UNIVERSIDAD NACIONAL DE ITAPÚA

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

Rodrigo Jacinto Fernández Ojeda - Leslie Fabiana López Figueredo

Encarnación - Paraguay
2022

UNIVERSIDAD NACIONAL DE ITAPÚA

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

Rodrigo Jacinto Fernández Ojeda - Leslie Fabiana López Figueredo

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.

Tutor: Ing: Aldo Miguel Medina Venialgo

Encarnación - Paraguay
2022

UNIVERSIDAD NACIONAL DE ITAPÚA

FACULTAD DE INGENIERÍA

Ingeniería Informática

HOJA DE EVALUACIÓN DE TFG

INTEGRANTES DE LA MESA EXAMINADORA

- _____
- _____
- _____
- _____
- _____

CALIFICACIÓN FINAL: ___ (_____)

ACTA Nº:_____

FECHA: _____

_____ _____

Secretaria General **Decano**

Encarnación - Paraguay
2022

Dedico este trabajo a mi familia y a mi esposo por estar siempre a mi lado, por la confianza y el apoyo incondicional que me brindaron, sin ellos no hubiera sido posible cumplir esta meta.

Leslie López

Dedico este trabajo a mis padres Silvino y Juana por haberme apoyado durante toda la carrera, y vaya que me tomé mi tiempo, y a mi esposa Maga por estar más emocionada que yo por la culminación de este trabajo.

Rodrigo Fernández

AGRADECIMIENTOS

ÍNDICE

TABLA DE FIGURAS

LISTA DE TABLAS

ÍNDICE DE ANEXOS

LISTA DE ABREVIATURAS

ATDD Desarrollo Orientado a Pruebas de Aceptación (Acceptance Test Driven Development)

BDD Desarrollo Guiado por el Comportamiento (Behavior Driven Development)

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)

SLTC Modelo de Ciclo de Vida de Desarrollo de Software (Software Development Life Cycle Model)

TFG Trabajo Final de Grado

RESUMEN

En la actualidad las pruebas unitarias son una herramienta fundamental para el aseguramiento de la calidad del software. A medida que se avanza en el desarrollo de un software, la cantidad y complejidad de las pruebas unitarias aumentan y los recursos hardware del desarrollador son casi siempre las mismas que al inicio del proyecto, esto podría incidir negativamente en el tiempo de ejecución del conjunto de pruebas unitarias.

En el presente trabajo final de grado se desarrolló un sistema que permitirá al desarrollador de software, ejecutar pruebas unitarias de código para aplicaciones desarrolladas en Ruby on Rails utilizando la librería RSpec, el sistema soporta una arquitectura distribuida, es decir, la carga de trabajo que conlleva la ejecución de estas pruebas será distribuida en una red de computadoras con el fin de descentralizar la carga de trabajo que conlleva la ejecución de las pruebas unitarias, aumentando la velocidad de ejecución y reduciéndose el tiempo requerido para la obtención de resultados de la ejecución de las pruebas unitarias.

El sistema desarrollado está compuesto por dos aplicaciones, 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 de esta ejecución. La Aplicación Agente es un servicio en segundo plano, que puede ejecutarse en varias computadoras dentro de una red, y se encarga de ejecutar las pruebas unitarias y retornar los resultados a la Aplicación Coordinadora.

Durante su ejecución, la Aplicación Agente inicia un servidor UDP que se mantiene en espera, mientras que, la Aplicación Coordinadora inicia un cliente UDP y un servidor TCP. El cliente UDP de la Aplicación Coordinadora emite peticiones de difusión (Broadcast) a la red, estas peticiones son procesadas por el servidor UDP de las Aplicaciones Agente que se encuentren en ejecución. Cuando una Aplicación Agente recibe la petición UDP, procede a realizar dos operaciones principales: primero, se conecta a través del protocolo SCP a la computadora donde está ejecutándose la Aplicación Coordinadora y obtiene el código fuente de la aplicación cuyas pruebas unitarias se quiere ejecutar; segundo, inicia un cliente TCP que se conecta con el servidor TCP que se encuentra ejecutándose en la Aplicación Coordinadora, y mediante esta conexión se procede a coordinar la ejecución del conjunto de pruebas unitarias.

A partir de las pruebas llevadas a cabo sobre el sistema implementado, se logró mejorar el tiempo de ejecución de un conjunto de pruebas unitarias usando la arquitectura distribuida propuesta, en comparación a la ejecución de este mismo conjunto de pruebas en una sola computadora.

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

ABSTRACT

Currently, unit tests are a fundamental tool for software quality assurance. As software development progresses, the number and complexity of unit tests increase and the developer's hardware resources are always the same as at the beginning of the project, this could negatively affect the execution time of the set of tests.

