UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

TITULACIÓN POR TRABAJO PROFESIONAL

DISEÑO Y DESARROLLO DE LA INFRAESTRUCTURA DE PRUEBAS AUTOMATIZADAS PARA EL SISTEMA DE NOTIFICACIÓN DE EVENTOS DEL CLUSTER

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

PRESENTA:

FÉLIX EDUARDO MORALES MENDOZA

DIRECTOR DE TESIS:

M.C. ALEJANDRO VELÁZQUEZ MENA

CIUDAD UNIVERSITARIA 2014

AGRADECIMIENTOS

Todo sistema de *software* requiere del trabajo de muchas personas para ser funcional y satisfacer las necesidades de la sociedad. De la misma manera este trabajo no hubiera sido posible sin las interminables horas de lectura y correcciones para las que me ayudaron varias personas, de cierta manera todas ellas fueron *testers* de mi documento sin saberlo.

Agradezco de manera muy especial a Erika Guzmán por haberme motivado insistentemente a terminar este proyecto, y haber estado a mi lado durante todas las etapas de su construcción.

La ayuda incondicional de mis padres y mi hermanito fue fundamental para todos los aspectos de mi desarrollo ya que siempre me proporcionaron a manos llenas, las herramientas necesarias para resolver los problemas que se me presentan día con día en la vida. Además de forjar en mi los valores que me han mantenido firme a mis convicciones.

Agradezco también a mis compañeros y amigos de la universidad por haberme hecho pasar momentos inolvidables, y hacer de ése periodo algo muy ameno.

Al departamento de servidores Unix del Instituto de Ingeniería de la UNAM, un especial agradecimiento por brindarme muchas de las herramientas prácticas que me han sido de gran utilidad durante mi periodo laboral en la empresa.

A mi *manager* de la empresa quien de manera proactiva, mantuvo un gran esfuerzo para garantizar la finalización de este documento.

A mi Facultad de Ingeniería de la UNAM.

A mi Universidad Nacional Autónoma de México.

¡GOYA! ¡GOYA!
¡CACHUN, CACHUN, RA, RA!
¡CACHUN, CACHUN, RA, RA!
¡GOYA!
¡¡UNIVERSIDAD!!

ÍNDICE DE CONTENIDOS

1. INTRODUCCIÓN

1.1. OBJETIVO

Lograr que el componente NRE *(Notificación Rápida de Eventos)* satisfaga estándares elevados de calidad durante todo el proceso de desarrollo de *software*, desde fases iniciales de diseño y planeación, hasta su entrega al usuario y soporte posterior a éste.

Mediante el diseño y desarrollo de casos de prueba basados en el análisis intensivo de todos los puntos sensibles del *software*. Se provee de una infraestructura con mecanismos automatizados que facilitan la detección y reporte detallado de anomalías que pudiesen tener un impacto negativo sobre el *software* y la empresa, así los líderes de proyecto y los desarrolladores pueden trabajar en conjunto para encontrar soluciones efectivas y eficientes. Es así como el cliente recibe un producto completamente funcional y confiable.

1.2. ACERCA DE LA EMPRESA

Se trata de una empresa multinacional de origen estadounidense con oficinas centrales en Redwood City, California, Estados Unidos y oficinas filiales en más de 150 países del mundo. Es la tercer compañía más grande de desarrollo de *software*. Está valuada en veintiséis mil millones de dólares con ciento treinta mil empleados y más de cincuenta mil clientes.

Desde finales de los años ochenta, ha sido líder en el ramo de desarrollo de sistemas de gestión de bases de datos, entre otras cosas, esto es debido a que el proceso que enmarca el desarrollo de *software* y *hardware* se rige dentro de rigurosos estándares de calidad que se han perfeccionado con el tiempo y que han requerido de una cautelosa planeación. Actualmente invierte 4.5 billones de dólares al año destinados al desarrollo e investigación.

Han sido introducidas grandes características gradualmente para que el manejador de bases de datos sea uno de los más confiables y robustos del mercado. Con la última versión lanzada al público, se incrementa cinco veces la escalabilidad de su versión anterior, además de que se implementan mejoras orientadas al *software* y a la infraestructura como servicios. Esto con el fin de satisfacer las necesidades tan cambiantes del mercado, que actualmente, exige soporte en el campo móvil, social y de manejo de grandes cantidades de información.

Además del sistema de gestión de bases de datos, que es el producto central y de más importancia para la empresa, la empresa también desarrolla otros productos de *software* como el segundo lenguaje de programación más usado en el mundo. Se cuenta también con una gran variedad propia de sistemas operativos de alto desempeño y soluciones de virtualización a diferentes escalas.

La solución de almacenamiento distribuido de alto rendimiento propia, está compuesta de por lo menos quince millones de líneas de código, corre de manera nativa en una gran cantidad de sistemas operativos como zOS, Linux, Windows, Mac OS, AIX, HP-UX, y es por mucho, la más estable del mercado.

Además de *software*, la compañía también desarrolla, en conjunto con otras, soluciones de *hardware* que resuelven las enormes exigencias emergentes de almacenamiento y procesamiento de datos. La combinación de *hardware* de alto rendimiento y *software* de almacenamiento de la empresa permiten realizar 1.5 millones de operaciones de entrada y salida en un segundo, reduce los costos de energía eléctrica en un 87.5 %, realiza consultas diez veces más rápido que una base de datos convencional y reduce el espacio físico de servidores hasta en un 75 %.

1.3. EL CENTRO DE DESARROLLO EN MÉXICO

En el año 2010, bajo la necesidad de acelerar y mejorar el proceso de desarrollo de la nueva versión del manejador de bases de datos, se abrió el centro de desarrollo en la ciudad de Guadalajara, Jalisco. Es importante puntualizar que aunque la empresa cuenta con más centros de desarrollo distribuidos en otros puntos geográficos, la creación de la nueva versión del manejador de base de datos se realiza únicamente en tres países: México, Estados Unidos e India.

El centro de desarrollo en México ha tenido un crecimiento constante, recientemente se contrató al empleado número 500, y se espera que en un futuro no muy lejano iguale en tamaño a otros centros como el localizado en Bangalore, India.

Al igual que con universidades como Stanford, Oxford o Yale, se han establecido convenios con diferentes universidades mexicanas cuyo fin es mejorar el proceso de aprendizaje de los alumnos, de otorgarles la oportunidad de tener un crecimiento profesional mediante programas de *internship*, y además, facilitarles la tarea de conseguir un empleo de clase mundial en desarrollo de software.

Una gran cantidad de productos de la empresa son desarrollados parcialmente en el centro de desarrollo en México, entre los que se encuentran: la base de datos en memoria, diversos sistemas de replicación, entre otros.

1.4. EL EQUIPO DE TESTING

El equipo de pruebas, al cual pertenezco, tiene miembros en los diferentes centros de desarrollo del mundo, incluyendo el de Estados Unidos, el de Bangalore, y finalmente en China.

El objetivo primordial del equipo es diseñar y desarrollar estrategias de *testing* para verificar y asegurar la calidad de los diferentes productos de la empresa.

En México el equipo está compuesto por una docena de integrantes, los cuales realizan labores de desarrollo de pruebas para el manejador de bases de datos, así como para los diferentes componentes del sistema de almacenamiento distribuido de alto rendimiento.

1.5. EL EQUIPO DE TESTING ORIENTADO A NRE

Al ser un producto tan grande y complejo, SAD (*Sistema de Alta Disponibilidad*) requiere de un gran equipo de trabajo para superar el reto que representa proveer al cliente de una solución de verdadera alta disponibilidad. Para cada componente de SAD hay al menos un equipo de desarrollo y de *testing*, es importante mencionar que así como los diferentes componentes de *software* están interconectados mediante complejos procedimientos, llamadas a funciones, y punteros a memoria compartida, el equipo de desarrollo debe, análogamente, tener una estructura que permita a sus miembros interactuar entre las diferentes áreas, y compartir información entre ellas de manera efectiva. Es por esto que todos los miembros del grupo de desarrollo deben tener excelente capacidad de comunicación, para lograr un eficaz trabajo en equipo.

