La cantidad de pruebas que pasaron en *passed*.
- La cantidad de pruebas fallaron en *failed*.
- La lista de archivos fallidos en *failed_files*.
- El tiempo que le tomó a la Aplicación Agente para cargar los archivos de pruebas en *files_load*.
- El tiempo que le tomó a la Aplicación Agente ejecutar los archivos de pruebas en *finish_in*.
- El tiempo que le tomó a la Aplicación Coordinadora obtener las pruebas a ejecutar, enviarlas a la Aplicación Agente y luego procesar los resultados en *batch_run*.
- El tiempo que le tomó a la Aplicación Coordinadora todo el proceso, desde la copia del código fuente hasta terminar la ejecución de un conjunto de pruebas en *tests_runtime*.
- Las especificaciones del hardware en *hardware_spec*, esto es importante para las pruebas reenviadas, de este modo se sabe cuál Aplicación Agente ejecutó las pruebas.

Una vez procesadas las pruebas, se repite el proceso desde la sección 4.1.9 Coordinación de la ejecución de pruebas unitarias hasta que ya no existan pruebas que ejecutar.

4.2.12 Muestra de resultados de pruebas unitarias

Una vez terminado el proceso de ejecución de pruebas unitarias, se procede a mostrar un resumen de los resultados obtenidos. La información mostrada es configurable a través del archivo de configuración **liri-config.yml**. Para más detalles del archivo de configuración, ver el Anexo B.

4.3 IMPLEMENTACIÓN DEL SISTEMA

4.3.1 Creación de la gema Liri usando Bundler

Para utilizar las funcionalidades que nos ofrece Ruby y sus gemas, se decidió empaquetar el sistema Liri en una gema a la cual también se nombró como Liri.

A continuación se procede a detallar los pasos para crear y configurar la gema Liri, según lo indicado en «Bundler: How to create a Ruby gem with Bundler» (2022):

Generar la estructura de la gema

```
liri@liri-virtualbox:~$ bundle gem liri
```

Al ejecutar el comando indicado se solicitará especificar:

- El framework de pruebas a utilizar.
- Confirmar el uso de la licencia MIT.
- Confirmar la inclusión de un código de conducta.

En la Figura 22 se muestra el resultado de la ejecución del comando anterior.

```
liri@liri-sys:~/discourse$ bundle gem liri
Creating gem 'liri'...
This means that any other developer or company will be legally allowed to use yo
ur code for free as long as they admit you created it. You can read more about
tur code for free as long as they admit you created it. You can read more about
the MIT license at https://choosealicense.cohe MIT license at
https://choosealicense.com/licenses/mit. y/(n): y
MIT License enabled in config
Do you want to include a code of conduct in gems you generate?
Codes of conduct can increase contributions to your project by contributors who
prefer collaborative, safe spaces. You can read more about the code of conduct
at contributor-covenant.org. Having a code of conduct means agreeing to the
responsibility of enforcing it, so be sure that you are prepared to do that. Be
sure that your email address is specified as a contact in the generated code of
conduct so that people know who to contact in case of a violation. For
suggestions about how to enforce codes of conduct, see https://bit.ly/coc-
enforcement. y/(n): y
Code of conduct enabled in config
create liri/Gemfile
create liri/lib/liri.rb
create liri/lib/liri/version.rb
create liri/liri.gemspec
create liri/Rakefile
create liri/README.md
create liri/bin/console
create liri/bin/setup
create liri/.gitignore
create liri/.travis.yml
create liri/.rspec
create liri/spec/spec_helper.rb
create liri/spec/liri_spec.rb
create liri/LICENSE.txt
create liri/CODE_OF_CONDUCT.md
Initializing git repo in /home/liri/Documentos/liri
Gem 'liri' was successfully created. For more information on making a RubyGem
visit https://bundler.io/guides/creating_gem.html
```

Figura 22: Ejecución del comando *bundle gem liri*.

Fuente: Elaboración propia.

Configurar el archivo *liri.gemspec*

Es necesario configurar este archivo para poder compilar la gema. Además, esta información será mostrada en RubyGems.org cuando se publique la gema. Las configuraciones necesarias son las siguientes:

- **spec.authors:** el nombre de los autores de la gema.
- **spec.email:** el correo electrónico de los autores de la gema.
- **spec.summary:** una breve descripción de la gema.

- **spec.description:** una descripción más completa de la gema.
- **spec.homepage:** la página web de la gema, en nuestro caso, como no tenemos página web, solo especificamos el repositorio en donde alojamos la gema “<https://github.com/rofaceess/tfg>”.
- **spec.metadata[“allowed_push_host”]:** se especifica “<https://rubygems.org>”, el cual es la url RubyGems.org en donde la gema será publicada para ser accesible a todo el mundo.
- **spec.metadata[“source_code_uri”]:** el repositorio donde tenemos alojado el código fuente de la gema, en nuestro caso el código está en “<https://github.com/rofaceess/tfg/liri>”.
- **spec.metadata[“changelog_uri”]:** es el log de cambios de la gema, aquí se registra información relevante sobre los cambios que se van agregando a la gema a medida que se lanzan nuevas versiones, en nuestro caso todavía no lanzamos versiones por lo que solo referenciamos a “<https://github.com/rofaceess/tfg/blob/master/liri/README.md>”, ésto para evitar que lanzará un error indicando que el archivo gemspec es inválido.

Agregar dependencias

En el archivo `liri.gemspec` se especifican las dependencias que serán necesarias al momento de ejecutar la gema. Estas dependencias también pueden agregarse en el archivo `Gemfile`, lo que nos permite ir desarrollando y probando la gema, pero, al momento de usar la gema fuera del contexto del desarrollo se necesita que estas dependencias estén configuradas en `liri.gemspec`, por lo que es mejor especificarlos en este archivo.

Las dependencias usadas por la gema Liri son las siguientes:

- **Rubyzip:** es una librería Ruby para lectura y escritura de archivos zip («Rubyzip», 2022). Lo empleamos para comprimir el código fuente de la aplicación Ruby on Rails cuyas pruebas se deben ejecutar.
- **Commander:** es una solución muy completa para escribir programas Ruby ejecutables en línea de comandos («Commander», 2022). Esta gema nos permite que la gema Liri sea ejecutable.

- **Highline:** es una librería diseñada para facilitar las tediosas tareas de entrada y salida en línea de comandos. Proporciona un sistema robusto para solicitar datos de un usuario («Highline», 2022). Con esta gema solicitamos la contraseña del usuario al momento de ejecutar la gema Liri.
- **Net-scp:** es una implementación del protocolo SCP en el lenguaje Ruby («Net-scp», 2022), la cual usamos para compartir el código fuente comprimido a las Aplicaciones Agentes del sistema Liri.
- **Chronic_duration:** es un sencillo analizador de lenguaje natural en Ruby para el tiempo transcurrido («Chronic Duration», 2022), que usamos para mostrar información sobre tiempos de ejecución del sistema Liri.
- **Tty-progressbar:** es una herramienta para mostrar una o varias barras de progreso en línea de comandos («TTY Progressbar», 2022), de este modo mostramos el progreso de ejecución del sistema.
- **Terminal-table:** es un generador de tablas rápido y sencillo, pero rico en funciones, escrito en Ruby y soporta tablas con formato ASCII y Unicode («Terminal Table», 2022), con esta herramienta mostramos tablas que resumen la ejecución del sistema Liri.

Configurar el archivo Gemfile

En este archivo configuramos las dependencias que utilizamos durante el desarrollo. Las dependencias definidas en este archivo son:

- **Rake:** es un programa similar al comando make presente en Linux, pero, implementado para Ruby («Rake», 2022). Esta gema nos permite automatizar ciertas tareas, como ser: compilar, instalar y publicar la gema. Las tareas se definen en el archivo Rakefile.
- **Rspec:** es una herramienta para realizar desarrollo guiado por comportamiento en Ruby («RSpec», 2022), que nos permite implementar y ejecutar pruebas unitarias de la gema Liri.

- **Rubocop:** es un analizador y formateador de código estático en Ruby, basado en la guía de estilo comunitaria de Ruby («Rubocop», 2022), que nos permite seguir ciertos estándares al momento de escribir el código de la gema Liri.
- **Yard:** es una herramienta de generación de documentación para el lenguaje de programación Ruby («Yard», 2022), la cual nos permite generar la documentación de la gema Liri con base en los comentarios que agregamos al código.

Instalar dependencias

Una vez configurado el gemspec y agregado las dependencias en el gemspec y el Gemfile, se procede a instalar las dependencias especificadas accediendo a la carpeta **liri** y ejecutando el siguiente comando:

```
liri@liri-sys:~liri$ bundle install
```

Compilar e instalar la gema

Para comprobar que la gema es funcional, primero se debe compilarla e instalarla a través de los siguientes comandos:

```
liri@liri-sys:~liri$ gem build liri.gemspec
liri@liri-sys:~liri$ gem install liri-0.1.0.gem
liri@liri-sys:~liri$ gem list liri
```

El último comando mostrará la gema ya instalada, aunque todavía no se podrá ejecutar (Más adelante, se explicará cómo hacer que Liri sea ejecutable).

Como el trabajo de compilar e instalar la gema suele ser repetitiva, definimos ciertas tareas rake para facilitar el proceso, no se entrará en mucho detalle al respecto, pero si se quiere saber más, en (Grice, 2018) se pueden encontrar ejemplos de cómo definir estas tareas, o bien, accediendo al archivo Rakefile del repositorio de la gema Liri en Github.

Hacer que la gema Liri sea ejecutable

En pasos anteriores, se había agregado la gema commander, esta gema ayuda a hacer que la gema Liri sea ejecutable en línea de comandos, para lograr esto, se debe crear una carpeta llamada **exe**, y dentro de esta carpeta crear un archivo llamado **liri**. En base a lo indicado en («Commander», 2022), el archivo debe tener el siguiente contenido:

```
#!/usr/bin/env ruby
require 'rubygems'
require 'commander/import'
require 'liri'

program :name, 'liri'
program :version, Liri::VERSION
program :description, 'Ejecuta pruebas unitarias usando un sistema distribuido'

command :manager do |c|
  c.action do |args, options|
    puts 'Ejecutando Manager'
  end
end
```

Una vez creado el archivo, se debe repetir el proceso de compilar e instalar la gema, para luego ejecutar el siguiente comando:

```
liri@liri-sys:~liri$ liri manager
```

Al ejecutar el comando anterior se imprime el texto: Ejecutando Manager.

Tener en cuenta que el contenido indicado es solo un ejemplo de cómo se configuró el ejecutable.

Actualmente, el ejecutable de la gema Liri soporta las siguientes formas de ejecución:

- **liri manager:** Ejecuta la Aplicación Coordinadora.
- **liri m:** Es un alias de liri manager.
- **liri agent:** Ejecuta la Aplicación Agente.
- **liri a:** Es un alias de liri agent.
- **liri tests_count:** Retorna la cantidad de archivos de pruebas de una aplicación.
- **liri help:** Detalla cómo utilizar el ejecutable de la gema Liri. En la Figura 23 se muestra un ejemplo de la ejecución del comando.


```
liri@liri-sys:~/discourse$ liri help
NAME:

  liri

DESCRIPTION:

  Ejecuta pruebas unitarias usando un sistema distribuido

COMMANDS:

  agent      Ejecuta el programa agente
  help       Display global or [command] help documentation
  manager    Ejecuta pruebas unitarias
  tests_count Retorna la cantidad total de tests
  tests_files Retorna los archivos de tests

ALIASES:

  a agent
  m manager
  tc tests_count
  tf tests_files

GLOBAL OPTIONS:

  -h, --help      Display help documentation
  -v, --version    Display version information
  -t, --trace      Display backtrace when an error occurs
```

Figura 23: Ejecución del comando *liri help*.

Fuente: Elaboración propia.

4.3.2 Instalador de la Aplicación Agente

Las Aplicaciones Agentes deben ejecutarse como servicio al momento de iniciar la computadora donde están alojadas. Para ello se debe utilizar el instalador que instala la Aplicación Agente y lo convierte en un servicio que se ejecuta en segundo plano.

El instalador de la Aplicación Agente consiste en una carpeta comprimida en formato zip con la estructura de la Figura 24.

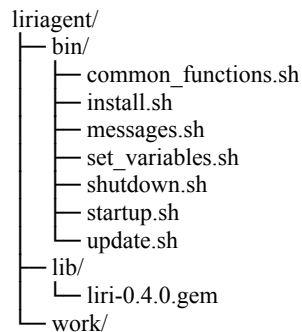


Figura 24: Estructura de carpetas del instalador del Agente Liri.

Fuente: Elaboración propia.

La carpeta bin contiene archivos con scripts en formato bash. El cometido de cada archivo es el siguiente:

- **common_functions.sh:** contiene funciones que se reutilizan en otros archivos, como son funciones para iniciar y detener el servicio de la Aplicación Agente, funciones para instalar RVM y Ruby, etc.
- **install.sh:** este es el script principal, el cual instala la Aplicación Agente y lo inicia como un servicio del sistema operativo Linux.
- **messages.sh:** contiene funciones para imprimir mensajes de error, advertencia, informativas, etc. Es utilizada dentro de los otros archivos bash.
- **set_variables.sh:** Define variables que son usadas en los otros archivos bash, como son, las rutas de las carpetas del instalador, la ubicación de RVM, la versión de la gema Liri, etc.
- **shutdown.sh:** Es el script que se ejecuta al momento de detener el servicio.
- **startup.sh:** Es el script que se ejecuta al momento de iniciar la Aplicación Agente como servicio del sistema operativo. Este script se encarga de configurar RVM, la versión de Ruby utilizada por la gema Liri y el gemset en donde se van a instalar los requerimientos de la gema Liri, luego inicia la Aplicación Agente usando el comando **liri a.**
- **update.sh:** Este script se utiliza para actualizar la versión de la gem Liri. Se usa cuando la gema Liri ya está instalada. Entonces este script detiene el servicio, instala la nueva versión de la gema y procede a iniciar de nuevo el servicio.

La carpeta **lib** contiene el paquete de la gema Liri, el script de instalación se encarga de instalar este paquete.

La carpeta **work** es utilizada por la Aplicación Agente para guardar la carpeta **liri** creada en la primera ejecución de la Aplicación. Esta carpeta guarda información relativa al proceso de ejecución tal y como se explicó en la sección 4.1.2 Inicio de configuraciones.

El ejecutar el script de instalación de la Aplicación Agente, se realizan las siguientes acciones:

- Se chequea los requerimientos y se muestran instrucciones indicando que programas son necesarios para finalizar con éxito la instalación. En el Anexo C, se detallan los requerimientos y el porqué de su necesidad para la distribución Manjaro.
- Se instala RVM, el cual es necesario para instalar Ruby.
- Se instala Ruby, el cual es necesario para instalar y ejecutar la gema Liri.
- Se configura el gemset para aislar las librerías utilizadas por la gema Liri.
- Se instala la gema Liri.
- Se crea, activa e inicia la Aplicación Agente como un servicio.