In this final degree project a system was presented that allowed the software developer to execute unit code tests for applications developed in Ruby on Rails using the RSpec library, the system supported by a distributed architecture, that is the workload that entails the execution of these tests will be distributed in a computer network in order to decentralize the workload involved in the execution of the unit tests, increasing the execution speed and reducing the time required to obtain the results of the execution of the unit tests.

To implement the System, different tools were used, such as TCP and UDP sockets for the automatic detection and connection between the different agent machines and the main master machine, the SCP protocol for sending the project in compressed form (zip) and the Rspec gem for running unit tests.

Keywords: Unit tests, Rspec, distributed architecture, agents, master.

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)

Para desarrollar un software, 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 Ciclo de Vida de Desarrollo de Software (SDLC, del inglés Software Development Life Cycle) para lograr una mejor situación de desarrollo y éxito económico de las mismas. 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 varias etapas como lo son :

- 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 mencionadas anteriormente 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 a éstas como lo son la planificación, despliegue y mantenimiento son necesarias para el desarrollo de un software porque la misma conlleva unos costos asociados, por lo que se hace necesaria la planificación y lo que se pretende una vez construido el software, es que éste debería poder utilizarse (despliegue y mantenimiento) (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: prueba de integración, pruebas de regresión, pruebas de humo, pruebas del sistema y pruebas unitarias, (Pauta Ayabaca y Moscoso Bernal, 2017).

En la etapa de 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 compilación 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. Éste 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). Éstos 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 & 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). 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.

	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.*
Elaboración propia.

En nuestra experiencia, cuando el desarrollador requiere ejecutar todas las pruebas unitarias de manera local, el tiempo que lleva la ejecución de estas pruebas unitarias está limitado por las características de procesamiento de su computadora. Es por ello que a medida que va creciendo el software, aumentan la cantidad de pruebas unitarias, por ende, el tiempo de ejecución de las pruebas unitarias también aumentan.

En la Tabla 2 se presenta una comparativa de algunas herramientas de integración continua que mejoran la velocidad de ejecución de pruebas unitarias. Dichas herramientas consisten en la fusión del trabajo de los desarrolladores, permitiendo mejorar la calidad del software df (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).

Como se mencionó anteriormente la integración continua ayuda mejorar la velocidad de la 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.

	Interfaz	Lenguajes Soportados	Instalación	Licencia
Jenkins	Web	Varios	Linux, Windows, OS X	MIT, Código Abierto
TeamCity	Web	Varios	Linux, Windows, OS X	Comercial, gratis bajo ciertas condiciones
Travis CI	Web	Varios	Servicio en la nube	Comercial, gratis bajo ciertas condiciones
Circle CI	Web	Varios	Servicio en la nube y servidor privado en Linux u OS X	Comercial
Codeship	Web	Varios	Servicio en la nube	Comercial
Dist Test	Web	Java	Linux	Apache License 2.0

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

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 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 código para aplicaciones Ruby on Rails utilizando la librería RSpec. 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 en una red de computadoras.

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 irán mostrando los resultados en la interfaz de línea de comandos, indicando las pruebas ejecutadas con éxito o que hayan fallado.

El sistema puede obtener el código fuente sobre el cual se ejecutarán las pruebas unitarias desde dos ubicaciones posibles, una de ellas es la computadora del desarrollador y la otra será desde un repositorio Git.

El sistema soporta la ejecución de pruebas unitarias que requieren conexión a base de datos. Al iniciar un proceso de ejecución de pruebas se crea una base de datos de prueba y al finalizar el proceso se elimina la base de datos, cada proceso trabaja sobre una base de datos diferente.

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 servicio en segundo plano que 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.