El reto para el equipo de *testing* en este producto, es crear una plataforma que permita probar todas las capacidades de SAD, desde una perspectiva unitaria, donde se verifique la conducta particular de una función en un componente, hasta una más global en la que se compruebe el funcionamiento de un *cluster* con múltiples bases de datos, cientos de servicios, y cientos de nodos en diferentes arquitecturas y sistemas operativos. A lo largo del tiempo se han creado una gran cantidad de herramientas de *testing* que gradualmente han facilitado el soporte a productos con conductas cada vez más complejas, por ello hoy en día el *tester* sólo debe preocuparse de crear casos de prueba para las nuevas funcionalidades, y no tanto por las capas inferiores, a menos que se trate de un problema de compatibilidad o integración.

NRE es un componente que requiere herramientas altamente especializadas que verifiquen que los mensajes generados por las diferentes fuentes de eventos, como son el demonio SAC (*Servicio de Administración del Cluster*) y la base de datos, sean transmitidos, procesados, recibidos e interpretados correctamente por todos los clientes que estén suscritos al servicio.

Este conjunto de herramientas debe ser mantenido y actualizado de acuerdo a los lineamientos de funcionamiento de NRE, además de que debe proporcionar al programador una manera confiable para localizar anomalías en el código.

1.6. HABILIDADES DE UN DESARROLLADOR DE PRUEBAS DE BACKEND

1.6.1. HABILIDADES TÉCNICAS

- Capacidad de análisis en distintos niveles.
- Amplios conocimientos de ciencias de la computación, desde conceptos de programación y estructuras de datos, hasta sistemas de archivos y sistemas operativos.
- Dominio de varios lenguajes de programación y *scripting.*
- Conocimiento de diversas herramientas de programación, tales como *debuggers* y entornos integrados de desarrollo.
- Conocimiento y gran desenvoltura en diferentes sistemas operativos.
- Facilidad para identificar y abstraer cualquier patrón repetitivo a una tarea automatizada y totalmente independiente.
- Elección adecuada de las herramientas para resolver un problema en particular.
- Comprensión del código del equipo de *development* para poder señalar los errores y proponer soluciones.
- Capacidad de rápido aprendizaje de nuevas tecnologías.

1.6.2. HABILIDADES SOCIALES E INTELECTUALES

- Facilidad de comunicación tanto en español, como en inglés.
- Trabajo en equipo.
- Evaluación de la complejidad de tareas proporcionando tiempos esperados de entrega adecuados.
- Trabajo efectivo bajo presión.
- Facilidad de encontrar y reportar anomalías en el *software*, además de aportar información relacionada con las consecuencias.
- Exponer los riesgos de una manera que ayude a la empresa, al equipo de desarrollo y a los clientes.
- Fungir una conducta similar a la de un consejero.
- Elección de preguntas correctas e investigación de problemas correctos.
- Pensamiento crítico y altamente escéptico, ya que en cualquier lugar puede haber errores escondidos.
- Audacia para enfrentar situaciones que no habían sido realizadas con anterioridad.

James Bach, menciona en su sitio web que existen incluso tipos de *testers* dependiendo de sus habilidades y personalidad [Bach13].

Aparentemente, para que un equipo de *testers* sea exitoso, debe estar compuesto por desarrolladores de pruebas con cualidades distintas. La clasificación es la siguiente:

- **Administrativo**
  Se trata del tipo de *tester* que quiere que las tareas se realicen con puntualidad y forma, está más preocupado por cómo va el calendario que por el proceso técnico.

- **Técnico**
  Ellos conocen a detalle las pruebas y el código que se encuentran analizando, debido a esto están en constante comunicación con el desarrollador del producto y pueden entenderse en términos de código.
  Usualmente ellos mismos desarrollan las herramientas que necesitan y no les interesa tanto la teoría de *testing* ni ponen especial atención en el calendario. Son usualmente conocidos como *SDETs.*

- **Analítico**
  Tienen la tendencia a realizar un análisis muy complejo acerca de los problemas a los que se enfrentan, usan la matemática a su favor, realizan modelos apoyándose en diagramas y matrices. Leen documentos de especificación a detalle y, por lo general, tienden a realizar diseños de pruebas tan óptimos que pueden llegar a paralizar el proceso de planeación.

- **Social**
  Prefiere trabajar en equipo, saben que hay personas que ya realizaron una gran cantidad de tareas y que no es necesario crear una solución completa de *software*. Conocen a muchas personas en una gran cantidad de equipos y en caso de ser requeridos, son empleados para un proyecto en particular.

- **Usuario**
  Se trata de un usuario con gran experiencia en el producto, colabora con el equipo de *testing* a manera de consultor. Por lo general sólo conocen la conducta aparente del producto y no el funcionamiento lógico interno del sistema, pues no tienen acceso a ésta.

- **Desarrollador**
  El mismo desarrollador del producto puede fungir como *tester*, sin embargo, por lo general sólo realiza pruebas para comprobar que su modificación o componente en particular funciona como debe.