5 RESULTADOS

5.1 USO DEL SISTEMA

Como se indicó en la Figura 16, el sistema Liri está compuesto por una Aplicación Coordinadora y una Aplicación Agente. El ejecutable de la gema soporta el paso de parámetros para especificar cuál de las aplicaciones ejecutar.

En las siguientes secciones se explicará como instalar y usar el sistema, según el tipo de aplicación.

5.1.1 Uso de la Aplicación Coordinadora

Instalación de la aplicación

Para ejecutar la Aplicación Coordinadora, se debe instalar la gema Liri ejecutando el siguiente comando:

```
liri@liri-sys:~$ gem install liri
```

El comando anterior instala la gema desde el repositorio de RubyGems.org.

Ejecución de la aplicación

Luego de haber instalado la gema liri, se debe ubicar en la raíz del proyecto en la cual se estará trabajando y ejecutar el comando **liri m**.

Como ejemplo para la demostración y explicación del uso del sistema Liri, se utilizó la aplicación open source denominada **Discourse**. Vea el Anexo D para los detalles de instalación y configuración de **Discourse**. Usando **Discourse** como ejemplo de un proyecto hecho en Ruby on Rails, el programador debe ubicarse dentro de la carpeta raíz de la misma y ejecutar el comando de la siguiente manera:

```
liri@liri-sys:~discourse$ liri m
```

Inmediatamente después, la aplicación pedirá el ingreso de la contraseña del usuario del sistema operativo en la cual se está ejecutando la aplicación (Ver Figura 25).

Esta contraseña será utilizada por la Aplicación Agente para obtener el código fuente de la aplicación **Discourse**.

Figura 25: *Ejecución de la Aplicación Coordinadora.*
Fuente: Elaboración propia.

Luego, la Aplicación Coordinadora procede a comprimir el código fuente, mostrando en pantalla el tiempo que toma dicha compresión con una barra de progreso (Ver Figura 26).

Figura 26: *Proceso de compresión del código fuente.*
Fuente: Elaboración propia.

Una vez iniciada la comunicación entre ambas aplicaciones, las Aplicaciones Agentes proceden a obtener el código fuente del proyecto Ruby on Rails cuyos tests se desean ejecutar, alojado en la computadora Coordinadora.

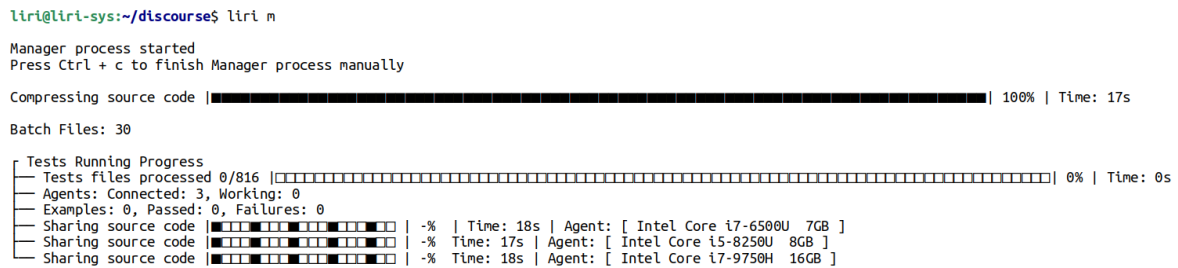


Figura 27: *Proceso de comunicación con las Aplicaciones Agentes.*

Fuente: Elaboración propia.

Como se puede observar en la Figura 27, mientras cada Aplicación Agente obtiene el código fuente del proyecto, en pantalla se muestra una barra de progreso en donde indica el tiempo que lleva a cada Aplicación Agente obtener dicho código fuente comprimido.

Una vez que obtienen todo el código fuente, cada Aplicación Agente procede a ejecutar las pruebas unitarias que le corresponda.

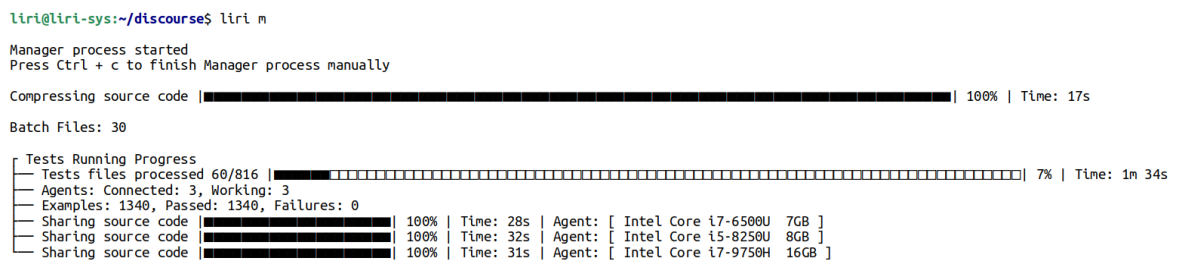


Figura 28: *Proceso de ejecución de las pruebas unitarias.*

Fuente: Elaboración propia.

En la Figura 28 se puede observar todas las variables que se muestran y actualizan durante el proceso de ejecución de la Aplicación Coordinadora.

Batch Files indica la cantidad de archivos de pruebas unitarias que estará ejecutando cada Aplicación Agente.

En **Test Files processed** se muestra en tiempo real cuántos archivos ya han sido procesados, como así también el porcentaje y tiempo de ejecución de las mismas, con una barra de progreso que se actualiza a la par que se va obteniendo los resultados de las Aplicaciones Agentes.

Agents Connected indica la cantidad de Aplicaciones Agentes conectadas. Cuando una Aplicación Agente se conecta, ya sea al principio o a la mitad de la ejecución, este parámetro se actualizará a +1.

Las Aplicaciones Agentes deben obtener el código fuente comprimido de la computadora en donde está ejecutándose la Aplicación Coordinadora para luego proceder con la descompresión de la misma y así comenzar con la ejecución de las pruebas unitarias, en éste punto es cuando **Agents Working** se incrementa +1, indicando la cantidad de Aplicaciones Agentes que iniciaron la ejecución de pruebas unitarias.

Examples muestra la cantidad total de pruebas unitarias ejecutadas, **Passed** la cantidad de pruebas que pasaron exitosamente y **Failures** la cantidad de pruebas que fallaron.

Sharing source code indica el tiempo que toma a cada Aplicación Agente obtener el código fuente comprimido.

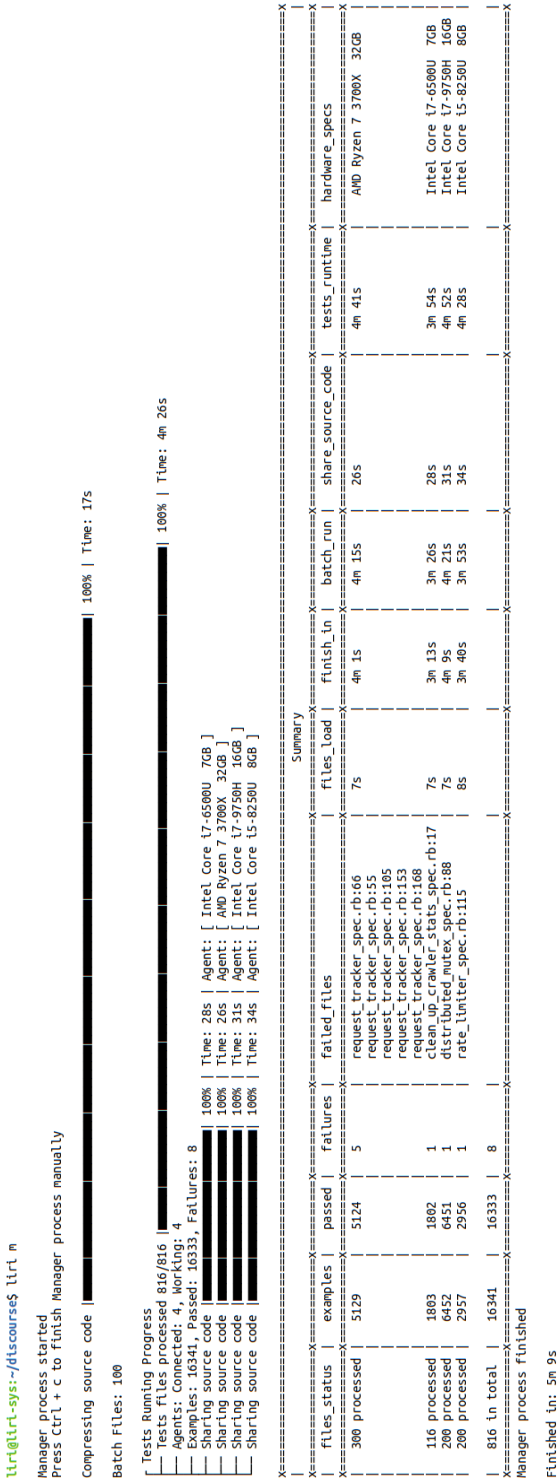


Figura 29: Resumen de ejecución del sistema Liri.

Fuente: Elaboración propia.

Luego de que las pruebas unitarias se hayan ejecutado en su totalidad, en la consola se desglosa una tabla resumida **Summary** (Ver Figura 29), en donde se presenta información relevante de cada una de las Aplicaciones Agentes que hayan trabajado con la Aplicación Coordinadora. La tabla tiene las siguientes columnas:

file_status: Cantidad de archivos por Agente con sus respectivos estados, que pueden ser: *pending* (pendiente), *sent* (enviado), *resent* (reenviado) y *processed* (procesado).

examples: Cantidad de pruebas unitarias ejecutadas por cada Aplicación Agente.

passed: Cantidad de pruebas unitarias que pasaron.

failures: Cantidad de pruebas unitarias que fallaron.

failed_files: Nombres de los archivos que contienen pruebas que fallaron, además especifica la línea de la prueba que falla.

files_load: Indica el tiempo que tardó RSpec en cargar los archivos de pruebas unitarias, antes de proseguir con la ejecución de las mismas.

finish_in: Indica el tiempo que tardó cada Aplicación Agente en ejecutar todas las pruebas.

batch_run: Incluye una suma de tiempos, es decir, muestra la sumatoria del tiempo que pasó desde que la Aplicación Coordinadora obtiene las pruebas a ejecutar, las envía a cada Aplicación Agente y finalmente recibe los resultados.

share_source_code: tiempo que toma a cada Aplicación Agente obtener de la computadora en donde está ejecutándose la Aplicación Coordinadora el código fuente comprimido

tests_runtime: Es la suma de *share_source_code* y *batch_run*.

hardware_specs: Detalla dos características relevantes de cada Aplicación Agente, las cuales son: tipo de procesador y memoria RAM.

Debajo de la tabla Summary, se puede observar el texto **Finalizado en:** en donde se indica el tiempo total que tardó el sistema Liri en ejecutar todas las pruebas unitarias.

5.1.2 Uso de la Aplicación Agente

Instalación de la aplicación

La Aplicación Agente se puede instalar de dos formas, la primera de ellas es instalando la gema Liri, ejecutando el siguiente comando:

```
liri@liri-sys:~$ gem install liri
```

Esto instala la gema desde el repositorio de RubyGems.org.

Instalar la aplicación de este modo no es lo más conveniente porque se necesitaría iniciar manualmente la Aplicación Agente cada vez que se inicia la computadora en donde está alojada.

La otra forma de instalar la Aplicación Agente es usando el instalador que se encuentra alojado en el repositorio de Github de la gema Liri, cuya dirección es: <https://github.com/roface/liri>. Instalando de esta forma, la Aplicación Agente se ejecutará como un servicio del sistema operativo.

En el Anexo C, se explica cómo instalar la Aplicación Agente, usando el instalador alojado en el repositorio Github. Si bien, el ejemplo dado es para instalar en la distribución Manjaro, cabe aclarar que el instalador también se ejecutó y se comprobó su funcionamiento exitoso en las siguientes distribuciones Linux: Debian, Ubuntu y Fedora.

Ejecución de la aplicación

En caso de haber instalado Liri de la primera forma, es decir, descargando la gema, se debe abrir la consola de línea de comando y ejecutar el siguiente comando:

```
liri@liri-sys:~$ liri a
```

En la Figura 30 se muestra un ejemplo de ejecución de la Aplicación Agente.

```
liri@liri-sys:~/discourse$ liri a
Agent process started
Press Ctrl + c to finish Agent process manually
```

Figura 30: Ejecución de la Aplicación Agente.

Fuente: Elaboración propia.

En caso de haber utilizado el instalador alojado en Github, basta con encender la computadora/as en donde está instalada, y la Aplicación Agente se iniciará automáticamente como un servicio del sistema operativo.

En la primera ejecución del programa, se crea la carpeta llamada *liri*, donde se guarda la estructura de carpetas y archivos relativos a la ejecución de la Aplicación Agente tal y como se indicó en la sección 4.1.2 Inicio de configuraciones. Si la Aplicación Agente fue ejecutada usando el comando ***liri a***, la carpeta se creará en la misma ruta en donde se ejecutó el comando, pero si fue iniciado como un servicio, entonces la carpeta se creará dentro de la carpeta *work* del instalador de la Aplicación Agente.

5.2 EVALUACIÓN DEL SISTEMA IMPLEMENTADO

5.2.1 Entornos de pruebas del sistema implementado

Para la evaluación y validación del sistema implementado se utilizó un proyecto Open Source hecho en Ruby on Rails denominado **Discourse**, el cual posee **16.341** pruebas unitarias, distribuidas en un total de **816** archivos. Vea el Anexo D para los detalles de instalación y configuración de **Discourse**.

El sistema implementado fue probado en una red local inalámbrica y las características de cada computadora que formaron parte de las pruebas, están especificadas en la Tabla 5. Cada una de las computadoras posee características diferentes y se listan de menor a mayor potencia.

	HP	ASUS	MSI	AORUS
Mem.	7GB	8GB	16GB	32GB
Proces.	Intel Core i7-6500U 2.50GHz 2 cores 4 threads	Intel Core i5-8250U 1.60GHz 4 cores 8 threads	Intel Core i7-9750H 2.95GHz 6 cores 12 threads	AMD Ryzen 7 3700X 8 cores 16 threads
Almac.	HDD 1 TB	Western Digital SSD M2 500G 3D NAND	Samsung SSD 512GB 3D TLC NAND	Samsung SSD 980 PRO 500GB
SO	Ubuntu 20.04 Focal Fossa	Manjaro Linux 21.2.6 Qonos	Manjaro Linux 21.2.6 Qonos	Manjaro Linux 21.2.6 Qonos

Tabla 5: Especificaciones de las computadoras utilizadas en las pruebas del sistema Liri.