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

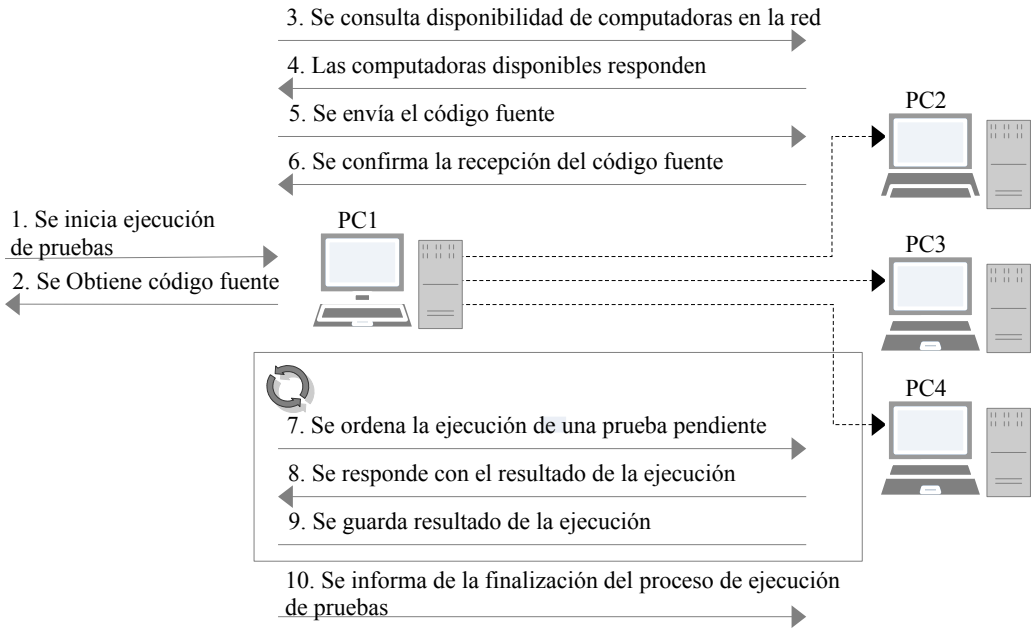


Figura 1: *Prototipo de la Arquitectura del Sistema Implementado.*
Elaboración propia.

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 todas, por este motivo los desarrolladores no se animaban a realizar 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, el Sistema desarrollado en el marco de este TFG tendrá las siguientes ventajas:

- 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 descargado del repositorio, así también sobre el código del desarrollador utilizando la misma arquitectura distribuida mencionada en el punto anterior.

• La cantidad de pruebas unitarias a ejecutar en cada computadora involucrada será determinada de manera automática, en comparación a las herramientas de integración continua existentes que requieren la determinación manual de la cantidad de pruebas a ejecutar en cada computadora cuando realizan una ejecución distribuida.

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

Luego de la planificación lo siguiente que hay que hacer para construir un software es averiguar qué es exactamente lo que tiene que hacer el mismo (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 los modelos utilizados en la etapa de análisis representan los requisitos del usuario desde distintos puntos de vista (el qué), los modelos que se utilizan en la fase de **diseño** representan las características del sistema que nos permitirán implementarlo de forma efectiva (el cómo). (Berzal, 2004, p. 12)

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

Una vez que se analizó y diseñó lo que se espera que haga el software, es el momento de pasar a la etapa de **implementación**. En esta etapa 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 ésta 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 (...), bibliotecas y componentes suministrados por terceras partes, etc.". (p.21)

Dada la naturaleza del software, que ni 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 desti-

nado 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 se pretende indicar 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 sigue la siguiente secuencia de fases: análisis, diseño, código y prueba.

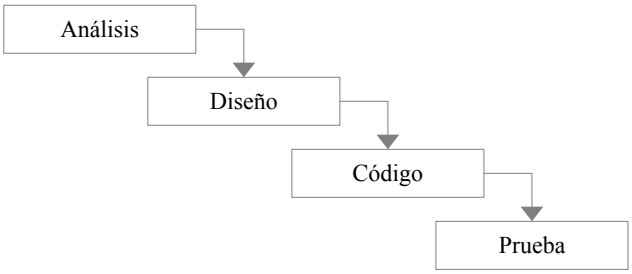


Figura 2: *Modelo en Cascada.*
Adaptado de El modelo lineal secuencial (p. 20), por Roger S. Pressman, 2002, McGraw-Hill Interamericana. .

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.

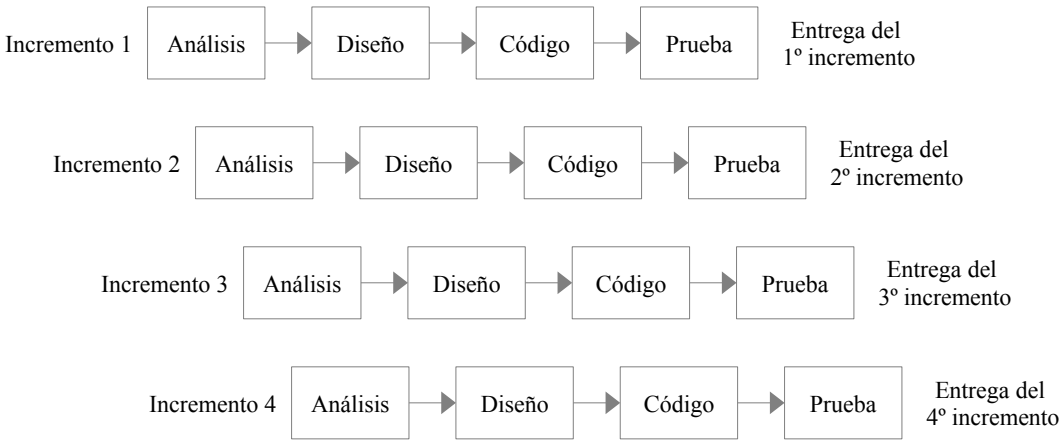


Figura 3: *Modelo Incremental.*
Adaptado de El modelo incremental (p. 24), por Roger S. Pressman, 2002, McGraw-Hill Interamericana.

En el modelo incremental, el primer incremento suele ser un producto esencial, por lo que se cubren sólo 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).