1.7. DESCRIPCIÓN GENERAL DEL CICLO DE DESARROLLO DE SOFTWARE EN LA EMPRESA

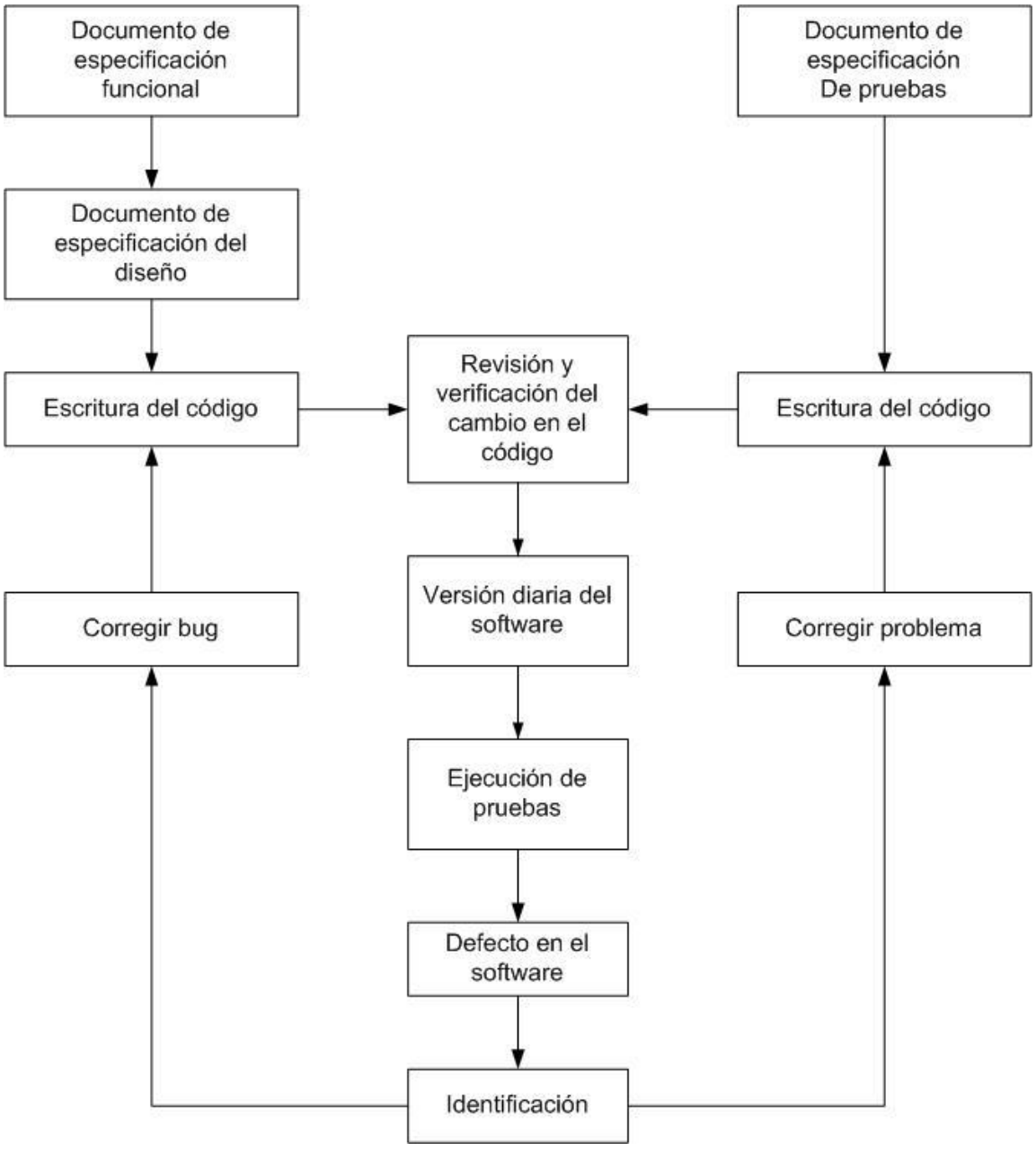

Fig. 1 : Diagrama a detalle del ciclo de desarrollo.

El proceso de desarrollo es iterativo y está orientado a realizar el proyecto de la manera más completa posible y sin fallos. Se pueden identificar dos grandes ramas, la de desarrollo del producto, y la de desarrollo de las pruebas. En la primera rama, el desarrollo está basado en los documentos de especificación funcional y estructural que fueron realizados cuidadosamente por un grupo de arquitectos de *software*, mientras que en la segunda, la de *testing*, sólo es necesario el documento de especificación de pruebas que fue creado por el desarrollador de pruebas con más experiencia, y en el que, con frecuencia, el arquitecto también contribuye directamente, pues tiene extensivos conocimientos de todas las funcionalidades del proyecto.

Una vez que el código ha sido escrito y ha pasado todas las pruebas de manera local, la transacción debe ser propuesta al comité que puede estar conformado por los arquitectos, gerentes de calidad y otros desarrolladores. Es recomendable usar una herramienta de revisión centralizada que brinde a los participantes un ambiente colaborativo en el que se puedan hacer comentarios en el código, asignar autoría y propiedad a los archivos, además de otorgar calificaciones.

Deben existir varios niveles de autorización y verificación de la transacción, en el primero se consideran aspectos de cumplimiento con los requerimientos en los diferentes documentos de especificación, seguimiento de las buenas prácticas de desarrollo, errores tipográficos y todo aquel factor que pudiera afectar sólo al área en el que se efectuó el cambio. En ésta primera fase participan los gerentes inmediatos, arquitectos del proyecto y desarrolladores en la misma área. Posteriormente en la segunda fase de aprobación, se requiere un jurado que verifique que los cambios no representan ningún riesgo a nivel de integración con las demás pruebas, además de verificar que se cumpla con cierto grado de calidad determinado por algún parámetro medible.

Una vez conseguida la autorización, el desarrollador integra sus cambios con los de muchos otros, el conjunto de todas las transacciones constituyen la versión del *software* para esa fecha. Debe crearse una versión del *software* de manera diaria. Posterior a su creación, se deben correr absolutamente todas las pruebas de todos los equipos para determinar que no haya nuevos fallos. Si los hay, deben ser identificados y encaminados al equipo y desarrollador correspondiente para ser corregidos. Los problemas sólo pueden ser de dos tipos, ya sea de código de producción, o bien, de pruebas, los cuales son representados por *bugs* y *test issues* respectivamente.

Una vez que la totalidad de las funcionalidades del proyecto han sido alcanzadas y se cuenta con un nivel de calidad elevado, el proyecto puede darse por concluido. Sin embargo, las pruebas siguen corriendo y es responsabilidad de los desarrolladores brindar el mantenimiento necesario.

2. ORGANIGRAMA

2.1. ORGANIGRAMA GENERAL DE LA EMPRESA

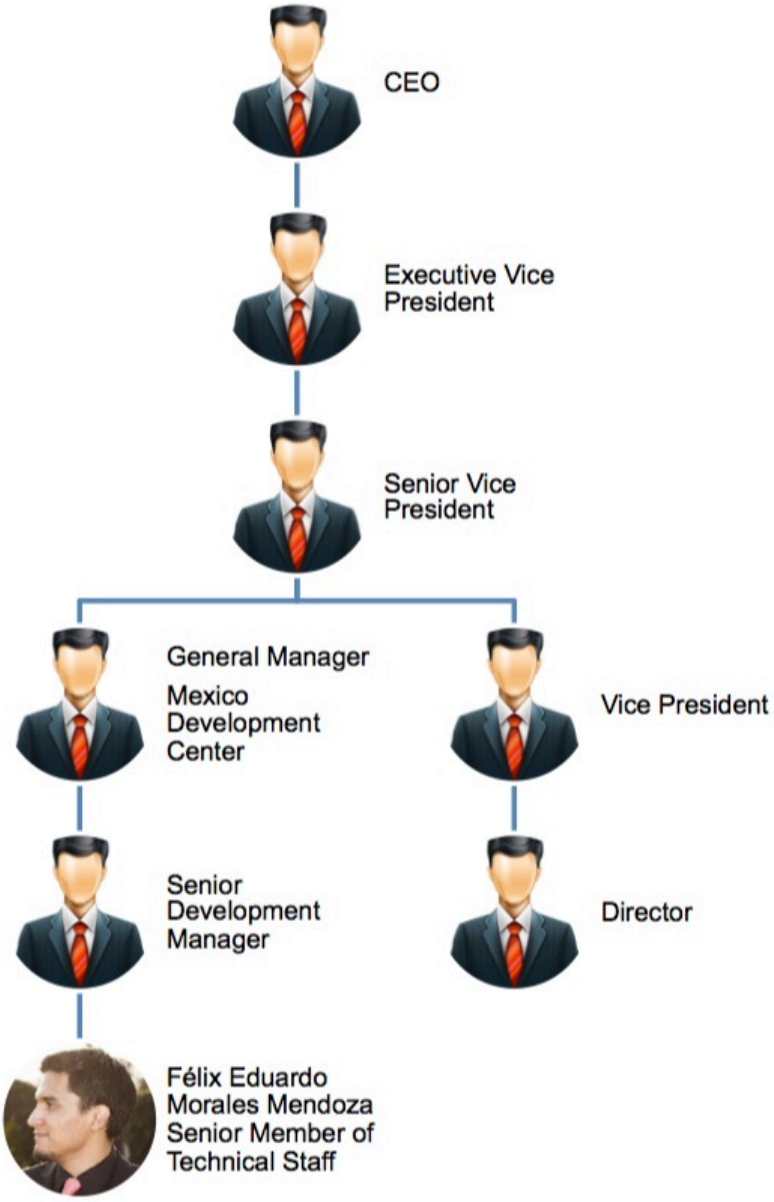

Fig. 2 : Esquema de la estructura orgánica.

3. DESCRIPCIÓN DE ACTIVIDADES

3.1. REPORTE DE BUGS DE CÓDIGO DE DESARROLLO Y PRUEBAS

Identifiqué una gran cantidad de fallos en los siguientes productos de la empresa: NRE, RUC (*Repositorio Universal de Conexiones*), base de datos y SN (*Servicios de Notificación*), todos ellos fueron catalogados apropiadamente y reportados con las respectivas partes involucradas, las cuales dieron seguimiento y realizaron la corrección necesaria.