Fuente: Elaboración propia.

Para realizar las pruebas del sistema Liri, se definieron varios entornos de pruebas. En la Tabla 6 se puede observar las computadoras involucradas en cada uno de los entornos agrupados por sus roles, que pueden ser Computadora Coordinadora y/o Computadora Agente.

Entorno de Prueba	Computadora Coordinadora	Computadoras Agente
A	HP	HP
B	HP	HP ASUS
C	HP	HP ASUS MSI
D	HP	HP ASUS MSI AORUS

Tabla 6: *Entornos utilizados para las pruebas del sistema Liri.*

Fuente: Elaboración propia.

5.2.2 Formato de registro de resultados

Para realizar las pruebas del sistema Liri, se tomaron en cuenta las variables indicadas en la Tabla 7.

Variable	Descripción
Entorno de prueba	Es el entorno de prueba utilizado según lo especificado en la Tabla 6.
Tamaño del conjunto de archivos	Es el tamaño del conjunto de archivos que serán ejecutados por cada Agente. El rango de tamaño elegible en el proyecto Discourse es muy amplio, por este motivo se seleccionaron 3 tamaños diferentes para realizar las pruebas, las cuales son: 30, 50 y 100, de este modo se tiene más variedad en los resultados, y así también se limita la cantidad de pruebas a realizar.

Tabla 7: *Variables consideradas en las pruebas del sistema Liri.*

Fuente: Elaboración propia.

Para registrar los resultados de las pruebas del sistema Liri, se elaboró una planilla como la que se puede observar en la Tabla 8, luego se resume estos resultados en la planilla de la Tabla 9.

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: _____		Aplicación: _____		
Entorno de prueba: _____		Cantidad de pruebas unitarias: _____		
Tamaño de Conjunto: _____		Cantidad de archivos: _____		
Tiempo de ejecución: _____				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas

Tabla 8: Planilla detallada para el registro de resultados de la ejecución del sistema Liri.

Fuente: Elaboración propia.

PLANILLA RESUMIDA DE RESULTADOS							
Aplicación: _____			Cantidad de pruebas unitarias: _____		Cantidad de archivos: _____		
VARIABLES			RESULTADOS				
N.º de prueba	Entorno de prueba	Tamaño de Conjunto	Tiempo de ejecución	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas

Tabla 9: Planilla resumida para el registro de resultados de la ejecución del sistema Liri.

Fuente: Elaboración propia.

En el Anexo E se presenta el registro de todas las pruebas realizadas al sistema Liri. En el Anexo F se presentan los resultados de pruebas realizadas sobre otra aplicación diferente a **Discourse**.

5.2.3 Ejecución de las pruebas unitarias con el método convencional

Inicialmente se realizó la ejecución de pruebas unitarias de la aplicación **Discourse** sin el sistema Liri. Los tiempos que le llevó a cada una de las computadoras ejecutar dichas pruebas se puede visualizar en la Tabla 10.

Computadora	Tiempo de ejecución
HP	22m 41s
ASUS	24m 30s
MSI	13m 35s
AORUS	12m 33s

Tabla 10: Tiempo de ejecución de las pruebas por computadora sin Liri.

Fuente: Elaboración propia.

Como se puede observar en la Tabla 10 la computadora más rápida en ejecutar todas las pruebas fue AORUS y la que demoró más fue la HP. Observando las características de ambas computadoras en la Tabla 5, se visualiza que la computadora AORUS cuenta con mayor memoria y mejor procesador, por ende es más potente que la computadora HP.

5.2.4 Ejecución de las pruebas unitarias utilizando el sistema implementado

Para analizar los datos obtenidos en cada ejecución de Liri, se dividieron en 2 puntos:

- Tiempo de ejecución de pruebas unitarias por Agente.
- Cantidad de pruebas unitarias ejecutadas por Agente.

Tiempo de ejecución de pruebas unitarias por Agente

En la Tabla 11 se detalla los tiempos que le tomó a Liri ejecutar todas las pruebas según el entorno de ejecución y tamaño de conjunto de archivos.

En cada entorno se ejecutaron **16.341** pruebas, de las cuales **16.333** pasaron y **8** fallaron.

	Entorno A 1 Agente	Entorno B 2 Agentes	Entorno C 3 Agentes	Entorno D 4 Agentes
Conjunto de 30 archivos	26m 10s	13m 33s	08m 00s	05m 49s
Conjunto de 50 archivos	24m 12s	13m 14s	07m 32s	05m 24s
Conjunto de 100 archivos	22m 57s	12m 42s	07m 41s	05m 09s

Tabla 11: *Tiempos de ejecución según conjunto de archivos y entornos de pruebas.*

Fuente: Elaboración propia.

En base a los datos obtenidos de la Tabla 11 se pudo elaborar el gráfico comparativo de la Figura 31.

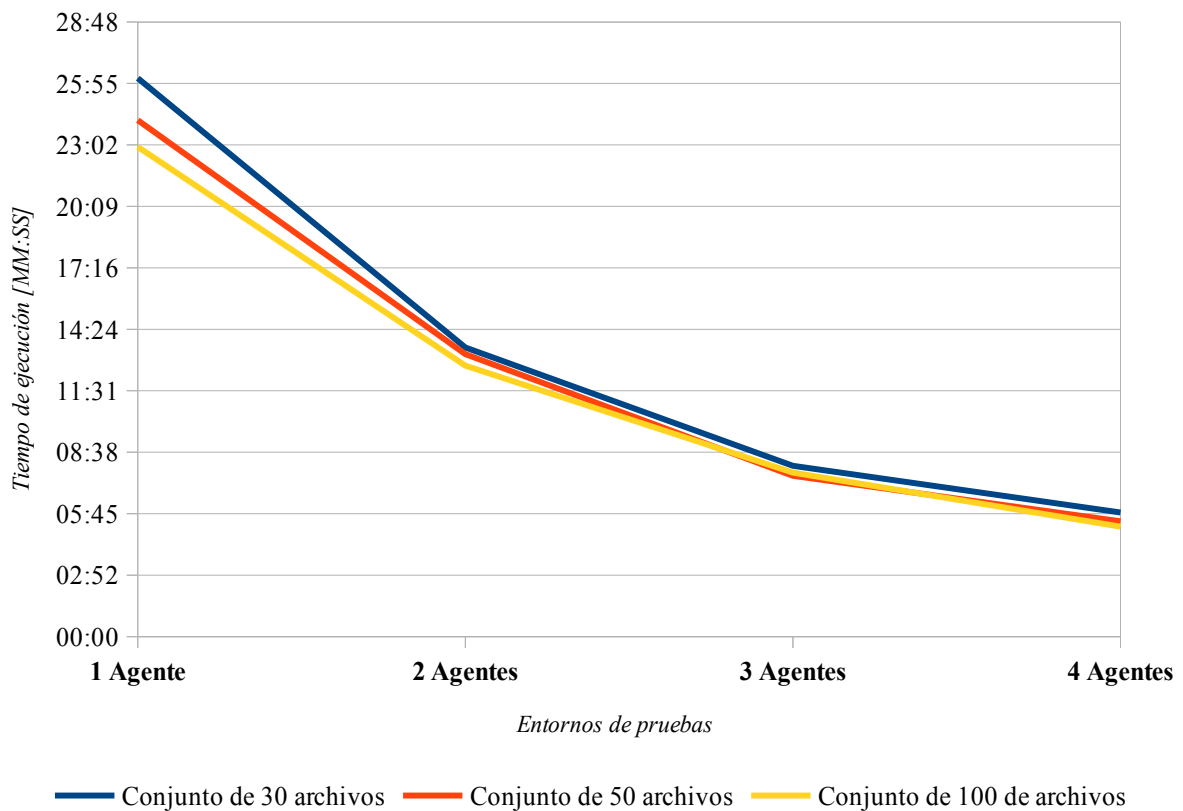


Figura 31: Comparativa de tiempos de ejecución según conjunto de archivos y entornos de pruebas.

Fuente: Elaboración propia.

Como resultado se puede apreciar que cuanto más agentes se encuentran trabajando para la Aplicación Coordinadora, menos tiempo lleva a Liri ejecutar las pruebas unitarias. Con 2 Agentes se ve optimizado el tiempo de ejecución hasta más de 2 veces con respecto a lo que demoró Liri en ejecutar las pruebas unitarias con 1 Agente, y con 3 y 4 Agentes se redujo hasta más de 3 veces.

En cambio, como se puede observar en la Figura 31, a partir de 2 Agentes la variación del tamaño de los conjuntos no produce un impacto significativo en el tiempo de ejecución, considerando esto, se pudo elaborar una fórmula para calcular los tamaños recomendables para los conjuntos. Esta fórmula se presenta en el Anexo G.

Cantidad de pruebas unitarias ejecutadas por Agente

Al tener varias computadoras con diferentes características hardware, siempre habrá una variación en la cantidad de pruebas unitarias ejecutadas por cada computadora. Para demostrar esta variación se seleccionó y analizó el entorno de prueba D que contiene 4 computadoras Agente.

En las Tablas 12, 13 y 14 se indican las cantidades de pruebas unitarias ejecutadas por cada computadora según el tamaño de conjunto de archivos.

Computadora	Cantidad de pruebas ejecutadas
HP	2.408
ASUS	2.891
MSI	4.166
AORUS	6.876

Tabla 12: Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 30 archivos.

Fuente: Elaboración propia.

Computadora	Cantidad de pruebas ejecutadas
HP	2.748
ASUS	2.640
MSI	4.732
AORUS	6.221

Tabla 13: Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 50 archivos.

Fuente: Elaboración propia.

Computadora	Cantidad de pruebas ejecutadas
HP	1.803
ASUS	2.957
MSI	6.452
AORUS	5.129

Tabla 14: Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 100 archivos.

Fuente: Elaboración propia.

De acuerdo a los datos obtenidos de las Tablas 12, 13 y 14 se pudo elaborar los gráficos comparativos de las Figuras 32, 33 y 34.

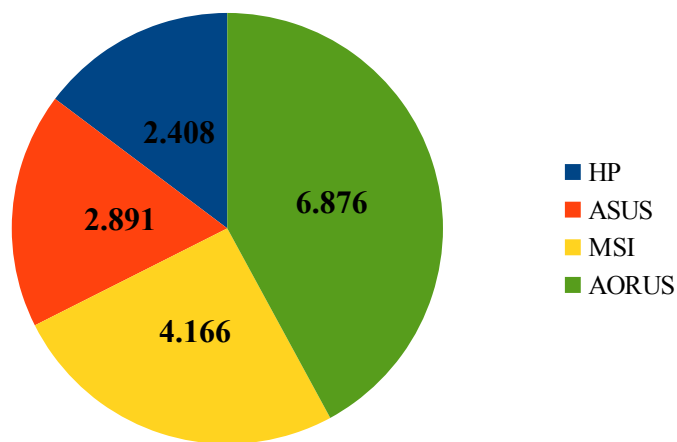


Figura 32: Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 30 archivos.
Fuente: Elaboración propia.

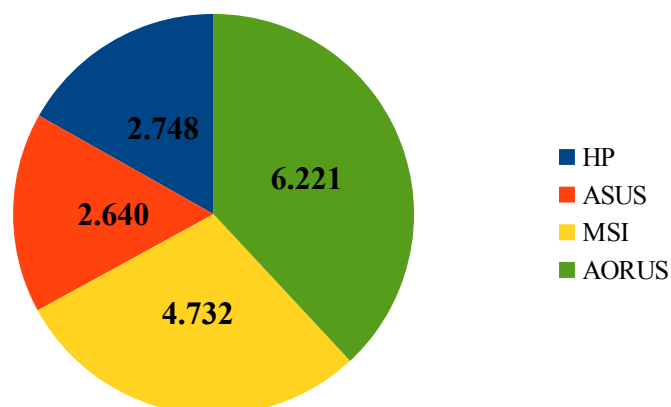


Figura 33: Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 50 archivos.
Fuente: Elaboración propia.

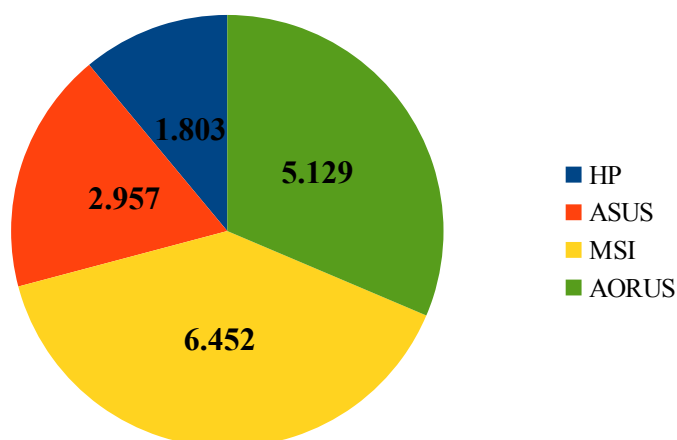


Figura 34: Cantidad de pruebas ejecutadas por cada Agente, con un conjunto de 100 archivos.
Fuente: Elaboración propia.

Como se puede observar en los gráficos, las computadoras con más potencia ejecutaron mayor cantidad de pruebas unitarias, pero esto no siempre es así, como ejemplo de esto último se puede observar la Figura 34, donde MSI ejecutó mayor cantidad de pruebas en comparación a AORUS, esto se debe a que la cantidad y complejidad de pruebas unitarias que contiene cada archivo son diferentes, y de forma aleatoria se puede asignar a cualquier computadora la ejecución de archivos con mayor cantidad de pruebas y/o más complejas.

5.2.5 Comparativa de tiempos entre el método convencional y el sistema implementado

De acuerdo a los datos obtenidos de la Tabla 11 y el tiempo de ejecución de la computadora HP indicado en la Tabla 10, se pudo elaborar la Tabla 15 en la que se puede visualizar el tiempo (en segundos) que tomó a la computadora HP ejecutar las pruebas unitarias con el método convencional, junto con los tiempos de ejecución (en segundos) utilizando el sistema Liri con diferentes entornos y tamaño de conjunto de archivos.

	Convencional HP	Entorno A 1 Agente	Entorno B 2 Agentes	Entorno C 3 Agentes	Entorno D 4 Agentes
Conjunto de 30 archivos	1.361	1.570	813	480	349
Conjunto de 50 archivos	1.361	1.452	794	452	324
Conjunto de 100 archivos	1.361	1.377	762	461	309

Tabla 15: Tiempo de ejecución de pruebas unitarias utilizando el método convencional y Liri, con sus diferentes entornos y tamaño de conjunto de archivos.

Fuente: Elaboración propia.