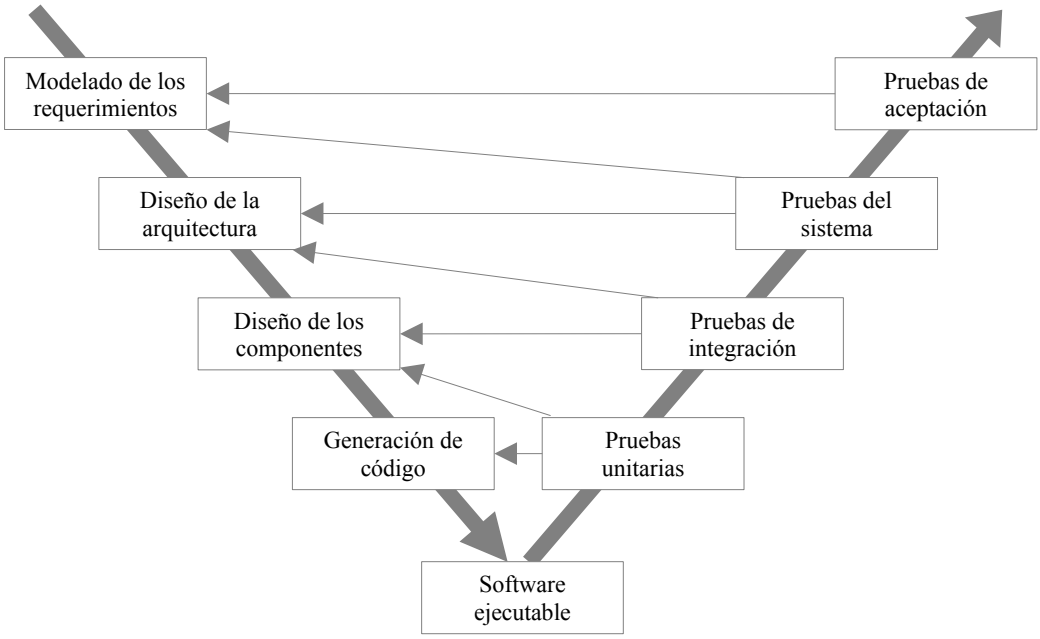


Figura 4: *Modelo V.*
Adaptado de El modelo en V (p. 35), por Roger S. Pressman, 2010, McGraw-Hill Interamericana.

2.2 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).

2.2.1 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)

2.2.2 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 et al., 2020, p. 7).

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 Srint 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) mencionan 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

"El 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)

2.2.3 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).

Cada artefacto contiene un compromiso:

- Product Backlog, es el Objetivo del Producto.
- Para el Sprint Backlog, es el Objetivo del Sprint.
- Para el Increment es la Definición de Terminado. (pp. 10-11)

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 Dueño de Producto (Product Owner) es el responsable de la Lista de Producto, incluyendo su contenido, disponibilidad y ordenación.

La Lista de Producto es dinámica; cambia constantemente para identificar lo que el producto necesita para ser adecuado, competitivo y útil. Mientras el producto exista, la lista de producto también.

La Lista de Producto 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 de la Lista de Producto 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)

2.3 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) .

2.3.1 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 las 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 probadores puedan realizar todas las pruebas posibles y encontrar todos los defectos posibles, pero los principios 2 y 1, 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.

2.3.2 Métodos De Testeo

• **Caja Negra:** Es la técnica de prueba sin tener ningún conocimiento del funcionamiento interno de la aplicación. El probador ignora la arquitectura del sistema y no tiene acceso al código fuente. Por lo general, al realizar una prueba de caja negra, un probador 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 probador una ventaja sobre alguien con conocimiento limitado del dominio. A diferencia de las pruebas de caja negra, donde el evaluador solo prueba la interfaz de usuario de la aplicación, en las pruebas de caja gris, el evaluador tiene acceso a los documentos de diseño y la base de datos. Con este conocimiento, el probador puede preparar mejor los datos de prueba y los escenarios de prueba al hacer el plan de prueba (tutorialspoint.com, 2011).

• **Caja Blanca:** La prueba de 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 evaluador debe tener conocimiento del funcionamiento interno del código. El evaluador

debe echar un vistazo al código fuente y descubrir qué unidad/fragmento del código se está comportando de manera inapropiada (tutorialspoint.com, 2011).

2.3.3 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 (Smoke Testing o Ad Hoc):** 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 giro 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, volumen, esfuerzo (Stress), recuperación y múltiples sitios. (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.*
Adaptado de "Guía de Testing de Software: Principales tipos de Pruebas de Software" por A. C. Rocha,2010, p. 1 (https://gtsw-es.blogspot.com/2010/12/principales-tipos-de-testing-de.html)

2.4 DESARROLLO DIRIGIDO

2.4.1 Desarrollo Dirigido Por Pruebas De Aceptación (ATDD)

El desarrollo guiado por pruebas de aceptación (ATDD) 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 un taller de especificación. El taller 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).