3.1.1. METODOLOGÍA USADA

"Todo fallo corresponde exactamente a un aspecto que no fue especificado".
V.A. Vyssotsky, miembro de los laboratorios Bell.

Por lo general, un *bug* puede ser reportado por una gran cantidad de agentes que se pueden clasificar en dos ramas principales: los internos y los externos. Dentro de los internos se encuentran los desarrolladores de pruebas y del producto, arquitectos de software y otros empleados que estén relacionados con el área de desarrollo, así como el sistema automatizado de reporte de incidentes. Los agentes externos son los clientes que cuentan con el sistema instalado en un ambiente de producción. Un *bug* es localizado por la manifestación de un incidente que representa una funcionalidad faltante o errónea.

Una vez que el *bug* se manifiesta, es necesario identificar su naturaleza, para lo que se requiere recopilar la mayor cantidad de información respecto a las condiciones particulares en que se encontraba el *software* cuando ocurrió el fallo, como pueden ser variables de entorno, estado de los demonios, condiciones de carga en el sistema, etcétera. Además de los datos referentes al estado, es de gran importancia conocer los pasos exactos para reproducirlo.

Todos estos datos son introducidos en un sistema especializado en el almacenamiento de *bugs* con el fin de que el fallo se pueda resolver de la manera más rápida posible y de que se tenga un seguimiento eficaz. A menudo también es conveniente proporcionar un ambiente en el que el *bug* se esté ejecutando o bien, una carpeta con los resultados de la ejecución de una prueba.

Una vez que el *bug* fue reportado y asignado en el sistema, el desarrollador del área apropiada procede a resolverlo usando toda la información que se encuentra disponible en el reporte. Es importante mencionar que no hay una distinción entre *bugs* de código de pruebas y código de desarrollo respecto a la forma en cómo fueron reportados.

Además de fallos en el sistema, un *bug* puede ser reportado para solicitar nuevas funcionalidades importantes con el objetivo de tener un mejor seguimiento de la tarea.

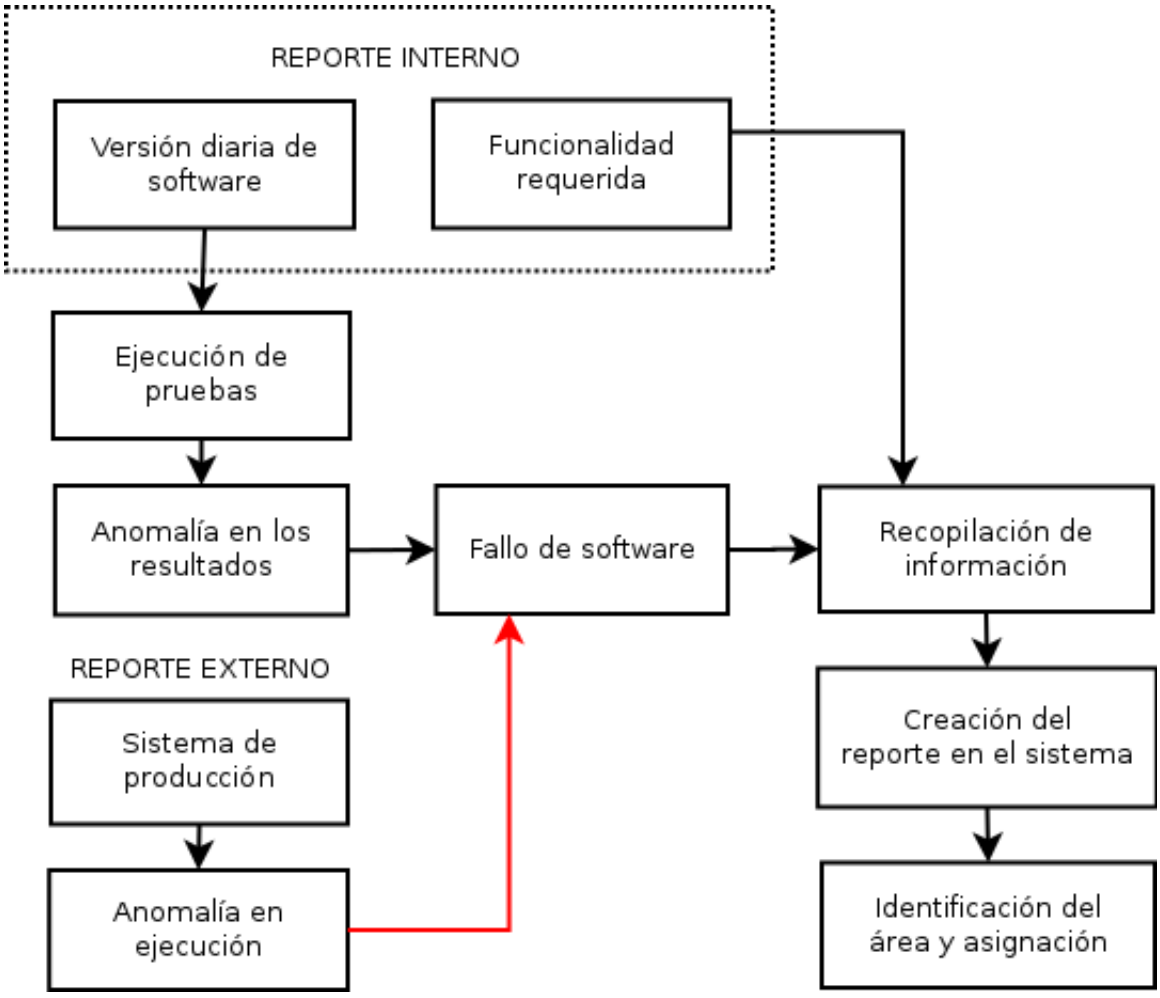

Fig. 3 : Metodología de reporte de *bugs.*

3.2. SOLUCIÓN DE BUGS EN CÓDIGO DE DESARROLLO Y PRUEBAS

Realicé la corrección de *bugs* en código de desarrollo y pruebas para varios componentes del sistema de *cluster* de la empresa entre los que destacan el sistema de notificación de eventos, y el sistema de escritura de bitácoras.

3.2.1. METODOLOGÍA USADA

La solución de un *bug* en código de desarrollo y pruebas depende completamente del tipo de problema, sin embargo la ruta más general es consultar la información del *bug* en el sistema y estar en contacto con el agente que lo haya notificado.

Se debe crear el ambiente siguiendo los pasos señalados en la descripción del *bug*, y proceder a identificar el problema usando la herramienta adecuada. Los fallos de software pueden aparecer de manera aislada o conjunta por lo que depende de la habilidad y criterio del desarrollador el determinar la solución adecuada.

Una vez que se propone una solución al problema es necesario corroborar que sea la que efectivamente resuelva la situación, esto se consigue ejecutando las pruebas que hayan fallado o que estén relacionadas con el incidente, y verificando que todas ellas hayan pasado con éxito.

La solución definitiva tiene dos métodos de entrega dependiendo de quién haya reportado el *bug*. Si el problema fue reportado por algún agente interno, es decir, un arquitecto de software, un gerente, o bien, otro desarrollador, se genera una transacción que se mezcla con la versión diaria del software. Por otro lado, si fue un cliente quien reportó el incidente, se le proporciona un parche que puede instalar en su sistema de producción. Al igual que en el reporte interno, el cambio también se mezcla con la versión diaria del software dependiendo de la severidad del problema.

Después de que la solución fue entregada, el sistema de almacenamiento de *bugs* se actualiza con la información de la solución y se cierra.