Teniendo en cuenta los datos de la Tabla 15, se pudo calcular el porcentaje de optimización del tiempo de ejecución de las pruebas unitarias utilizando el sistema Liri con los diferentes entornos, con respecto al tiempo que tomó a HP ejecutar las mismas pruebas con el método convencional.

Para el cálculo de porcentaje se utilizó la siguiente fórmula:

$$\text{Optim. \%} = \left(1 - \frac{\text{Tiempo de ejecución utilizando Liri con el entorno } x \text{ (seg)}}{\text{Tiempo de ejecución método convencional (seg)}}\right) * 100 \quad (5.1)$$

Aplicando la fórmula anterior sobre los datos de la Tabla 15 se obtuvieron los valores indicados en la Tabla 16.

	Entorno A 1 Agente	Entorno B 2 Agentes	Entorno C 3 Agentes	Entorno D 4 Agentes
Conjunto de 30 archivos	-15,3 %	40,2 %	64,7 %	74,3 %
Conjunto de 50 archivos	-6,6 %	41,6 %	66,7 %	76,1 %
Conjunto de 100 archivos	-1,1 %	44,0 %	66,1 %	77,2 %

Tabla 16: Porcentaje de optimización en la ejecución de pruebas utilizando el sistema Liri.

Fuente: Elaboración propia.

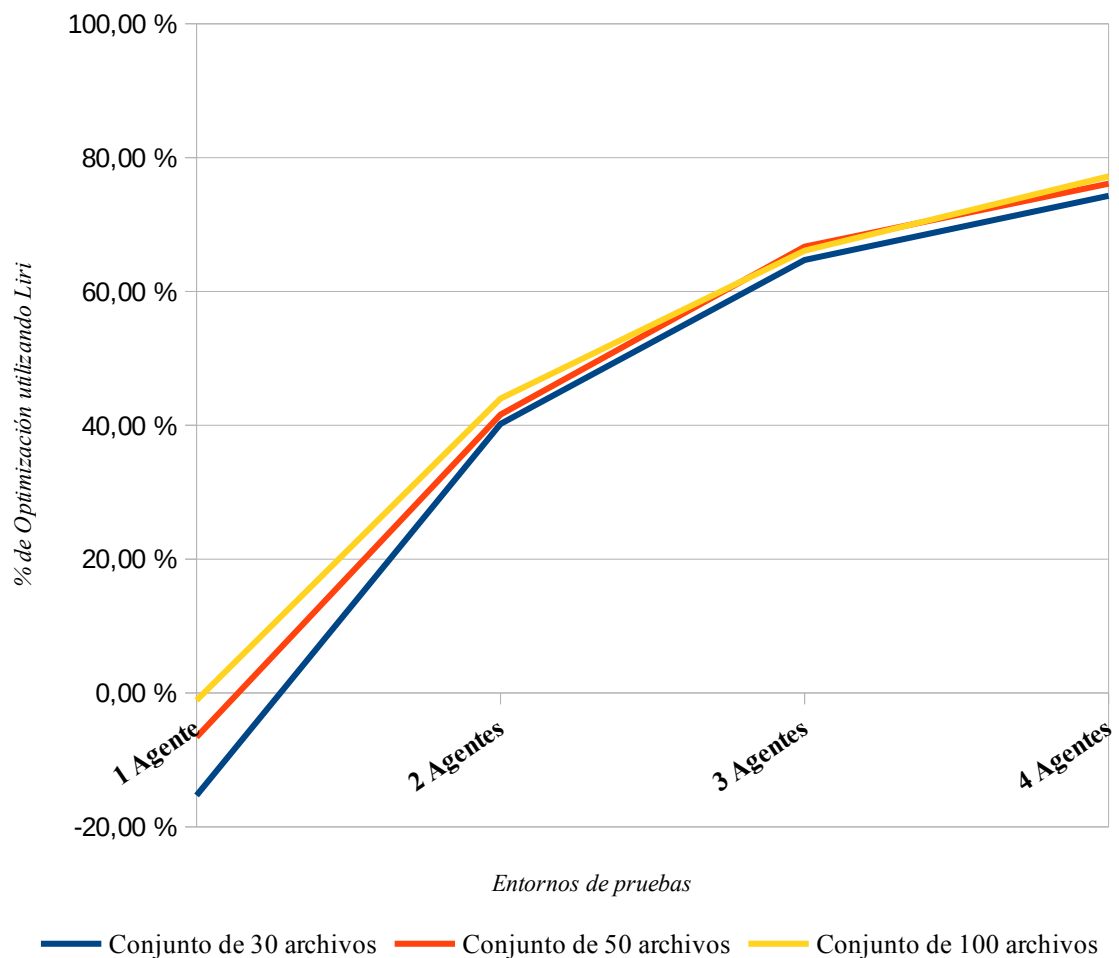


Figura 35: Porcentaje de optimización en la ejecución de pruebas utilizando el sistema Liri.
Fuente: Elaboración propia.

Como se puede observar en la Figura 35, con 1 Agente no hubo mejora utilizando el sistema Liri, esto se debe a que Liri realiza varios trabajos extras con respecto al método convencional para poder ejecutar de forma distribuida las pruebas, como lo son: la compresión y descompresión del código fuente, la conexión que realiza la Aplicación Coordinadora con la Aplicación Agente, la obtención del código fuente, etc.

También se puede contemplar que con tan sólo 2 Agentes ya se presentan mejoras en el tiempo de ejecución hasta un poco más del 40% con respecto al tiempo que tomó ejecutar dichas pruebas con el método convencional. Con 3 y 4 Agentes, el porcentaje de optimización crece hasta más de un 70%.

Con ésto podemos concluir que Liri a partir de 2 Agentes, mejora de manera satisfactoria y óptima el tiempo de ejecución de las pruebas unitarias, mejorando aún más el tiempo de ejecución con una mayor cantidad de Agentes.

5.2.1 Validación de resultados

En la Tabla 17 se presenta una planilla donde se validan los resultados obtenidos.

VALIDACIÓN DE RESULTADOS			
Aplicación: Discourse		Cantidad de pruebas unitarias: 16.341	Cantidad de archivos: 816
			Si No
	Instalación y uso del sistema		
1	La Aplicación Coordinadora es fácil de instalar.	X	
2	La Aplicación Agente es fácil de instalar.	X	
3	El sistema es fácil de usar.	X	
4	El sistema es configurable.	X	
5	El sistema puede ser usado desde una línea de comandos.	X	
6	El sistema ejecuta las pruebas unitarias del código fuente sobre la que el desarrollador está trabajando localmente.	X	
7	El sistema es capaz de ejecutar pruebas unitarias de más de una aplicación.	X	
	Durante la ejecución del sistema		
8	Se muestra un contador de tiempo del proceso de compresión de código fuente.	X	
9	Se muestra un contador de tiempo del proceso de copia de código fuente en las computadoras donde se están ejecutando las Aplicaciones Agentes.	X	
10	Se muestra un contador de tiempo del proceso de ejecución de pruebas unitarias.	X	
11	Se muestran las Aplicaciones Agentes conectadas y trabajando.	X	
12	La Aplicación Coordinadora sondea constantemente la red en busca de nuevos Agentes.	X	
13	La Aplicación Coordinadora permite la conexión con nuevos Agentes en medio de una ejecución previamente iniciada.	X	
	En cada ejecución exitosa del sistema Liri para una misma aplicación		
14	Siempre se procesan todos los archivos de pruebas.	X	
15	Siempre se ejecuta la misma cantidad de pruebas.	X	
16	Siempre pasan la misma cantidad de pruebas cuando no hubieron cambios en la aplicación.	X	
17	Siempre fallan la misma cantidad de pruebas cuando no hubieron cambios en la aplicación.	X	
18	Siempre fallan las mismas pruebas.	X	
19	Cada conjunto de archivos de pruebas contiene archivos aleatorios.	X	
20	Los tiempos de ejecución son similares para un mismo entorno de prueba.	X	
21	Se detectan nuevas pruebas agregadas.	X	

	Resultados		
22	Se muestra la cantidad de archivos procesados por cada Agente.	X	
23	Se muestra la cantidad de pruebas unitarias.	X	
24	Se muestra la cantidad de pruebas que pasaron.	X	
25	Se muestra la cantidad de pruebas que fallaron.	X	
26	Se muestran la lista de pruebas que fallaron.	X	
27	Se muestra información de las Aplicaciones Agentes.	X	
28	Se muestra el tiempo total de ejecución del sistema.	X	

Tabla 17: *Planilla de validación del sistema Liri.*

Fuente: Elaboración propia.

6 CONCLUSIÓN

El presente Trabajo Final de Grado tuvo como objetivo desarrollar un sistema que permita reducir el tiempo de ejecución de pruebas unitarias de aplicaciones Ruby on Rails utilizando la librería Rspec.

En base a la arquitectura propuesta, se implementó un sistema compuesto por una Aplicación Coordinadora y una Aplicación Agente. El sistema implementado se empaqueta en una gema Ruby, lo cual facilita su instalación y uso. También se desarrolló un instalador para la Aplicación Agente, el cual permite su instalación como un servicio del sistema operativo.

Para utilizar el sistema, se debe tener instalado y ejecutando uno o más Agentes en una red de computadora, para luego ejecutar el comando que corresponde a la Aplicación Coordinadora en la ruta de una aplicación Ruby on Rails. Una vez iniciada la Aplicación Coordinadora, ésta se conectará automáticamente con las Aplicaciones Agentes en ejecución dentro de la red e iniciará el proceso de ejecución de pruebas unitarias.

Como se pudo observar en las evaluaciones realizadas del sistema Liri sobre la aplicación open source llamada Discourse, se logró demostrar que a partir de 2 Agentes ya se presentan mejoras en el tiempo de ejecución hasta un poco más del 40% con respecto al tiempo que tomó ejecutar dichas pruebas con el método convencional. Con 3 y 4 Agentes, el porcentaje de optimización crece hasta más de un 70% y se intuye que el tiempo de ejecución irá reduciéndose a medida que se aumenta la cantidad de Agentes, por lo tanto, se concluye que se cumple satisfactoriamente con los objetivos del presente Trabajo Final de Grado logrando ayudar al desarrollador a reducir de manera eficiente el tiempo que conlleva la ejecución de pruebas unitarias.

7 LÍNEAS DE INVESTIGACIÓN FUTURA

Para mejorar el sistema implementado se proponen las siguientes mejoras:

- **Agregar contraseña al código fuente comprimido.** El archivo comprimido compartido entre las máquinas agentes debería estar protegido con una contraseña, para evitar que terceros puedan acceder al código fuente en caso de obtener acceso al archivo comprimido.
- **Codificar el usuario y contraseña guardados en el archivo `liri-credentials.yml`.** Actualmente, el usuario y la contraseña se guardan como texto plano. Esta información debería codificarse para evitar que terceros puedan acceder a estos datos en caso de obtener acceso al archivo indicado.
- **Codificar la información de usuario y contraseña enviados a las Aplicaciones Agentes.** Actualmente, estos datos se envían como texto plano, y, será más seguro enviar esta información codificada y que las Aplicaciones Agentes lo decodifiquen únicamente cuando lo vayan a utilizar.
- **Ejecución de pruebas en paralelo.** Las Aplicaciones Agentes deberían poder ejecutar varias pruebas en paralelo acorde a alguna configuración que indique en cuántos hilos se requiere ejecutar las pruebas. Si se le indica una Aplicación Agente que ejecute 50 archivos de pruebas unitarias, se podría ejecutar 25 archivos en un hilo y los otros 25 archivos en otro hilo, de este modo, teóricamente la velocidad de ejecución aumentaría.
- **Ejecución de pruebas de múltiples aplicaciones en paralelo.** Si un desarrollador X y un desarrollador Y, inician la ejecución de pruebas unitarias al mismo tiempo, de una misma aplicación, que utilicen información compartida, como ser caché o base de datos, o de diferentes aplicaciones que no compartan información, la ejecución de estas pruebas unitarias van a interferir entre sí. Esto se debe a que la Aplicación Agente no se implementó de tal manera a soportar la ejecución de pruebas unitarias de varias aplicaciones al mismo tiempo. En una aplicación real del sistema, es necesario poder ejecutar las pruebas de varias aplicaciones en paralelo.

- **Distribuir los archivos de pruebas unitarias a ejecutar de manera inteligente.** En caso de que la potencia de las computadoras involucradas en la ejecución de pruebas sea variada, sería útil llevar un registro de los tiempos que la lleva a cada computadora ejecutar cierto conjunto de archivos de prueba. De este modo, a medida que se va recabando información, se podrían enviar los archivos de pruebas que toman más tiempo a las computadoras que puedan ejecutar estas pruebas en menos tiempo, con el objetivo de acelerar más el tiempo de ejecución. Actualmente, los archivos de pruebas se comparten a las Aplicaciones Agentes de forma totalmente aleatoria.
- **Identificar cada prueba unitaria.** Actualmente, se indica a las Aplicaciones Agentes que conjunto de archivos de pruebas unitarias deben ejecutar. Para balancear mejor la distribución de pruebas unitarias ejecutadas por cada Aplicación Agente, sería mejor indicarles el conjunto de pruebas unitarias a ejecutar en vez del conjunto de archivos de pruebas a ejecutar. Eventualmente, un archivo podría contener pocas pruebas y otro archivo tener el doble, triple o más pruebas. Esta situación causaría un impacto negativo en el tiempo de ejecución cuando una Aplicación Agente recibe archivos de pruebas a ejecutar que le tomen demasiado tiempo, en comparación al tiempo que le toma a otras Aplicaciones Agentes el ejecutar otros conjuntos de archivos, y el resultado sería peor si la computadora que recibe la mayor carga de trabajo resulta ser la menos potente.
- **Comprobar la fórmula para el cálculo del tamaño de conjuntos:** Realizar pruebas más exhaustivas para comprobar la efectividad de la fórmula presentada en el Anexo G.