2.4.2 Desarrollo Dirigido Por Comportamiento (BDD)

BDD (Behaviour-Driven Development) comienza desde el punto de vista de que los comportamientos del software son más fáciles de entender para las partes interesadas cuan-

do 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 un taller 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 no solo estén en la misma página, sino 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).

Dan North, 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.

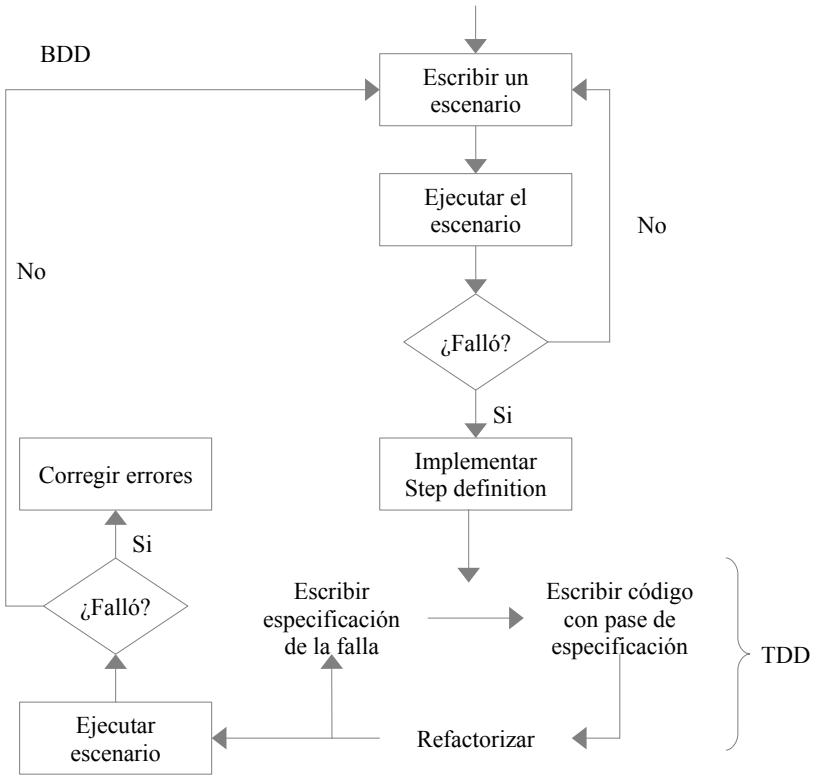


Figura 5: *Desarrollo Guiado por Comportamiento.*
Adaptado de Estructura del Desarrollo Guiado por Comportamiento (p. 196), por Aldo Emanuel Soraluz Soralus, Miguel Ángel Valles Coral y Danny Lévano Rodríguez, 2021, Ingeniería y Desarrollo.

2.4.3 Desarrollo Dirigido Por Pruebas (TDD)

El desarrollo basado en pruebas es un método mediante el cual se crean pruebas unitarias, en pequeños pasos incrementales y, en los mismos pequeños pasos incrementales, se crea el código para cumplir con esas pruebas (Black et al., 2017).

Estas pruebas unitarias permiten a los desarrolladores de software 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 solo 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 necesita refactorizar el código con frecuencia. La fortaleza de TDD es que solo desarrolla el código mínimo que se requiere para pasar las pruebas unitarias existentes, y solo luego pasa a la siguiente prueba y código (Black et al., 2017).

2.5 PRUEBAS UNITARIAS

Una prueba unitaria procura una "unidad" de código de forma aislada y compara los resultados reales con los esperados. Las pruebas unitarias invocan uno o más métodos de una clase 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:

- Debe ser automatizado y repetible.
- Debe ser fácil de implementar.
- Debe ser pertinente.
- Cualquiera debería poder ejecutarlo con solo presionar un botón.
- Debe ejecutarse rápidamente.
- Debe ser coherente en sus resultados (siempre devuelve el mismo resultado si no cambia nada entre ejecuciones).
- Debe tener el control total de la unidad bajo prueba.
- Debe estar completamente aislado (se ejecuta independientemente de otras pruebas).

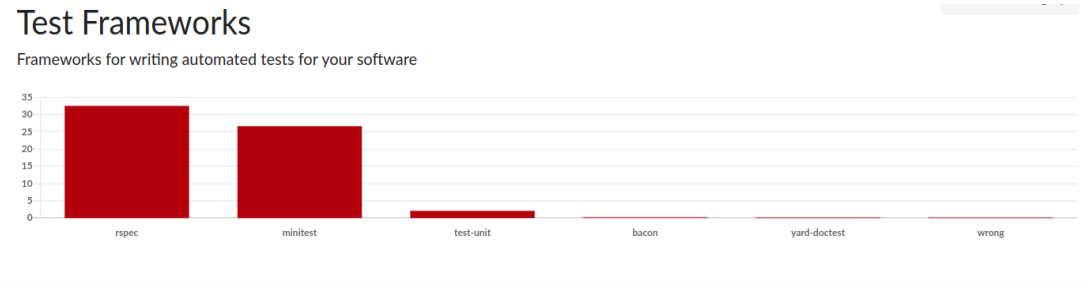
• Cuando falla, debería ser fácil detectar lo que se esperaba y determinar cómo identificar el problema.