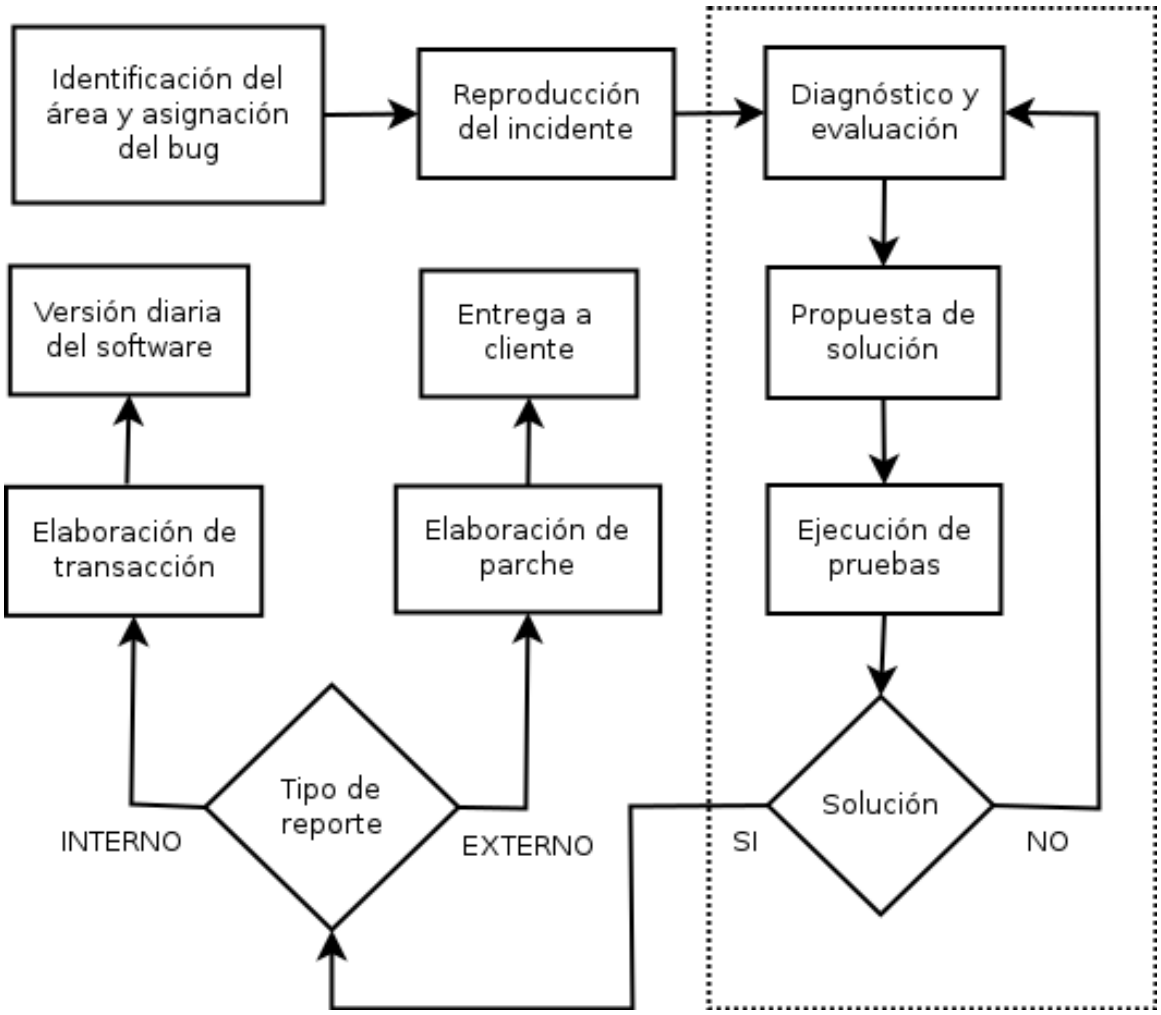

Fig. 4 : Metodología de solución de *bugs*.

3.3. DISEÑO DE NUEVOS CASOS DE PRUEBA PARA DIFERENTES COMPONENTES DE SOFTWARE

Realicé el diseño de una gran cantidad de pruebas para diversos componentes y funcionalidades tanto de NRE, como del sistema de escritura de bitácoras.

3.3.1. METODOLOGÍA USADA

El diseño de los casos de prueba se debe llevar a cabo por un grupo pequeño de desarrolladores que acuerden la forma en cómo serán implementadas, tomando en cuenta el enfoque de *testing*. Seleccionan las herramientas necesarias, las cuales incluso pueden no existir, y descartan factores que representan extrema complejidad. La identificación del enfoque de *testing* juega un papel fundamental debido a que si es elegido el caso de *testing* estructural, se usa el documento de especificación del proyecto para determinar los factores a verificar, por el otro lado, en el caso del *testing* funcional, el caso de prueba se diseña tomando en cuenta simplemente una conducta esperada.

Al final del proceso de diseño, los casos de prueba deben poseer las siguientes características [Kaner06]:

- Cada caso de prueba debe probar solamente un aspecto particular del código, que puede ir desde una función hasta un comportamiento esperado.
- Cuando un problema existe, la prueba debe revelarlo.
- Cuando se revela, el fallo debe ser válido.
- Los fallos que se revelan deben ser de interés para el cliente.
- No debe ser redundante.
- Debe motivar al cliente a tomar las medidas necesarias para evitar las consecuencias del fallo encontrado en un caso de prueba.
- Debe ser ejecutable tal y como fue diseñada la prueba.
- Su manutención debe ser sencilla y fácil de realizar.
- El repetirla una cantidad indeterminada de veces debe ser sencillo y no debe representar costo alguno.
- Debe revelar características que están asumidas por defecto.
- Debe ser fácil de evaluar.
- Debe ser posible medir diferentes aspectos de su ejecución.
- Debe proporcionar información útil de depuración al desarrollador.
- Debe ser identificable, es decir, contar con una descripción apropiada, nombre o identificador y con un número que indique su orden en la secuencia de ejecución.

3.4. IMPLEMENTACIÓN DE LOS CASOS DE PRUEBA

Los casos de prueba fueron realizados usando varios lenguajes de programación. Es importante mencionar que en ningún caso fue usado un *framework* comercial de *testing*, todas las herramientas fueron desarrolladas como parte del proyecto.

3.4.1. METODOLOGÍA USADA

Un caso de prueba al ser por sí mismo una pieza fundamental de *software*, debe ser escrito siguiendo las mismas metodologías y buenas prácticas que existen para la creación de código de *software* de producción.

Deben ser necesariamente escritos como Brian Kernighan describe en su ensayo
*"*A regular expression matcher*"* , ("*Un programa para identificar expresiones regulares*") [Kernighan07]:

> *El código es hermoso si es simple -- claro y fácil de entender. El código es hermoso si es compacto -- sólo la cantidad necesaria de código para realizar la labor requerida y no más -- no debe ser críptico al punto de que no pueda ser entendido. El código es hermoso si es lo suficientemente general para resolver una gran cantidad de problemas de una manera uniforme. Uno podría describirlo como elegante, que demuestra buen gusto y refinamiento.*

El lenguaje de programación usado en un caso de prueba puede ser muy variado, e incluso puede llegar a requerir de la combinación de varios. Hoy en día existen una gran cantidad de herramientas multiplataforma y multilenguaje que facilitan la creación de casos de prueba, dado que muchos procesos de identificación se realizan de manera automatizada.

Cada caso de prueba requiere tener una estructura para facilitar su lectura y mantenimiento posterior, de forma muy básica son: configuración previa, ejecución principal y limpieza posterior.