REFERENCIA BIBLIOGRÁFICA

- ¿Qué es el kernel de Linux? (2019). Recuperado 24 de abril de 2022, de <https://www.redhat.com/es/topics/linux/what-is-the-linux-kernel>
- Acerca de Ruby. (s. f.). Recuperado 21 de abril de 2022, de <https://www.ruby-lang.org/es/about/#fn1>
- Anwer, M. (2020). Top 5 Open Source Version Control Systems In 2021. Recuperado 28 de abril de 2022, de Containerize website: <https://blog.containerize.com/2020/12/11/top-5-open-source-version-control-systems-in-2021/>
- Athanasiou, D., Nugroho, A., Visser, J., & Zaidman, A. (2014). Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering*, 40(11), 1100-1125. <https://doi.org/10.1109/TSE.2014.2342227>
- Atlassian. (2022). Qué es Git. Recuperado 2 de mayo de 2022, de <https://www.atlassian.com/es/git/tutorials/what-is-git>
- Atlassian Corporation. (2021). Qué es el control de versiones. Recuperado 26 de abril de 2022, de Atlassian Corporation website: <https://www.atlassian.com/es/git/tutorials/what-is-version-control>
- Bertolino, A. (2001). CHAPTER 5 SOFTWARE TESTING. En A. Abran & J. W. Moore (Eds.), *Guide to the Software Engineering Body of Knowledge SWEBOK* (pp. 69-86). Los Alamitos, California: IEEE Computer Society.
- Bertolino, A. (2007). Software Testing Research: Achievements, Challenges, Dreams. En IEEE (Ed.), *Future of Software Engineering (FOSE '07)* (pp. 85-103). Minneapolis, MN, USA: IEEE. <https://doi.org/10.1109/FOSE.2007.25>.
- Berzal, F. (2004). *El ciclo de vida de un sistema de información* (p. 42). p. 42. Departamento de Ciencias de la Computación e Inteligencia Artificial, Universidad de Granada. Recuperado de <https://elvex.ugr.es/idbis/db/docs/lifecycle.pdf>
- Black, R., Coleman, G., Walsh, M., Cornanguer, B., Forgács, I., Kakkonen, K., & Sabak, J. (2017). *Agile Testings Foundations: An ISTQB Foundation Level Agile Tester guide* (T. C. I. for I. BCS, Ed.). Swindon, Inglaterra.
- Blondeau, V., Etien, A., Anquetil, N., Cresson, S., Croisy, P., Stéphane, D., & Ducasse, S. (2017). What are the Testing Habits of Developers? A Case Study in a Large IT Company. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 58-68. Shanghai, China: IEEE. <https://doi.org/10.1109/ICSME.2017.68>

- Blum, R., & Bresnahan, C. (2015). *Linux Command Line and Shell Scripting Bible*. (3^a ed.). John Wiley & Sons.
- Bundler: How to create a Ruby gem with Bundler. (2022). Recuperado 4 de abril de 2022, de https://bundler.io/v2.0/guides/creating_gem.html
- Bustos, G. (2021). ¿Qué Es GitHub Y Cómo Utilizarlo? Recuperado 30 de abril de 2022, de Tutoriales Hostinger website: <https://www.hostinger.es/tutoriales/que-es-github>
- Cataldi, Z., Lage, F., Pessacq, R., & García Martínez, R. (1999). Ingeniería de software educativo. En D. de P. de la F. de I. UBA (Ed.), *V Congreso Internacional de Ingeniería Informática* (pp. 185-199). Recuperado de <http://laboratorios.fi.uba.ar/lsi/c-icie99-ingenieriasoftwareeducativo.pdf>
- Chauhan, V. K. (2014). Smoke Testing. *IJSRP*, 4(2), 1-5. Recuperado de <http://www.ijsrp.org/research-paper-0214/ijsrp-p2663.pdf>
- Chowdhury, A. E., Bhowmik, A., Hasan, H., & Rahim, M. S. (2017). Analysis of the Veracities of Industry Used Software Development Life Cycle Methodologies. *AJSE*, 16(2), 75-82. <https://doi.org/10.53799/ajse.v16i2.71>
- Chronic Duration. (2022). Recuperado 16 de junio de 2022, de https://github.com/henrypoydar/chronic_duration
- Cicnavi. (2011). THE DIFFERENCE BETWEEN UNICAST, MULTICAST AND BROADCAST MESSAGES. Recuperado 28 de marzo de 2022, de Utilize Windows: Information Technology Info website: <http://www.utilizewindows.com/the-difference-between-unicast-multicast-and-broadcast-messages/>
- CircleCI. (2022). Recuperado 7 de agosto de 2022, de <https://github.com/marketplace/circleci>
- Cisco. (2022). Introducción a redes: Direcciones de red IPv4. Recuperado 23 de marzo de 2022, de Cisco Networking Academy website: <http://itroque.edu.mx/cisco/cisco1/course/module8/#8.1.3.4>
- CloudBees CodeShip. (2022). Recuperado 7 de agosto de 2022, de <https://www.cloudbees.com/products/codeship>
- Commander. (2022). Recuperado 7 de abril de 2022, de <https://github.com/commander-rb/commander>
- Comprar TeamCity. (2022). Recuperado 7 de agosto de 2022, de <https://www.jetbrains.com/es-es/teamcity/buy/#on-premises>
- Coulouris, G., Dollimore, J., & Kindberg, T. (2001). *Sistemas Distribuidos: Conceptos y Diseño* (3^a Ed.). Madrid, España: Pearson Educación S.A.
- Davis Ryan. (2022). minitest-5.15.0 Documentación. Recuperado 17 de marzo de 2022, de <https://docs.seattlerb.org/minitest/>

- Diepenbeck, M., & Drechsler, R. (2015). Behavior Driven Development for Tests and Verification. En *Formal Modeling and Verification of Cyber-Physical Systems* (pp. 275-277). Wiesbaden: Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-09994-7_11
- Discourse. (2022). Recuperado 2 de mayo de 2022, de <https://github.com/rofaccess/discourse>
- dist _ test. (2022). Recuperado 7 de agosto de 2022, de https://github.com/cloudera/dist_test
- Dominguez, J. A. (2002). An Overview of Defense in Depth at each layer of the TCP/IP Model. *Global Information Assurance Certification*, p. 15. Recuperado de <https://www.giac.org/paper/gsec/2233/overview-defense-in-depth-layer-tcp-ip-model/103817>
- Duarte, C., & Amílcar, F. (2011). *Behavior-Driven Development*. Recuperado de https://www.academia.edu/510270/Behavior-Driven_Development
- Fitzgerald, B., & Stol, K. J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123, 176-189. <https://doi.org/10.1016/j.jss.2015.06.063>
- Flanagan, D., & Matsumoto, Y. (2008). *The Ruby Programming Language* (1ª ed.; M. Loukides, Ed.). O'Reilly Media.
- Fúquene Ardila, H. J. (2011). Implementación cliente servidor mediante sockets. *Revista Vínculos*, 8(2), 48-59. <https://doi.org/10.14483/2322939X.4197>
- Ganesan, D., Lindvall, M., McComas, D., Bartholomew, M., Slegel, S., Medina, B., ... Montgomery, L. P. (2013). An analysis of unit tests of a flight software product line. *Science of Computer Programming*, 78(12), 2360-2380. <https://doi.org/10.1016/j.scico.2012.02.006>
- Garrels, M. (2008). Bash Guide for Beginners. *Linux*, p. 216. Recuperado de https://tldp.org/LDP/Bash-Beginners-Guide/html/intro_10.html
- Gómez, D., Jústiz, D., & Delgado, M. (2013). Unit Tests of Software in a University Environment. *Computacion y Sistemas*, 17(1), 69-77. Recuperado de http://www.scielo.org.mx/scielo.php?script=sci_arttext&pid=S1405-55462013000100008&lng=es&tlng=en
- Grice, S. (2018, abril 2). Making a Command-line Ruby Gem — Write, Build, and Push. Recuperado 5 de abril de 2022, de Medium website: <https://stephenagrice.medium.com/making-a-command-line-ruby-gem-write-build-and-push-aec24c6c49eb>
- Guides - RubyGems Guides. (2022). Recuperado 4 de abril de 2022, de <https://guides.rubygems.org/>

- Hendricksons, E. (2008). Acceptance Test Driven Development (ATDD): an Overview. Recuperado 12 de marzo de 2022, de <https://testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview/>
- Highline. (2022). Recuperado 7 de abril de 2022, de <https://github.com/JEG2/highline>
- Hourquebie, L. (2018). Testing en Ruby: Minitest vs RSpec. Recuperado 17 de marzo de 2022, de Unagi website: <https://medium.com/unagi/testing-en-ruby-minitest-vs-rspec-683a3d5fab98>
- How to Start Logging With Ruby on Rails. (2022). Recuperado 28 de abril de 2022, de Better Stack website: <https://betterstack.com/community/guides/logging/how-to-start-logging-with-ruby-on-rails/#step-6-configuring-logger>
- IETF. (1981). *Transmission Control Protocol DARPA Internet Program Protocol Specification*. Recuperado de <https://www.rfc-editor.org/rfc/pdf/rfc793.txt.pdf>
- ISTQB. (2018). *Certified Tester Foundation Level (CTFL)* (p. 1450). p. 1450. ISTQB. Recuperado de https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CTFL_Syllabus_2018_v3.1.1.pdf
- Jenkins. (2022). Recuperado 7 de agosto de 2022, de <https://github.com/jenkinsci/jenkins>
- Juganaru Mathieu, M. (2014). *Introducción a la programación* (1ª ed.). Grupo Editorial Patria. Recuperado de http://librunam.dgbiblio.unam.mx:8991/F/BRA9Q9FFM63CI7Q3NLXUADFT8QFXVX8UHN4CQ5QJ2IUUNJIKNS-43152?func=full-set-set&set_number=030026&set_entry=000007&format=999
- Kaur, S., Singh, K., & Singh, Y. (2016). A Comparative Analysis of Unicast, Multicast, Broadcast and Anycast Addressing Schemes Routing in MANETs. *International Journal of Computer Applications*, 133(9), 16-22. <https://doi.org/10.5120/ijca2016908001>
- Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V. P., Itkonen, J., Mäntylä, M. V., & Männistö, T. (2015). The highways and country roads to continuous deployment. *IEEE Software*, 32(2), 64-72. <https://doi.org/10.1109/MS.2015.50>
- Linux Professional Institute. (s. f.). Shells y Shell Scripting. Recuperado 26 de abril de 2022, de https://learning.lpi.org/en/learning-materials/102-500/105/105.2/105.2_01/?gclid=CjwKCAjwsJ6TBhAIEiwAfl4TWDGIT611CG0B3oI0OvflKtyhjWoIWrcrPDKIXO7lIioBb8MCiZP4VhoCuIUQAvD_BwE
- Logger. (2022). Recuperado 28 de abril de 2022, de <https://ruby-doc.org/stdlib-2.7.0/libdoc/logger/rdoc/Logger.html>
- Maeda, S. (2022). The Ruby Language FAQ. Recuperado 21 de abril de 2022, de <http://docs.ruby-doc.com/docs/TheRubyLanguageFAQ/>

- Marrero, D. (2000). *Protocolos de Transporte en Redes de Comunicaciones* (Universida). Las Palmas de Gran Canaria, España. Recuperado de <http://hdl.handle.net/10553/1337>
- Myers, G. J., Badgett, T., Thomas, T. M., & Sandler, C. (2004). *The Art of Software Testing* (2ª ed.). New Jersey: John Wiley & Sons.
- Net-scp. (2022). Recuperado 7 de abril de 2022, de <https://github.com/net-ssh/net-scp>
- North, D. (2006). Introducing BDD. Recuperado 17 de abril de 2022, de DAN NORTH & ASSOCIATES LTD website: <https://dannorth.net/introducing-bdd/>
- Nyakundi, H. (2021). SCP Linux Command – How to SSH File Transfer from Remote to Local. Recuperado 20 de mayo de 2022, de <https://www.freecodecamp.org/news/scp-linux-command-example-how-to-ssh-file-transfer-from-remote-to-local/>
- Oladimeji, P. (2007). Levels of Testing (Swansea Univerity). Swansea Univerity. Recuperado de <https://www.cs.swan.ac.uk/~csmarkus/CS339/dissertations/OladimejiP.pdf>
- Olan, M. (2003). Unit testing: test early, test often. *Journal of Computer Science in College*, 19(2), 319-328. <https://doi.org/10.5555/948785.948830>
- Oracle. (2010). Introducción al conjunto de protocolos TCP/IP (Guía de administración del sistema: servicios IP). Recuperado 12 de marzo de 2022, de Oracle website: <https://docs.oracle.com/cd/E19957-01/820-2981/6nei0r0r9/index.html>
- Osherove, R. (2014). *The Art of Unit Testing* (2ª ed.). Shelter Island, NY 11964: Manning Publications Co.
- Pauta Ayabaca, L., & Moscoso Bernal, S. (2017). Verificación y Validación de Software. *Killkana Técnica*, 1(3), 25-32. https://doi.org/10.26871/killkana_tecnica.v1i3.112
- Petersen, R. (2008). *Linux: The Complete Reference* (6ª ed.). McGraw-Hill.
- Pineda, D., & Méndez, D. (2018). Socket , Sistema Elemental para la Comunicación Interprocesos. Recuperado 28 de marzo de 2022, de https://www.researchgate.net/publication/333731438_Socket_Sistema_Elemental_para_la_Comunicacion_Interprocesos
- Pressman, R. S. (2002). *Ingeniería del Software Un enfoque práctico* (5ª ed.; McGraw-Hill/Interamericana de España, Ed.). Madrid, España.
- Pressman, R. S. (2010). *Ingeniería del software Un enfoque práctico* (7ª ed.; McGraw-Hill Interamericana, Ed.). México, D. F.
- Publishing your gem - RubyGems Guides. (2022). Recuperado 8 de abril de 2022, de <https://guides.rubygems.org/publishing/>
- Rake. (2022). Recuperado 7 de abril de 2022, de <https://github.com/ruby/rake>

- Rocha, A. C. (2010). Guía de Testing de Software: Principales tipos de Pruebas de Software. Recuperado 29 de marzo de 2022, de <https://gtsw-es.blogspot.com/2010/12/principales-tipos-de-testing-de.html>
- RSpec. (2022). Recuperado 12 de marzo de 2022, de <https://relishapp.com/rspec/>
- Rubocop. (2022). Recuperado 7 de abril de 2022, de <https://github.com/rubocop/rubocop>
- Ruby Version Management - The Ruby Toolbox. (2022). Recuperado 4 de abril de 2022, de https://www.ruby-toolbox.com/categories/ruby_version_management
- RubyGems.org | your community gem host. (2022). Recuperado 4 de abril de 2022, de <https://rubygems.org/>
- RubyGems Basics - RubyGems Guides. (2022). Recuperado 4 de abril de 2022, de <https://guides.rubygems.org/rubygems-basics/>
- Rubyzip. (2022). Recuperado 7 de abril de 2022, de <https://github.com/rubyzip/rubyzip>
- Ruparelia, N. B. (2010). Software Development Lifecycle Models. *SIGSOFT Softw. Eng. Notes*, 35(3), 8–13. <https://doi.org/10.1145/1764810.1764814>
- RVM: Ruby Version Manager - Documentation. (2022). Recuperado 4 de abril de 2022, de <https://rvm.io/>
- RVM: Ruby Version Manager - Gemset Basics. (2022). Recuperado 7 de abril de 2022, de <https://rvm.io/gemsets/basics>
- RVM: Ruby Version Manager - Installing RVM. (2022). Recuperado 5 de abril de 2022, de <https://rvm.io/rvm/install>
- Schwaber, K., & Sutherland, J. (2013). La Guía de Scrum La Guía Definitiva de Scrum: Las Reglas del Juego. *Scrum.org*, 21. Recuperado de <http://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-es.pdf>
- Schwaber, K., & Sutherland, J. (2020). *La Guía de Scrum* (p. 15). p. 15. Recuperado de <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Spanish-Latin-South-American.pdf>
- Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5, 3909-3943. <https://doi.org/10.1109/ACCESS.2017.2685629>
- Soraluz Soraluz, A. E., Valles Coral, M. Á., & Lévano Rodríguez, D. (2021). Desarrollo guiado por comportamiento: buenas prácticas para la calidad de software. *Ingeniería y Desarrollo*, 39(1), 190-204. Recuperado de http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0122-34612021000100190
- SSH.COM. (s. f.). SSH Protocol – Secure Remote Login and File Transfer. Recuperado de <https://www.ssh.com/academy/ssh/protocol>