2.5.1 Librerías Para Testeo Unitario En Diferentes Lenguajes

En la Tabla 1 se presentaron algunas librerías o frameworks que permiten implementar pruebas unitarias.

2.5.2 Frameworks De Pruebas Unitarias En Ruby

En Ruby tenemos una gama amplia de frameworks para pruebas unitarias, cada una de ellas con sus pros y contras. En la Figura 6 podemos ver el ranking de las librerías más utilizadas en los últimos tiempos.



Full | Compact | Table | Order by **Project Score**

	PROJECT SCORE	DOWNLOADS	STARS	FORKS	FIRST RELEASE	LATEST RELEASE	REVERSE DEPENDENCIES
rspec	★ 32.52	★ 592,017,445	2,839	233	2005-08-11	★ 2022-02-09	★ 56,444
minitest	26.63	473,286,466	★ 2,986	★ 479	2008-10-09	2021-12-15	11,824
test-unit	2.11	35,850,789	242	89	2008-03-20	2021-12-19	2,768
bacon	0.25	368,364	424	56	2008-01-06	2012-12-21	297
yard-doctest	0.22	3,169,720	94	15	2014-06-16	2019-09-12	105
wrong	0.21	358,627	434	26	2010-07-06	2013-11-11	56

Figura 6: *Ranking de frameworks de pruebas unitarias en Ruby.*
Datos expresados de acuerdo a la puntuación del proyecto. Test frameworks, 2022. https://www.ruby-toolbox.com/categories/testing_frameworks?display=table&order=score

2.5.3 Comparativa Entre Rspec Y Minitest

MiniTest: es una herramienta de prueba para Ruby que proporciona un conjunto completo de instalaciones de prueba. También es compatible con el desarrollo basado en el comportamiento, la simulación y la evaluación comparativa (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 testeando.
> - **Setup:** Bloque de código que se ejecuta **antes de correr cada test y** una vez por cada uno de ellos, para configurar un entorno común.
> - **Teardown:** Bloque de código que se ejecuta **luego de correr cada test**, y una vez por cada uno de ellos. Se utiliza para realizar una limpieza post-test.

Rspec: Es una herramienta de desarrollo dirigida por el comportamiento para programadores de Ruby. BDD es un enfoque para el desarrollo de software que combina el desarrollo dirigido por pruebas, el diseño dirigido por dominios y la planificación dirigida por pruebas de aceptación (“RSpec”, 2022).

> 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, y permite utilizar diferentes descripciones y contextos para clarificar las incumbencias de aquello que se está testeando.
>
> (Hourquebie, 2018)

2.6 PROTOCOLOS

2.6.1 Función De La Capa De Transporte

Según Cortés (2008), la capa de transporte es la encargada de entablar o establecer una sesión de comunicación entre dos softwares, y a su vez poder transmitir datos entre sí. Existen 2 tipos de protocolos de comunicación:

- Protocolo de control de transmisión (TCP)
- Protocolo de datagramas de usuario (UDP)

2.6.2 Protocolo De Control De Transmisión (TCP)

El protocolo de control de transmisión ('Transmission Control Protocol', TCP) 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).

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 datagramas, 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 es un protocolo que 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 enviará datos sin esperar la confirmación de los bytes ya enviados. (pp. 21-22)

Para lograr todos los puntos de fiabilidad citados anteriormente TCP posee una cabecera un poco más complicada (Ver Figura 7).