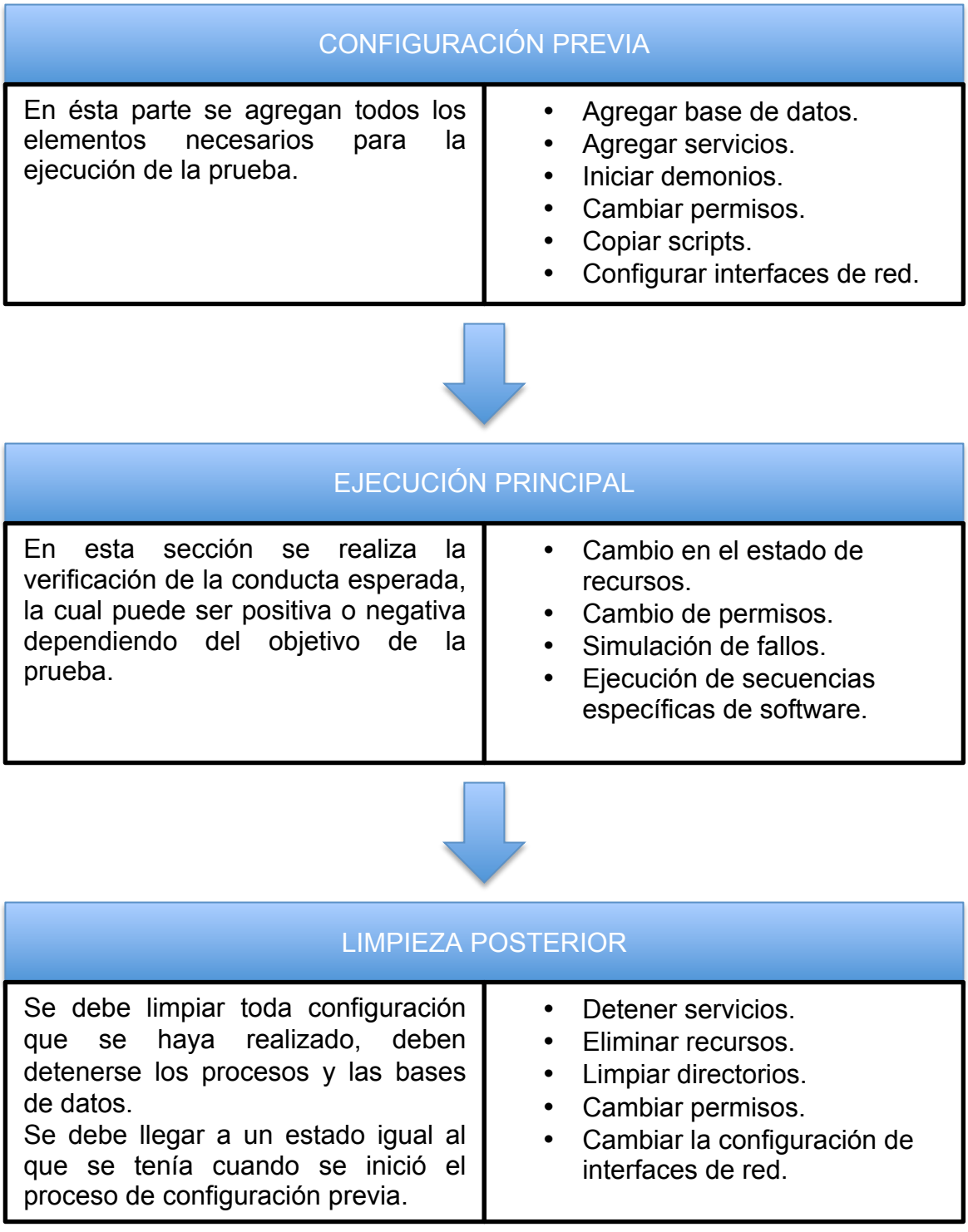

Fig. 5 : Estructura general de un caso de prueba.

Una vez que el caso de prueba fue escrito, es necesario integrarlo a la versión diaria de software. El proceso de integración a la versión diaria del software requiere de varios permisos, el primero es del gerente de desarrollo inmediato, el segundo son los desarrolladores involucrados en la transacción, y el tercero son los gerentes de área. Cada uno proporciona comentarios acerca de los cambios que se están realizando en la transacción. Todos los casos de prueba se ejecutan de manera repetitiva en un periodo determinado por su importancia y complejidad.

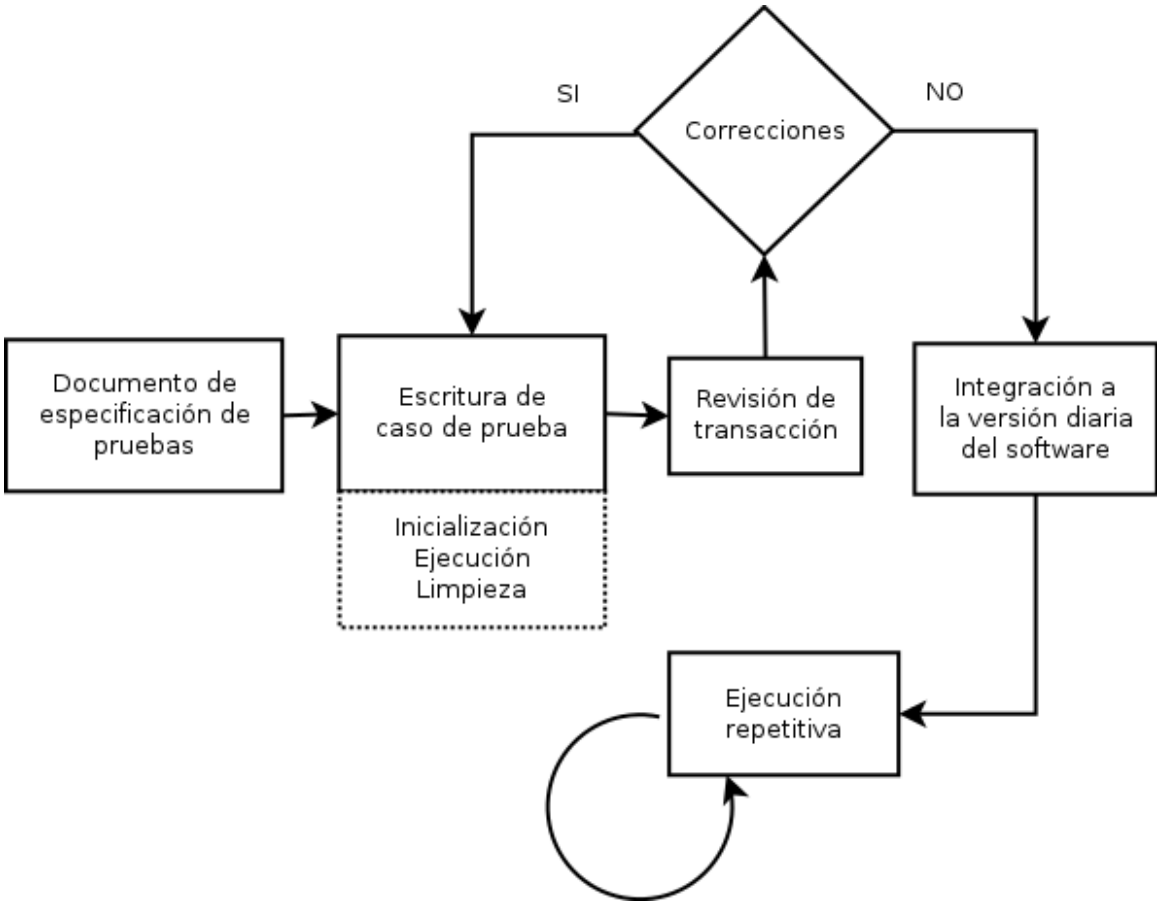

Fig. 6 : Implementación de los casos de prueba.

3.5. MANTENIMIENTO CONSTANTE DEL SISTEMA DE PRUEBAS E IMPLEMENTACIÓN DE MEJORAS

Labores de mantenimiento de código de pruebas:

- Realicé de manera constante, la revisión de las corridas nocturnas de las pruebas de NRE y otros componentes con el objetivo de determinar posibles fallas causadas por cambios en el código de desarrollo, o bien, por errores en el código de pruebas.
- Corregí los errores que se manifestaron durante las corridas nocturnas, y a su vez, reporté los fallos en código de desarrollo.
- Elaboré nuevos esquemas para la optimización y mejora de la infraestructura de automatización de mantenimiento.