- Tavarez, J. A. (2014). El software y su importancia. Recuperado 12 de abril de 2022, de El día cada día mejor website: <https://eldia.com.do/el-software-y-su-importancia/>
- Terminal Table. (2022). Recuperado 16 de junio de 2022, de <https://github.com/tj/terminal-table>
- The LWN.net Linux Distribution List. (2021). Recuperado 23 de abril de 2022, de <https://lwn.net/Distributions/>
- Travis CI. (2022). Recuperado 7 de agosto de 2022, de <https://www.travis-ci.com>
- TTY Progressbar. (2022). Recuperado 16 de junio de 2022, de <https://github.com/piotrmurach/tty-progressbar>
- tutorialspoint.com. (2011). *Software Testing Tutorial*.
- Verdejo Alvarez, G. (2003). *SEGURIDAD EN REDES IP* (Universitat Autònoma de Barcelona). Universitat Autònoma de Barcelona. Recuperado de <https://www.cs.upc.edu/~gabriel/>
- Wang, A. (2016). Quality Assurance at Cloudera: Distributed Unit Testing - Cloudera Engineering Blog. Recuperado 14 de julio de 2018, de Cloudera Engineering Blog website: <http://blog.cloudera.com/blog/2016/05/quality-assurance-at-cloudera-distributed-unit-testing/>
- What is a gem? - RubyGems Guides. (2022). Recuperado 4 de abril de 2022, de <https://guides.rubygems.org/what-is-a-gem/>
- Yard. (2022). Recuperado 7 de abril de 2022, de <https://github.com/lsegal/yard>

ANEXO

Anexo A - Instalación de RVM, Ruby y Bundler

A continuación se muestran los comandos a ser ejecutados en una línea de comandos, para instalar RVM y Ruby en la distribución Linux denominada Manjaro. Las instrucciones están basadas según lo indicado en («RVM: Ruby Version Manager - Documentation», 2022) y («RVM: Ruby Version Manager - Installing RVM», 2022).

Pasos

1. Instalar claves gpg2

```
liri@liri-sys:~$ gpg2 --recv-keys 409B6B1796C275462A1703113804BB82D39DC0E3  
7D2BAF1CF37B13E2069D6956105BD0E739499BDB
```

2. Instalar RVM

```
liri@liri-sys:~$ sudo pacman -S curl  
liri@liri-sys:~$ \curl -sSL https://get.rvm.io | bash -s stable  
liri@liri-sys:~$ source $HOME/.rvm/scripts/rvm
```

3. Instalar Ruby

```
liri@liri-sys:~$ rvm install 2.7.2  
liri@liri-sys:~$ rvm use 2.7.2
```

El comando **rvm use** sirve para establecer la versión de Ruby a utilizar.

4. Crear un Gemset

RVM ofrece configuraciones de Ruby independientes y compartimentadas, esto se logra a través de los gemset, esto permite tener conjuntos de gemas separadas en gemsets acorde a las necesidades de cada proyecto en la que se esté trabajando («RVM: Ruby Version Manager - Gemset Basics», 2022). Con los siguientes comandos procedemos a crear y utilizar un gemset para el proyecto liri:

```
liri@liri-sys:~$ rvm gemset create liri  
liri@liri-sys:~$ rvm gemset use liri
```

5. Verificar Bundler

```
liri@liri-sys:~$ bundle -v
```

Al instalar Ruby ya se instala por defecto una versión de bundler. Al momento de la ejecución del comando indicado, el resultado es el siguiente: Bundler version 2.1.4

Anexo B – Configuración del sistema Liri

Como se explicó en la sección 4.1.2 Inicio de configuraciones, para configurar el comportamiento del sistema Liri, se tiene el archivo **liri-config.yml**. El contenido del archivo de configuración es el siguiente:

```
# Este archivo .yml define ciertas configuraciones para la ejecución del sistema
# Configuraciones exclusivas del Manager
manager:
  udp_request_delay: 3      # Define cada cuantos segundos el Manager enviará un broadcast UDP para
                             # sondear la red buscando Agents
  test_files_by_runner: 30 # Cantidad de archivos de tests a ejecutar en cada tanda
  # Configuraciones que definen que barras se muestran o no
  bar:
    # Mostrar o no el progreso de distribución de código fuente
    share_source_code: true
  # Configuraciones que definen la información mostrada al terminar la ejecución de las pruebas
  print:
    table:
      summary: true  # Configuración para imprimir un resumen del proceso realizado por los Agents
      detailed: false # Configuración para imprimir un resumen detallado del proceso realizado
                      # por los Agents
    column:
      failed_files: true  # Muestra los archivos que contienen las pruebas que fallaron
      files_load: true    # Muestra el tiempo que tardó RSpec en cargar los archivos de pruebas
      finish_in: true     # Muestra el tiempo que tardó el Agent en ejecutar las pruebas
      batch_run: true     # Muestra el tiempo que pasó desde que el Manager obtiene las
                          # pruebas a ejecutar, las envía y recibe los resultados
      share_source_code: true # Muestra el tiempo que tardan los Agents en obtener el código
                              # fuente y descomprimirlo
    failures:
      summary: false # Muestra una lista con los nombres de los archivos que contienen las pruebas
                    # que fallaron
      detailed: false # Muestra una lista detallada con los nombres de los archivos que contienen
                     # las pruebas que fallaron
# Configuraciones compartidas entre Manager y Agent
general:
  # Configuración del nombre del archivo de código fuente comprimido enviado a los Agents
  # Obs.: Puede que NO sea útil para el usuario poder cambiar este nombre
  compressed_file_name: compressed_source_code
  # Define las carpetas que deben ignorarse en la compresión del código fuente. Debe ir separado
  # por comas y debe haber un espacio después de ignored_folders_in_compress:
  ignored_folders_in_compress: .git,liri,documents,installers,log,logs
  # Define a cuantos digitos se van a redondear los tiempos mostrados en pantalla
  times_round: 0
  # Define el tipo de redondeo a utilizar. floor para piso, roof para techo
  # El que da mejores resultados según las pruebas realizadas es usar times_round 0 y
  # times_round_type floor
  times_round_type: floor
  # Ej.:
  # Con times_round 0 y times_round_type floor, 3.5516 se convierte a 3s
  # Con times_round 1 y times_round_type floor, 3.5516 se convierte a 3.5s
  # Con times_round 2 y times_round_type floor, 3.5516 se convierte a 3.55s
  # Con times_round 0 y times_round_type roof, 3.5516 se convierte a 4s
  # Con times_round 1 y times_round_type roof, 3.5516 se convierte a 3.6s
  # Con times_round 2 y times_round_type roof, 3.5516 se convierte a 3.55s
  # Configuraciones de logs
  log:
    # Configuración del log mostrado en la terminal
    stdout:
      # Define si se muestra el log de ejecución del programa en línea de comando. Puede ser true
      # o false. Por defecto siempre se imprimirá el log en un archivo dentro de la carpeta logs
      # Si se pone a true, interfiere con la barra de progreso del Manager, mejor tenerlo en false
      show: false
      # Define los colores del texto del log. Puede ser none (no muestra colores), severity
      # (colorea código de error), severity_date (colorea código de error y fecha),
      # full (colorea el texto entero)
      colorize: full
    # Configuración del log guardado en los archivos .log
    file:
      colorize: full # Puede ser none, severity o full
  library:
    compression: Zip # Configuración de la librería de compresión a utilizar para comprimir el código
                     # fuente. Valores soportados: Sólo Zip hasta ahora
    unit_test: Rspec # Configuración de la librería de pruebas unitarias a ejecutar
                    # Valores soportados: Sólo Rspec hasta ahora
# Configuración de puertos
ports:
  udp: 2000 # Configuración del puerto a través del cual se realizará la primera comunicación
            # entre Manager y Agent
  tcp: 2500 # Configuración del puerto a través del cual el Agent y el Manager intercambiarán
            # las pruebas a ejecutar y los resultados de esa ejecución
```

Anexo C – Instalación del Agente Liri dentro de la distribución Manjaro

El entorno de trabajo para realizar las pruebas del sistema Liri, debe configurarse en los siguientes aspectos:

Instalación de requerimientos: Consiste en instalar los programas que son requeridos para que el script de instalación de la Aplicación Agente pueda finalizar exitosamente y el sistema Liri pueda funcionar sin inconvenientes. El instalador de la Aplicación Agente ya se encarga de instalar RVM y Ruby, por este motivo se requiere instalar los programas: curl, gcc y make, los cuales son requisitos para la instalación de RVM y Ruby.

Instalación del sistema: Consiste en ejecutar el script de instalación del Agente Liri.

Configuración del sistema operativo: Consiste en realizar las configuraciones específicas según lo requiera cada distribución para que el sistema Liri funcione sin inconvenientes.

Pasos:

1. Actualizar sistema operativo

```
liri@liri-virtualbox:~$ sudo pacman -Syu
```

No se pueden instalar nuevos programas sin antes actualizar el sistema operativo. Este proceso puede llevar un tiempo, en algunos casos puede que ocurran errores que impidan que el sistema sea actualizado correctamente, resolver este tipo de problemas queda fuera de este instructivo.

2. Instalar requerimientos

```
liri@liri-virtualbox:~$ sudo pacman -S curl gcc make
```

3. Descargar el instalador de la Aplicación Agente

```
liri@liri-virtualbox:~$ wget -O liriagent.zip  
https://github.com/roface/liri/blob/master/installers/liriagent.zip?raw=true
```

4. Descomprimir el instalador

```
liri@liri-virtualbox:~$ unzip liriagent.zip
```

5. Ejecutar el instalador

```
liri@liri-virtualbox:~$ cd liriagent/bin  
liri@liri-virtualbox:~/liriagent/bin$ ./install.sh
```

Para más información lea atentamente las instrucciones mostradas al ejecutar el comando.

En algunos momentos se puede requerir el ingreso de la contraseña de usuario.

En la Figura 36 se muestra un ejemplo de la ejecución del instalador. El instalador detecta el sistema operativo utilizado y muestra ciertas instrucciones para facilitar el proceso de instalación. Se comprobó el funcionamiento satisfactorio del instalador en las distribuciones Linux: Manjaro, Debian, Ubuntu y Fedora, por lo que el instalador tiene instrucciones específicas para cada una de esas distribuciones, estas instrucciones se mostrarán al momento de ejecutar el instalador. Si el instalador es ejecutado en alguna otra distribución que no sean las listadas anteriormente, se mostrarán instrucciones genéricas.

```
liri@liri-sys:~/discourse$ ./install.sh
+-----+
INICIO: Proceso de instalación del programa Agent
+-----+
INFO: Distribución detectada: Ubuntu

INFO: Para finalizar satisfactoriamente la instalación debe tener actualizada el sistema operativo y tener instalado los programas necesarios
> sudo apt-get update
> sudo apt update
> sudo apt install openssh-server curl gcc make inxi
INFO: Comandos probados en Ubuntu 21.10 (Impish Indri)
INFO: En algunos momentos se requerirá el ingreso de la contraseña sudo o root
INFO: Asegurese de que el servicio ssh esté instalado y ejecutandose
> sudo systemctl status sshd
> sudo systemctl enable sshd
> sudo systemctl start sshd

Presione Enter para continuar o la tecla 's' + Enter para salir
```

Figura 36: *Instalación de la Aplicación Agente.*

Fuente: Elaboración propia.

6. Comprobar que la Aplicación Agente está ejecutándose

```
liri@liri-virtualbox:~/liriagent/bin$ sudo systemctl status liriagent.service
```

El resultado del comando anterior debe ser similar a lo mostrado en la Figura 37.

```
● liriagent.service - Liri Agent
   Loaded: loaded (/etc/systemd/system/liriagent.service; enabled; vendor pre>
   Active: active (running) since Tue 2022-07-05 08:24:08 -04; 12h ago
   Main PID: 908 (startup.sh)
     Tasks: 3 (limit: 9212)
    Memory: 54.7M
   CGroup: /system.slice/liriagent.service
           └─ 908 /bin/bash /home/liri/liriagent/bin/startup.sh 2.7.2
              1600 ruby /home/liri/.rvm/gems/ruby-2.7.2@liri/bin/liri a --tr>

abr 20 08:24:08 liri systemd[1]: Started Liri Agent.
```

Figura 37: *Estado de la Aplicación Agente.*

Fuente: Elaboración propia.

6. Configurar el sistema operativo

```
liri@liri-virtualbox:~/liriagent/bin$ sudo systemctl enable sshd.service
liri@liri-virtualbox:~/liriagent/bin$ sudo systemctl start sshd.service
```

Ese necesario activar e iniciar el servicio ssh porque es usado por la Aplicación Agente para obtener el código fuente de la aplicación cuyas pruebas se requiere ejecutar.

Anexo D - Configuración de entorno para probar el sistema Liri

Existe una gran variedad de aplicaciones open source sobre las cuales se puede ejecutar Liri. Se descargaron e intentaron ejecutar las pruebas de varias aplicaciones, algunas se pudieron configurar y ejecutar sus pruebas, en cambio, otras no se pudieron configurar correctamente, al final seleccionamos las aplicaciones Discourse y Consul para nuestras pruebas del Sistema Liri.

En general las aplicaciones deben configurarse en los siguientes aspectos:

Instalación y configuración de base de datos: Consiste en instalar el programa de base de datos requerida y crear los usuarios correspondientes.

Instalación de Ruby: Consiste en instalar la versión de Ruby requerida por la aplicación.

Descarga y configuración de la aplicación: Consiste en clonar el repositorio del proyecto alojado en Github. Configurar el archivo database.yml de la aplicación descargada. Crear los archivos .ruby-version en donde se especifica la versión de Ruby utilizada y el archivo .ruby-gemset en donde se especifica el gemset utilizado.