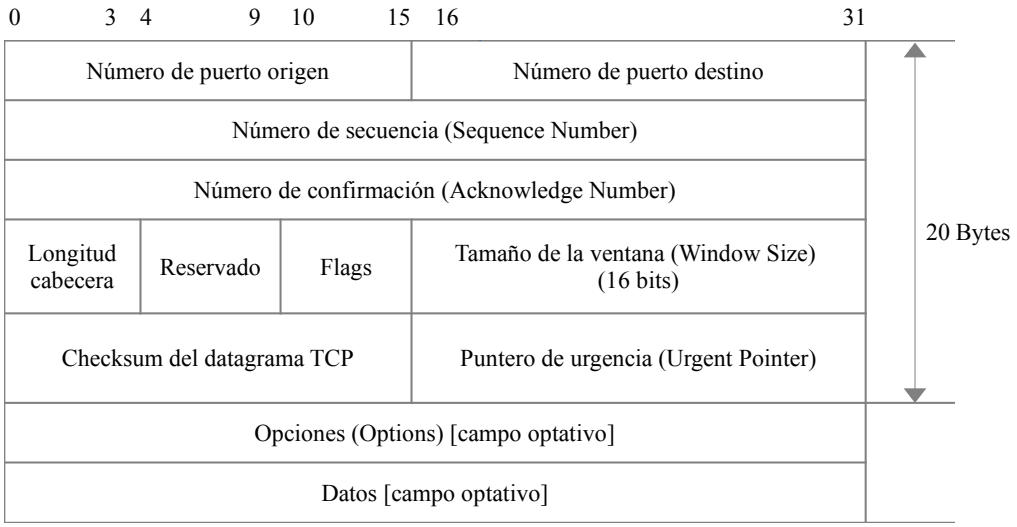


Figura 7: *Cabecera de un segmento TCP.*
Adaptado de Cabecera de un segmento TCP (p. 23), por Gabriel Verdejo Alvarez, 2003, Universitat Autònoma de Barcelona.

2.6.3 Protocolo De Datagramas De Usuario (UDP)

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 al llegar un mensaje, porque como se mencionó anteriormente las mismas no están garantizadas. UDP es mucho más simple en su construcción, pero mantiene la lógica de añadir un encabezado al mensaje pero mucho más corto que del TCP (Pineda y Méndez, 2018).

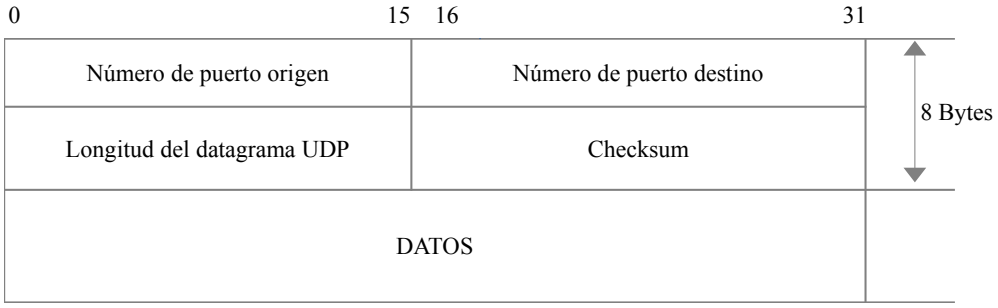


Figura 8: *Cabecera de un datagrama UDP.*
Adaptado de Cabecera de un datagrama UDP (p. 20), por Gabriel Verdejo Alvarez , 2003, Universitat Autònoma de Barcelona.

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).

2.6.4 Protocolo De Copia Segura (SCP)

La función Protocolo de copia segura (SCP) proporciona un método seguro y autenticado para copiar configuraciones de conmutadores o archivos de imágenes de conmutadores. SCP se basa en Secure Shell (SSH)

2.7 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 (...).

Un socket es un punto final de un enlace de comunicación de dos vías entre dos programas que se ejecutan a través de la red. (p. 49)

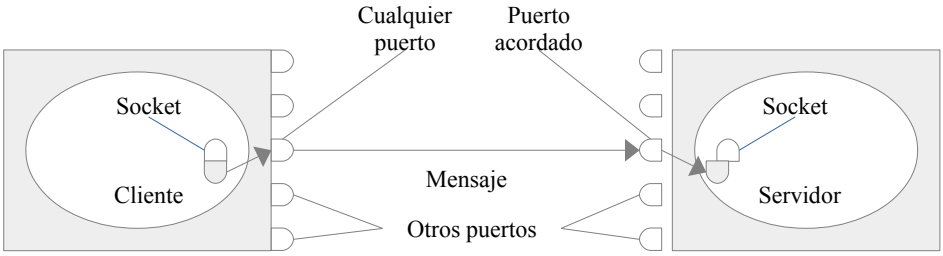


Figura 9: *Comunicación entre sockets*.
Adaptado de Sockets y puertos (p. 120), por George Coulouris, Jean Dollimore y Tim Kindberg, 2001, Pearson Education.

2.7.1 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.

- **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)

2.7.2 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).

El servidor es el encargado de responder las peticiones de uno o más clientes (Ver Figura 10)

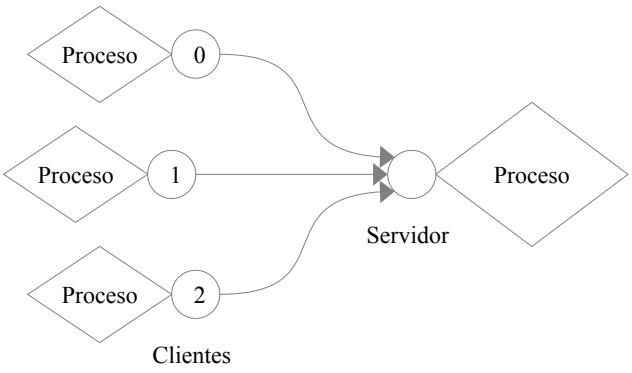


Figura 10: *Ejemplo básico de Cliente-Servidor.*
Adaptado de Ejemplo de Cliente Servidor (p. 3), por David Pineda y Daniel Méndez , 2018, Universidad de Chile.