Mejoras realizadas al sistema de pruebas:

- Automatización en la creación de ambientes específicos.
- Creación de librerías para la consolidación de clientes de procesamiento de mensajes de NRE.
- Optimización del código de pruebas.
- Creación de nuevas funcionalidades y macros internas que realizan acciones de propósitos específicos, tales como procesamiento de archivos de texto.

3.5.1. METODOLOGÍA UTILIZADA

El mantenimiento de la infraestructura responde a dos necesidades básicas, la primera es agregar funcionalidad, y la segunda es la reparación de un defecto en el diseño o implementación. En los sistemas de pruebas, se requiere que los procesos de configuración y verificación optimicen el uso de recursos para garantizar que el ciclo de desarrollo sea ágil y veraz, ya que los indicadores que éstos arrojan a los miembros del equipo son los que se toman en cuenta para determinar la calidad del *software*.

Todas las pruebas de un producto requieren ser ejecutadas cada día, dado que en un ambiente muy grande, diariamente se realizan más de cincuenta cambios en el código. El desarrollador de pruebas y los gerentes de calidad deben estar atentos de los resultados que las pruebas arrojan día con día para evaluar las estrategias con las que resolverán los problemas que se manifiesten, y tomar decisiones respecto al diseño de la infraestructura de pruebas. Además, las pruebas también pueden priorizarse de manera que sí el conjunto de pruebas más importantes que verifican el funcionamiento central del producto llegan a fallar, el proceso de desarrollo debe entrar en modo de emergencia y se deben resolver los problemas a la mayor brevedad posible.

Una vez que un desarrollador termina de solucionar un defecto o agregar una funcionalidad, debe ejecutar absolutamente todas las pruebas en las que sus cambios puedan tener algún impacto negativo.

La infraestructura de *testing* debe estar diseñada de manera que ejecutar pruebas no tome mucho tiempo y permita al desarrollador probar soluciones alternativas de manera considerablemente rápida.

Se deben extremar precauciones, ya que solucionar cualquier tipo de defecto tiene un riesgo substancial que va de 20 % a 50 % de introducir uno nuevo [Brooks95]. Además, todos los cambios tienden a destruir la estructura original del sistema y agregar cierto grado de entropía.

Para llevar un mejor seguimiento de los cambios realizados y sus posibles consecuencias, es recomendable tener un sistema que ante la manifestación de un fallo, notifique automáticamente al responsable, identifique el tipo de problema y que de ser posible, lo resuelva. Dicha solución suena demasiado perfecta para ser real, sin embargo existe y es usada todos los días por muchas grandes empresas de *software*.

Es de suma importancia contar con documentos de especificación de diseño, dado que entre muchas otras ventajas, proveen al desarrollador y gerentes de un mecanismo para corroborar que los nuevos cambios en el diseño o implementación van acorde con el plan. Hace evidente cuales funcionalidades aún no se realizan y, además, permite tener una visión global de la estructura, lo que facilita la modificación y adición de nuevas características en el diseño.

El documento de especificación de diseño debe establecer de manera clara y concisa la estructura del proyecto o producto. Debe comenzar de manera general y posteriormente irse enfocando lentamente a los factores más específicos. Incluye diagramas de flujo acerca del funcionamiento de los componentes.

La documentación también puede incluirse de manera implícita en el código del producto o prueba, esto se logra nombrando los componentes del programa de manera significativa, además de proporcionar comentarios eficaces. Una buena manera de hacer lo anterior es con las siguientes reglas [Martin08]:

Buenas prácticas:

- Escribir información referente a los derechos de autor.
- Ser informativo, debe explicar de manera eficaz, lo que la situación requiere.
- Revelar la verdadera intención del código.
- Mencionar las consecuencias que representaría un cambio en un segmento del código.
- Comentarios acerca de lo que falta elaborar (*TODO*).

Malas prácticas:

- Escribir comentarios redundantes, repetitivos.
- Escribir comentarios imprecisos.
- Escribir comentarios en código que se explica por sí mismo.
- Escribir comentarios que no tienen ningún significado o que no aportan información.
- Marcadores de posición.

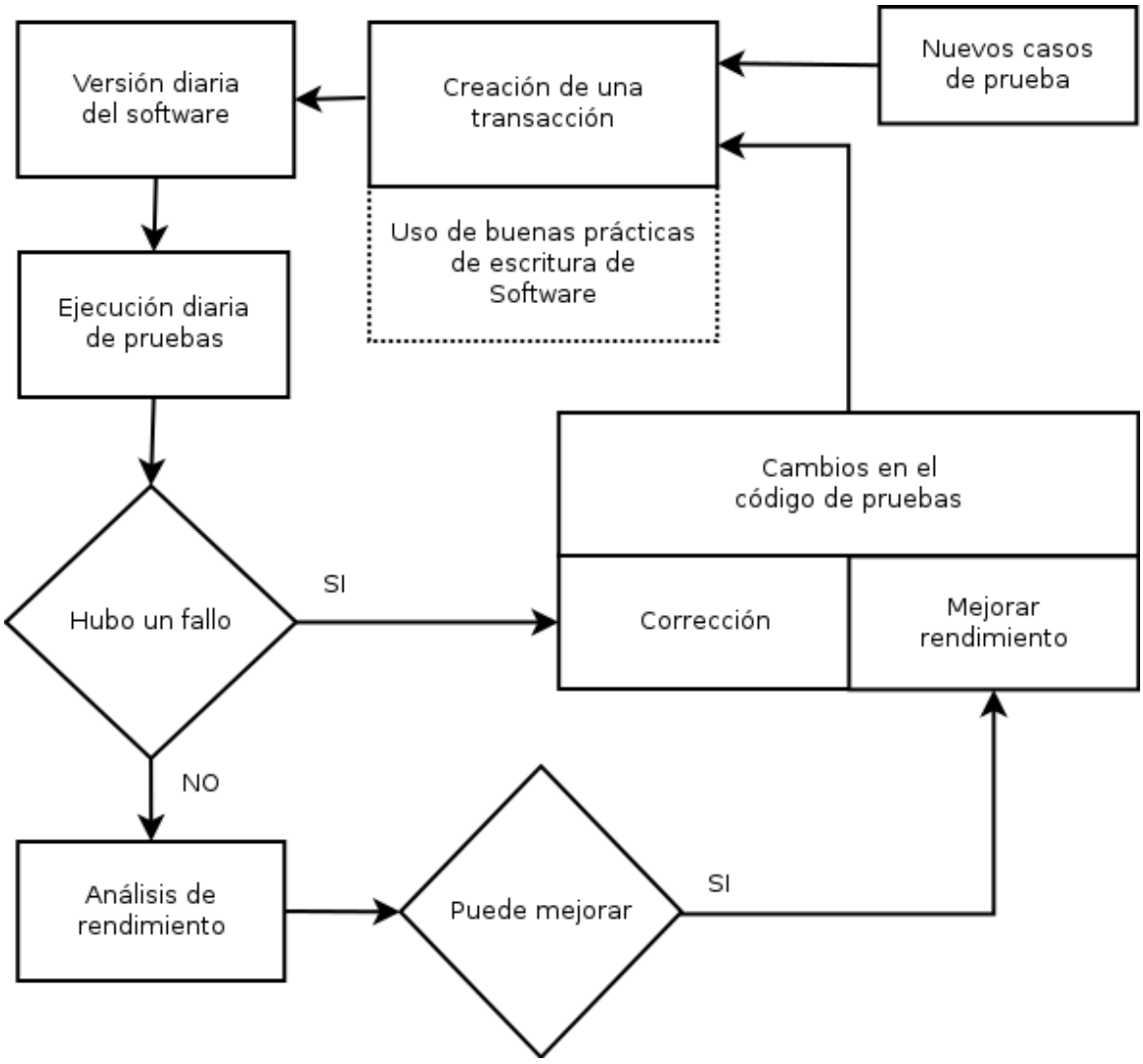

Fig. 7 : Mantenimiento de pruebas.