Estas configuraciones suelen variar entre aplicaciones, por este motivo decidimos crear un fork de las aplicaciones que probamos. En estos forks ya tenemos configurado lo necesario. Tener forks también nos permite tener unas aplicaciones de prueba que sean compatibles con la gema Liri, en caso de que los desarrolladores de esas aplicaciones realicen cambios que causen alguna incompatibilidad con la gema Liri.

Instalación de gemas: Consiste en instalar las gemas requeridas por la aplicación.

Inicialización de la base de datos: Consiste en crear e inicializar la base de datos de pruebas.

A continuación se indica como instalar y configurar la base de datos para ambas aplicaciones de prueba.

Pasos:

1. Instalación de Postgresql 13

```
liri@liri-virtualbox:~$ sudo pacman -S postgresql
```

2. Configuración de Postgresql

```
liri@liri-virtualbox:~$ sudo -iu postgres
postgres@liri-virtualbox:~$ initdb -D /var/lib/postgres/data
postgres@liri-virtualbox:~$ exit
liri@liri-virtualbox:~$ sudo systemctl start postgresql
liri@liri-virtualbox:~$ sudo systemctl enable postgresql
```

3. Configuración de la base de datos para las aplicaciones

```
liri@liri-virtualbox:~$ sudo su postgres
postgres@liri-virtualbox:~$ psql
postgres=#: create role liri with superuser login password 'liri';
postgres=#: create database discourse_test;
postgres=#: create database consul_test;
postgres=#: \q
```

Instalación y Configuración de Discourse

Discourse es una plataforma de discusión 100% Open Source desarrollada para la siguiente década de Internet. Se puede utilizar como una lista de correo, un foro de discusión o una sala de chat de larga duración («Discourse», 2022).

Pasos

1. Instalación y configuración de Redis

Discourse requiere Redis para ejecutar la migración de datos.

```
liri@liri-virtualbox:~$ sudo pacman -S redis
liri@liri-virtualbox:~$ sudo systemctl enable redis
liri@liri-virtualbox:~$ sudo systemctl start redis
```

2. Instalación de Ruby

Discourse requiere una versión específica de bundler, la cual instalamos dentro de un gemset luego de instalar la versión requerida de Ruby. Para más detalles sobre este paso, mirar el Anexo A.

```
liri@liri-virtualbox:~$ rvm install 2.7.2
liri@liri-virtualbox:~$ /bin/bash --login
liri@liri-virtualbox:~$ rvm use 2.7.2
liri@liri-virtualbox:~$ rvm gemset create discourse
liri@liri-virtualbox:~$ rvm gemset use discourse
liri@liri-virtualbox:~$ gem install bundler --version 2.3.4
```

3. Descargar Discourse

Clonamos el fork de discourse que creamos para nuestras pruebas.

```
liri@liri-virtualbox:~$ git clone https://github.com/rofacecess/discourse.git
```

4. Instalar gemas

```
liri@liri-virtualbox:~$ cd discourse
liri@liri-virtualbox:~/discourse$ /bin/bash --login
liri@liri-virtualbox:~/discourse$ bundle install
```

5. Iniciar la base de datos

```
liri@liri-virtualbox:~/discourse$ rake db:migrate:reset RAILS_ENV=test
```

6. Probar la ejecución de pruebas en el entorno local

```
liri@liri-virtualbox:~/discourse$ rake spec RAILS_ENV=test
```

7. Probar la ejecución de pruebas usando la gema Liri

```
liri@liri-virtualbox:~/discourse$ gem install liri
liri@liri-virtualbox:~/discourse$ liri m
```

En otra terminal de la misma computadora o en otra computadora puede ejecutar una Aplicación Agente.

```
liri@liri-virtualbox:~/discourse$ liri a
```

Instalación y Configuración de Consul

Consul es una aplicación de participación ciudadana y gobierno abierto desarrollado originalmente para el sitio web de participación ciudadana del Ayuntamiento de Madrid (“Consul”, 2022).

Pasos

1. Instalación de Ruby

Consul requiere una versión específica de bundler, la cual instalamos dentro de un gemset luego de instalar la versión requerida de Ruby. Para más detalles sobre este paso, mirar el Anexo A.

```
liri@liri-virtualbox:~$ rvm install 2.7.4
liri@liri-virtualbox:~$ /bin/bash --login
liri@liri-virtualbox:~$ rvm use 2.7.4
liri@liri-virtualbox:~$ rvm gemset create consul
liri@liri-virtualbox:~$ rvm gemset use consul
liri@liri-virtualbox:~$ gem install bundler --version 2.1.4
```

3. Descargar Consul

Clonamos el fork de Consul que creamos para nuestras pruebas.

```
liri@liri-virtualbox:~$ git clone https://github.com/rofacecess/consul.git
```

4. Instalar requerimientos

Consul usa una gema llamada rugged el cual requiere tener instalado cmake y pkgconf para que pueda ser instalado. Además, se requiere tener instalado nodejs para poder configurar la base datos.

```
liri@liri-virtualbox:~$ sudo pacman -S cmake pkgconf nodejs
```

Parte de las pruebas unitarias de Consul son pruebas de integración, las cuales requieren del navegador Google Chrome.

Para instalar Google Chrome se debe acceder a la configuración de preferencias del gestor de Software de Manjaro como lo indica la Figura 38 y habilitar la opción Enable

AUR Support como lo indica la Figura 39. Luego se cierra la ventana de preferencias y se busca e instala el navegador Google Chrome como se indica en la Figura 40.

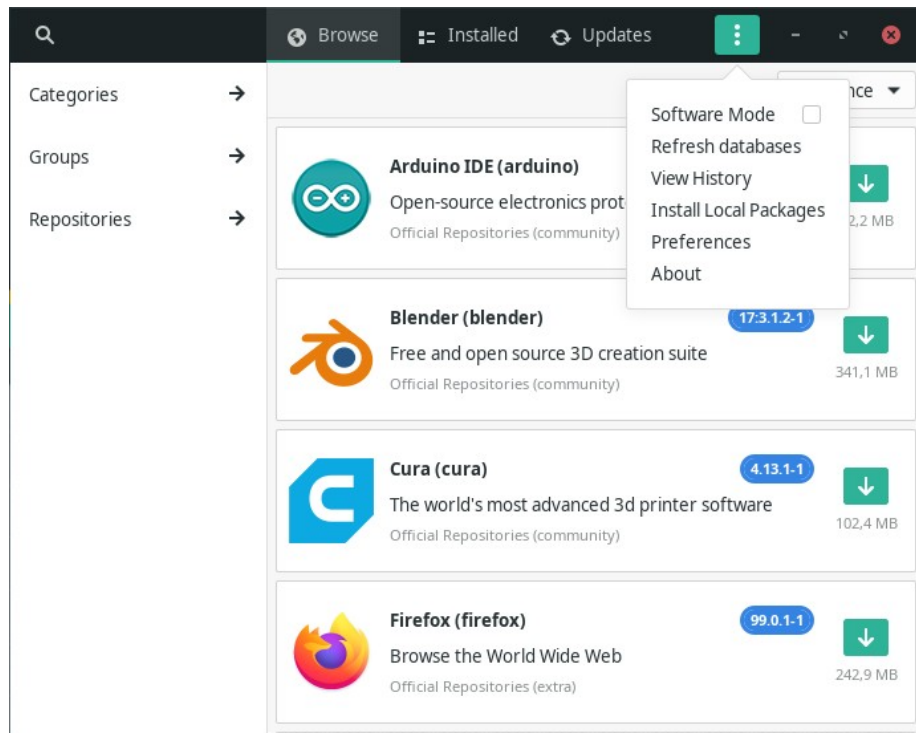


Figura 38: Gestor de software de Manjaro.

Fuente: Elaboración propia.

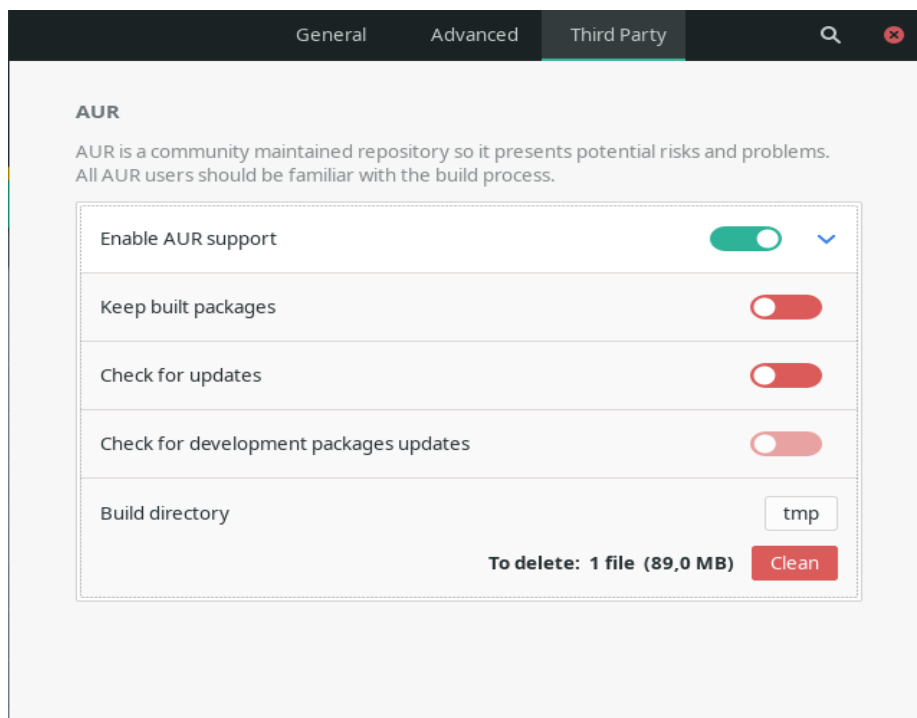


Figura 39: Configurar software de terceros en Manjaro.

Fuente: Elaboración propia.

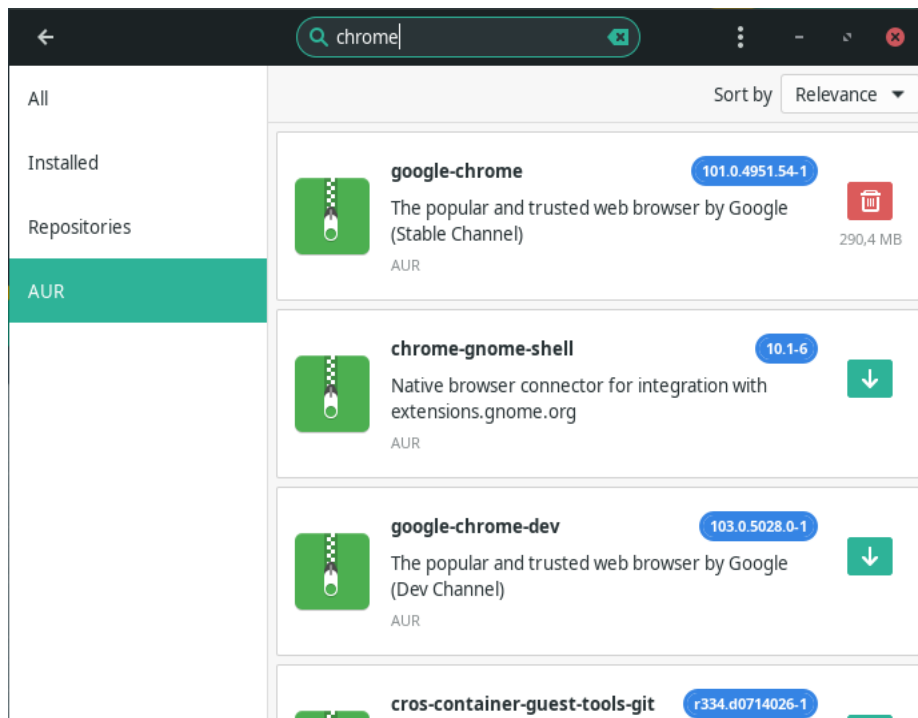


Figura 40: *Instalación de Google Chrome.*

Fuente: Elaboración propia.

5. Instalar gemas

```
liri@liri-virtualbox:~$ cd consul
liri@liri-virtualbox:~/consul$ /bin/bash --login
liri@liri-virtualbox:~/consul$ bundle install
```

6. Iniciar la base de datos

```
liri@liri-virtualbox:~/consul$ RAILS_ENV=test rake db:setup
```

7. Probar la ejecución de pruebas en el entorno local

```
liri@liri-virtualbox:~/consul$ bin/rspec
```

8. Probar la ejecución de pruebas usando la gema Liri

```
liri@liri-virtualbox:~/consul$ gem install liri
liri@liri-virtualbox:~/consul$ liri m
```

En otra terminal de la misma computadora o en otra computadora puede ejecutar una Aplicación Agente.

```
liri@liri-virtualbox:~/consul$ liri a
```

Anexo E – Resultados de la ejecución del sistema Liri sobre la aplicación Discourse

En esta sección se listan las tablas que registran de manera detallada los resultados de las diferentes pruebas de ejecución del sistema Liri y al final se muestra la Tabla 30 que resume todos los resultados.

Los entornos de prueba corresponden a los definidos en la Tabla 6.

Resultados de ejecución en todos los entornos con conjuntos de 30 archivos

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 1		Aplicación: Discourse		
Entorno de prueba: A		Cantidad de pruebas unitarias: 16.341		
Tamaño de Conjunto: 30		Cantidad de archivos: 816		
Tiempo de ejecución: 26m 10s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	816	16.341	16.333	8
Total	816	16.341	16.333	8

Tabla 18: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno A con un tamaño de conjunto de 30 archivos.

Fuente: Elaboración propia.

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 2		Aplicación: Discourse		
Entorno de prueba: B		Cantidad de pruebas unitarias: 16341		
Tamaño de Conjunto: 30		Cantidad de archivos: 816		
Tiempo de ejecución: 13m 33s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	390	7.015	7.013	2
ASUS	426	9.326	9.320	6
Total	816	16.341	16.333	8

Tabla 19: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno B con un tamaño de conjunto de 30 archivos.

Fuente: Elaboración propia.

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 3		Aplicación: Discourse		
Entorno de prueba: C		Cantidad de pruebas unitarias: 16341		
Tamaño de Conjunto: 30		Cantidad de archivos: 816		
Tiempo de ejecución: 08m 00s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	186	4.192	4.191	1
ASUS	270	4.541	4.540	1
MSI	360	7.608	7.602	6
Total	816	16.341	16.333	8

Tabla 20: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno C con un tamaño de conjunto de 30 archivos.

Fuente: Elaboración propia.