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)

2.8 DIFUSIÓN DE DATOS

2.8.1 Unidifusión (Unicast)

El proceso de unidifusión es 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).

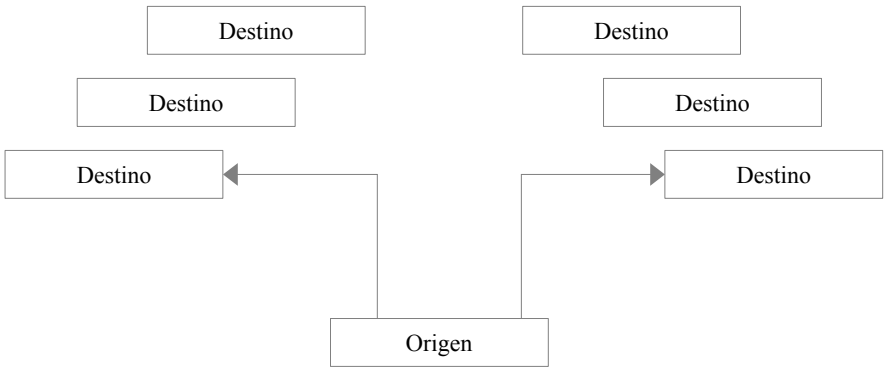


Figura 11: *Envío de paquete, utilizando la técnica Unicast.*
Adaptado de Unicast message Transmission in MANETs (p. 16), por Shivraj Kaur, Kulwinder Singh y Yadvinder Singh , 2016, International Journal of Computer Applications.

2.8.2 Multidifusión (Multicast)

Primeramente, es importante definir IGMP (Internet Group Managment Protocol) IGMP es un protocolo usado para registrar de forma dinámica cada uno de los hosts de un grupo multicast en una determinada LAN. Los hosts anuncian su pertenencia a grupos enviando mensajes IGMP a su router multicast local. Los routers con IGMP habilitado procesan los mensajes IGMP recibidos, y además periódicamente envían peticiones para descubrir qué grupos están activos o inactivos en una subred en concreto.

Ya que definimos IGMP podemos proseguir con Multicast. Multicast se basa en el concepto de un grupo. Un grupo de multidifusión es un grupo arbitrario de receptores que expresa su interés en recibir un flujo de datos en particular. Este grupo no tiene límites, las físicas o geográficas anfitriones pueden ubicarse en cualquier lugar de Internet o cualquier red interna privada. Los anfitriones que estén interesados en recibir los datos que fluyen a un grupo en particular deben unirse al grupo utilizando IGMP.

2.8.3 Difusión (Broadcast)

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

Si el paquete tiene una dirección broadcast, todos los dispositivos que reciban ese mensaje lo procesarán (Ver). Entonces, todos los dispositivos en el mismo segmento de red verán el mismo mensaje (Cicnavi, 2011).

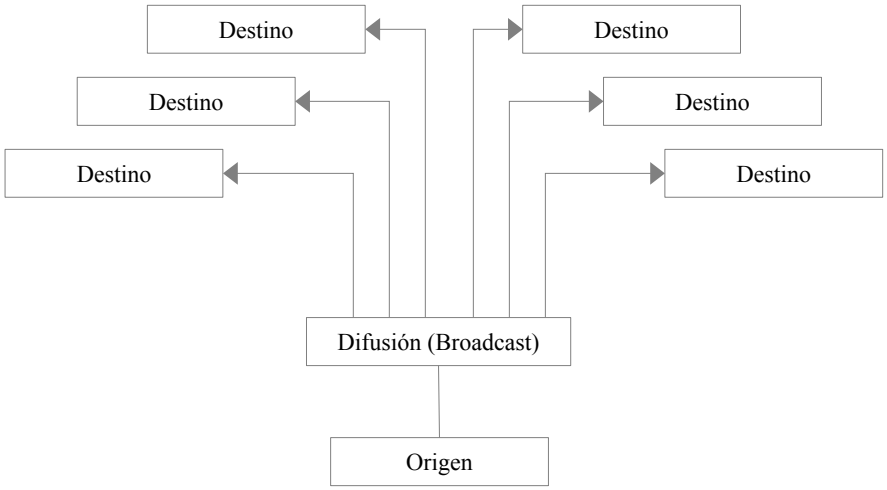


Figura 12: *Envío de paquete utilizando Broadcast.*
Adaptado de Broadcast message Transmission in MANETs (p. 17), por Shivraj Kaur, Kulwinder Singh y Yadvinder Singh , 2016, International Journal of Computer Applications.