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 4		Aplicación: Discourse		
Entorno de prueba: D		Cantidad de pruebas unitarias: 16341		
Tamaño de Conjunto: 30		Cantidad de archivos: 816		
Tiempo de ejecución: 05m 49s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	150	2.408	2.407	1
ASUS	156	2.891	2.891	0
MSI	210	4.166	4.160	6
AORUS	300	6.876	6.875	1
Total	816	16.341	16.333	8

Tabla 21: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno D con un tamaño de conjunto de 30 archivos.

Fuente: Elaboración propia.

Resultados de ejecución en todos los entornos con conjuntos de 50 archivos

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 5		Aplicación: Discourse		
Entorno de prueba: A		Cantidad de pruebas unitarias: 16.341		
Tamaño de Conjunto: 50		Cantidad de archivos: 816		
Tiempo de ejecución: 24m 12s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	816	16.341	16.333	8
Total	816	16.341	16.333	8

Tabla 22: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno A con un tamaño de conjunto de 50 archivos.

Fuente: Elaboración propia.

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 6		Aplicación: Discourse		
Entorno de prueba: B		Cantidad de pruebas unitarias: 16341		
Tamaño de Conjunto: 50		Cantidad de archivos: 816		
Tiempo de ejecución: 13m 14s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	416	6.893	6.886	7
ASUS	400	9.448	9.447	1
Total	816	16.341	16.333	8

Tabla 23: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno B con un tamaño de conjunto de 50 archivos.

Fuente: Elaboración propia.

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 7		Aplicación: Discourse		
Entorno de prueba: C		Cantidad de pruebas unitarias: 16341		
Tamaño de Conjunto: 50		Cantidad de archivos: 816		
Tiempo de ejecución: 07m 32s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	166	3.520	3.519	1
ASUS	300	4.406	4.399	7
MSI	350	8.415	8.415	1
Total	816	16.341	16.333	8

Tabla 24: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno C con un tamaño de conjunto de 50 archivos.

Fuente: Elaboración propia.

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 8		Aplicación: Discourse		
Entorno de prueba: D		Cantidad de pruebas unitarias: 16341		
Tamaño de Conjunto: 50		Cantidad de archivos: 816		
Tiempo de ejecución: 05m 24s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	150	2.748	2.748	0
ASUS	150	2.640	2.634	6
MSI	250	4.732	4.731	1
AORUS	266	6.221	6.220	1
Total	816	16.341	16.333	8

Tabla 25: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno D con un tamaño de conjunto de 50 archivos.

Fuente: Elaboración propia.

Resultados de ejecución en todos los entornos con conjuntos de 100 archivos

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 9		Aplicación: Discourse		
Entorno de prueba: A		Cantidad de pruebas unitarias: 16.341		
Tamaño de Conjunto: 100		Cantidad de archivos: 816		
Tiempo de ejecución: 22m 57s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	816	16.341	16.333	8
Total	816	16.341	16.333	8

Tabla 26: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno A con un tamaño de conjunto de 100 archivos.

Fuente: Elaboración propia.

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 10		Aplicación: Discourse		
Entorno de prueba: B		Cantidad de pruebas unitarias: 16341		
Tamaño de Conjunto: 100		Cantidad de archivos: 816		
Tiempo de ejecución: 12m 42s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	400	9.736	9.730	6
ASUS	416	6.605	6.603	2
Total	816	16.341	16.333	8

Tabla 27: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno B con un tamaño de conjunto de 100 archivos.

Fuente: Elaboración propia.

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 11		Aplicación: Discourse		
Entorno de prueba: C		Cantidad de pruebas unitarias: 16341		
Tamaño de Conjunto: 100		Cantidad de archivos: 816		
Tiempo de ejecución: 07m 41s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	200	4.822	4.820	2
ASUS	200	3.706	3.706	0
MSI	416	7.813	7.807	6
Total	816	16.341	16.333	8

Tabla 28: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno C con un tamaño de conjunto de 100 archivos.

Fuente: Elaboración propia.

PLANILLA DETALLADA DE RESULTADOS				
N.º de prueba: 12		Aplicación: Discourse		
Entorno de prueba: D		Cantidad de pruebas unitarias: 16.341		
Tamaño de Conjunto: 100		Cantidad de archivos: 816		
Tiempo de ejecución: 05m 09s				
Computadora	RESULTADOS			
	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
HP	116	1.803	1.802	1
ASUS	200	2.957	2.956	1
MSI	200	6.452	6.451	1
AORUS	300	5.129	5.124	5
Total	816	16.341	16.333	8

Tabla 29: Registro de resultados de pruebas de ejecución del sistema Liri en el entorno D con un tamaño de conjunto de 100 archivos.

Fuente: Elaboración propia.

Resultados de ejecución en todos los entornos con todos los conjuntos

En la Tabla 30 se registra el resumen de resultados de todas las pruebas de ejecución realizadas al sistema Liri.

PLANILLA RESUMIDA DE RESULTADOS							
Aplicación: Discourse		Cantidad de pruebas unitarias: 16.341			Cantidad de archivos: 816		
VARIABLES			RESULTADOS				
N.º de prueba	Entorno de prueba	Tamaño de Conjunto	Tiempo de ejecución	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
1	A	30	26m 10s	816	16.341	16.333	8
2	B	30	13m 33s	816	16.341	16.333	8
3	C	30	08m 00s	816	16.341	16.333	8
4	D	30	05m 49s	816	16.341	16.333	8
5	A	50	24m 12s	816	16.341	16.333	8
6	B	50	13m 14s	816	16.341	16.333	8
7	C	50	07m 32s	816	16.341	16.333	8
8	D	50	05m 24s	816	16.341	16.332	8
9	A	100	22m 57s	816	16.341	16.333	8
10	B	100	12m 42s	816	16.341	16.333	8
11	C	100	07m 41s	816	16.341	16.333	8
12	D	100	05m 09s	816	16.341	16.333	8

Tabla 30: Registro de resultados de pruebas de la ejecución del sistema Liri.

Fuente: Elaboración propia.

Anexo F – Resultados de la ejecución del sistema Liri sobre la aplicación Consul

Para demostrar que el sistema Liri funciona sobre otras aplicaciones, se utilizó otro proyecto Open Source hecho en Ruby on Rails denominado **Consul**, el cual posee **6.229** pruebas unitarias, distribuidas en un total de **497** archivos.

Cabe aclarar que Consul también tiene pruebas funcionales que se ejecutan con Selenium, es decir que Liri no ejecuta solamente pruebas unitarias. Esto es posible gracias a que Rspec permite implementar pruebas unitarias y funcionales.

En la Tabla 31 se presentan los entornos utilizados para las pruebas sobre la aplicación Consul.

Entorno de Prueba	Computadora Coordinadora	Computadoras Agente
E	AORUS	AORUS
F	AORUS	ASUS AORUS
G	AORUS	ASUS MSI AORUS

Tabla 31: Entornos utilizados para las pruebas del sistema Liri con la aplicación Consul.

Fuente: Elaboración propia.

En la Tabla 32 se presenta los resultados de la ejecución de las pruebas unitarias de la aplicación Consul de la forma convencional, es decir sin el sistema Liri.

En cada computadora se ejecutaron **6.229** pruebas, de las cuales **6.200** pasaron y **29** fallaron.

Computadora	Tiempo de ejecución
ASUS	62m 16s
MSI	39m 49s
AORUS	43m 22s

Tabla 32: Tiempo de ejecución por computadora de pruebas unitarias de la aplicación Consul sin usar Liri.

Fuente: Elaboración propia.

En la Tabla 33 se presenta el resumen de resultados de todas las pruebas de ejecución realizadas al sistema Liri sobre la aplicación Consul. Cabe aclarar que las pruebas que se hicieron fueron en menor cantidad y menos rigurosas porque no se necesitaba mucho nivel de detalle para demostrar que el sistema Liri funciona con otra aplicación aparte de Discourse.

PLANILLA RESUMIDA DE RESULTADOS							
Aplicación: Consul		Cantidad de pruebas unitarias: 6.229			Cantidad de archivos: 497		
VARIABLES			RESULTADOS				
N.º de prueba	Entorno de prueba	Tamaño de Conjunto	Tiempo de ejecución	Archivos procesados	Pruebas ejecutadas	Pruebas exitosas	Pruebas fallidas
1	E	50	44m 16s	497	6.229	6.200	29
2	F	50	28m 03s	497	6.229	6.200	29
3	G	50	18m 34s	497	6.229	6.200	29

Tabla 33: Registro de resultados de pruebas de la ejecución del sistema Liri sobre la aplicación Consul.

Fuente: Elaboración propia.

En la Tabla 34 se detallan los tiempos de ejecución del sistema Liri por cada entorno de prueba.

AORUS convencional	Entorno E 1 Agente	Entorno F 2 Agentes	Entorno G 3 Agentes
43m 22s	44m 16s	28m 03s	18m 34s

Tabla 34: Tiempo de ejecución del sistema Liri en cada entorno de prueba sobre la aplicación Consul.

Fuente: Elaboración propia.

Como se puede observar en la Figura 41, la diferencia entre la ejecución convencional comparando con la ejecución con el sistema Liri con 1 Agente es de 01m 06s, si bien, la diferencia es pequeña, esto demuestra que no es práctico utilizar el sistema Liri con 1 Agente por los gastos adicionales de tiempo que Liri usa en su ejecución. Pero, a partir de 2 Agentes ya se tiene una buena mejora del tiempo de ejecución, superando ya en este punto a la ejecución convencional de cada computadora utilizada en los entornos de prueba.

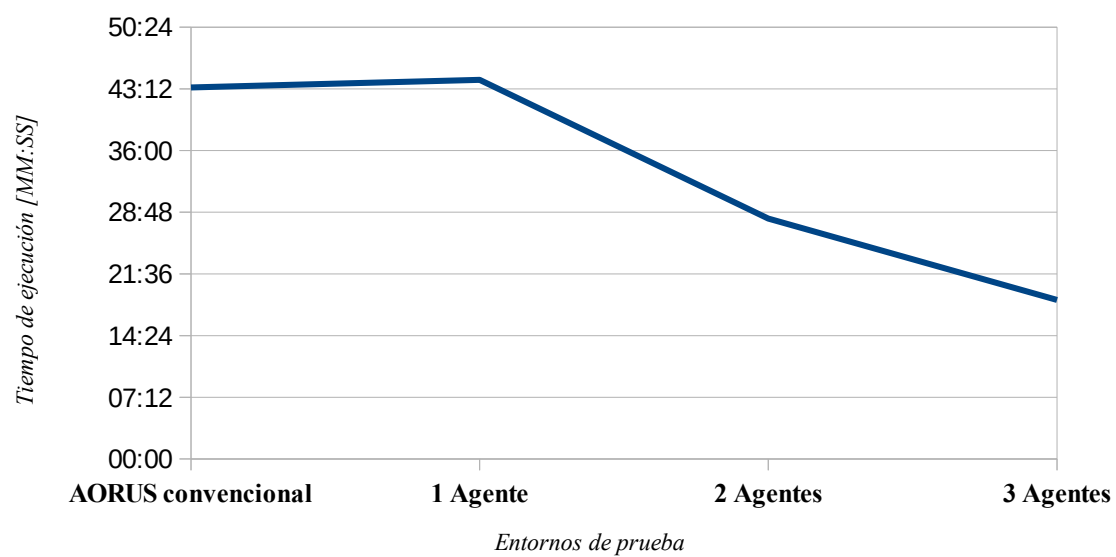


Figura 41: Comparativa de tiempos de ejecución del sistema Liri en cada entorno de prueba sobre la aplicación Consul.

Fuente: Elaboración propia.

Anexo G – Fórmula para el cálculo del tamaño de los conjuntos

Como se observó en la Figura 31, al utilizar los tamaños de conjunto 30, 50 y 100 se obtuvieron tiempos de ejecución muy similares en cada entorno de prueba.

Se realizaron varias operaciones matemáticas que incluyen como parámetros: la cantidad de archivos de prueba y la cantidad de Agentes, con el objetivo de obtener valores cercanos a los 3 tamaños de conjunto mencionados anteriormente, y, de este modo obtener una fórmula con la cual se puedan calcular un rango al cual pertenezcan los valores 30, 50 y 100.

Luego de probar varias operaciones se llegó a la siguiente fórmula:

$$\text{Tamaño de conjuntos} = \left\lceil \frac{\left(\frac{\text{Cant. de archivos de prueba}}{\text{Cant. de Agentes}} \right)}{x \in L = \{4, 5, 6, 7\}} \right\rceil \quad (\text{A.1})$$

El valor de x puede ser cualquiera de los valores del conjunto L. Si el resultado de la operación es un valor decimal, se le aplica la función techo para redondear a un valor entero.

En la Tabla 35 se puede observar que aplicando la fórmula para cada entorno de prueba de la Tabla 6, excepto al entorno A, se obtienen valores en un rango de 30 a 102 y en este rango entran los tamaños de conjunto: 30, 50 y 100 utilizados para las pruebas del sistema Liri. El entorno A incluye un sólo Agente, para este caso es mejor indicar un tamaño de conjuntos igual a la cantidad de archivos o usar el método convencional de ejecución sin Liri.

Entorno B 2 Agentes	Entorno C 3 Agentes	Entorno D 4 Agentes
$(816 / 2) / 4 = 102$	$(816 / 3) / 4 = 68$	$(816 / 4) / 4 = 51$
$(816 / 2) / 5 = 82$	$(816 / 3) / 5 = 55$	$(816 / 4) / 5 = 41$
$(816 / 2) / 6 = 68$	$(816 / 3) / 6 = 46$	$(816 / 4) / 6 = 34$
$(816 / 2) / 7 = 59$	$(816 / 3) / 7 = 39$	$(816 / 4) / 7 = 30$

Tabla 35: Aplicación de la fórmula para (A.1) a los entornos de prueba del Sistema Liri, excepto el entorno A.

Fuente: Elaboración propia.

Para utilizar el sistema Liri, es necesario indicar el tamaño de los conjuntos a ser ejecutados por las Aplicaciones Agentes, por este motivo, la fórmula (A.1) se propone como una guía para que el usuario potencial del sistema Liri tenga de donde partir al momento de indicar el tamaño de conjunto que va a utilizar.

Cabe aclarar que para asegurar la precisión de la fórmula (A.1) es necesario hacer pruebas más exhaustivas con aplicaciones que tengan diferentes cantidades de archivos de

prueba y entornos con múltiples Agentes, pero, comprobar la precisión de la fórmula presentada no forma parte de los objetivos del presente TFG.


Anexo H – Autorización de uso del TFG

Autorización de utilización para los derechos de autor

Autorizamos a que este Trabajo Final de Grado, pueda servir como base para otros trabajos, artículos o ponencias, siempre y cuando se reconozcan nuestros derechos como autor.



.....
Leslie López



.....
Rodrigo Fernández

Fecha: **26/08/2022**