3.6. ENTREVISTAS A LOS NUEVOS CANDIDATOS

Realicé una gran cantidad de entrevistas técnicas a los aspirantes a ocupar un lugar como desarrollador de *software* en el equipo de *software testing* de la empresa.

3.6.1. METODOLOGÍA USADA

Todo el personal que trabaja como parte del equipo técnico debe estar altamente capacitado en el campo requerido y debe también contar con ciertas características de personalidad. Es por esto que en el proceso de evaluación de nuevos candidatos se pretende evaluar de manera integral todas las habilidades, realizando diversas entrevistas.

Preliminarmente, es preciso conocer las aspiraciones generales que tienen los candidatos en la empresa, es decir, si prefieren trabajar en un ambiente de *backend* o de aplicación, si prefieren diseñar y escribir *software* para el producto o para *testing*, si el periodo que trabajarán será de tiempo completo o de *internship*, y si están dispuestos a cambiar su lugar de residencia, en caso de que no viva en la misma ciudad en la que se encuentra el centro de desarrollo.

Debe además contar con un currículum que preferentemente contenga información relacionada con certificaciones, proyectos escolares, laborales y personales además de buenas calificaciones, estudios posteriores a la universidad, y periodos de *internship* en otras empresas. El currículum debe ser claro y conciso, debe contener información relevante, además de estar escrito en orden cronológico. Un currículum corto y bien estructurado demuestra en el mayor de los casos, mejor preparación y aptitud [Mongan12].

De acuerdo a los datos anteriores, se puede identificar la orientación y dificultad que será empleada para realizar las entrevistas técnicas. Para los casos de desarrollo de *software* de producto y de *testing* se proponen ejercicios lógicos que incrementan su dificultad, en los que se espera que el candidato entienda completamente el problema, proponga una solución ideal, incluso la explique a detalle mientras escribe el código, y finalmente que realice preguntas.

Para el caso particular de *testing*, se realizan preguntas en las que además de resolver un problema lógico complejo, el candidato debe encontrar todos los posibles fallos y consecuencias que el sistema en cuestión puede llegar a tener en un momento determinado. Debe pensar en estrategias para realizar automatización de tareas repetitivas, para lo cual el tener habilidades de *scripting* representa una gran ventaja.

La observación de las conductas y la forma de razonamiento que el candidato manifestó durante la solución de los problemas propuestos es lo que permite

verificar que el nivel técnico mencionado en el currículum sea verídico. Es muy común que el entrevistador sea un miembro del equipo al que se quiere entrar, y que el problema que se propone sea muy similar a los problemas reales que se tienen en la empresa.

Es de suma importancia realizar una evaluación adecuada, dado que cualquier deficiencia que no sea localizada a tiempo puede manifestarse una vez que el candidato haya sido contratado, lo cual involucra el uso de recursos adicionales de la empresa para capacitarlo.

Las entrevistas son realizadas por diferentes miembros del equipo, ya sea *on site* o telefónicas, es importante contar con excelentes habilidades de comunicación en español e inglés, dado que las entrevistas se realizan en ambos idiomas dependiendo del entrevistador.

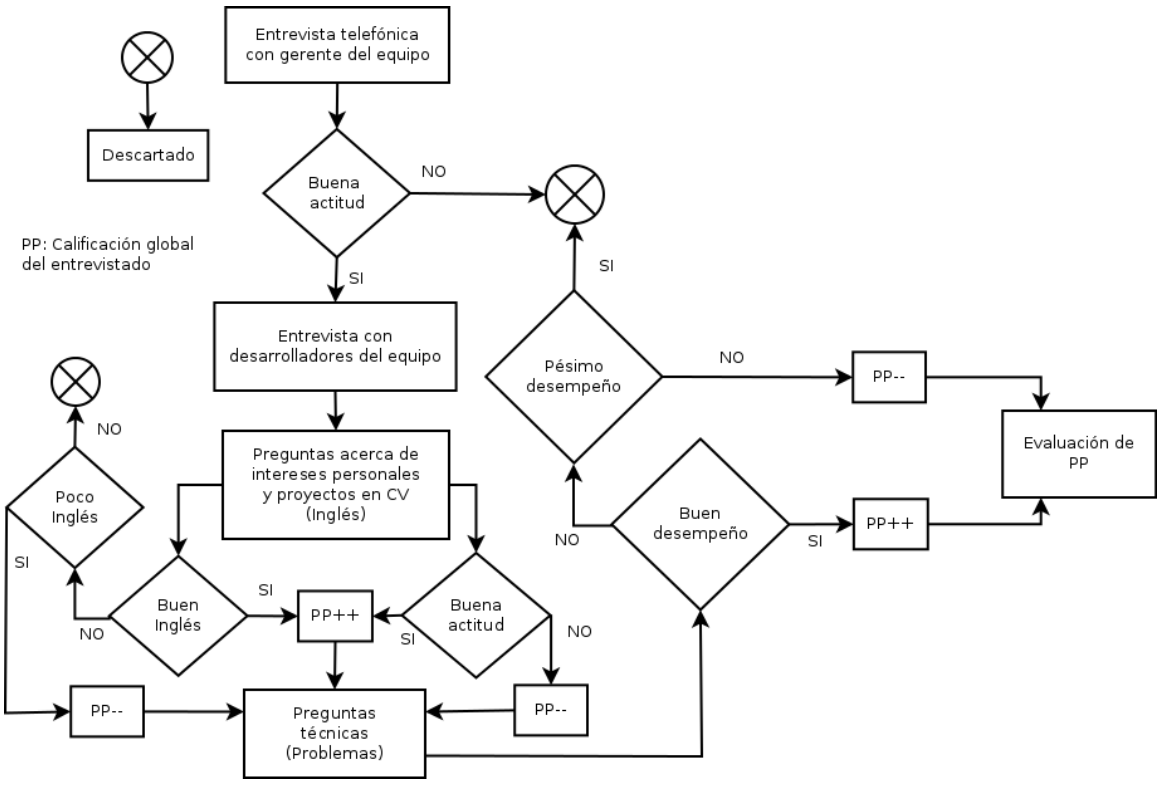

Fig. 8 : Entrevistas a aspirantes.

3.7. EVALUACIÓN DE TIEMPOS DE ENTREGA

Para todas las actividades que realicé he entregado de manera adecuada los estimados de tiempo para la resolución de las mismas.

3.7.1. METODOLOGÍA UTILIZADA

¿Cómo es que un proyecto se puede llegar a retrasar un año? -- Un día a la vez
Frederick P. Brooks. The mythical man month.

Éste es quizá uno de los temas más difíciles a los que el ingeniero de *software* debe enfrentarse, puesto que son muchas las variables que determinan el tiempo que toma terminar un proyecto. Se requiere haber trabajado en varios proyectos antes de poder predecir tiempos de entrega de manera autónoma. Cuando se hacen malas estimaciones se puede llegar a situaciones muy desagradables, en las que el desarrollador tiene una carga de trabajo gigantesca, debido a lo cual sufre de mucho estrés.

Existen metodologías formales y teorías muy elaboradas que están relacionadas con la determinación de tiempos de entrega, sin embargo, difícilmente son usadas en la práctica por el desarrollador, ya que le quitan tiempo que podría ser usado en corregir *bugs* o agregar funcionalidades al código. El siguiente conjunto de guías puede ser tomado a manera de manual para realizar buenas estimaciones:

- Dividir una tarea entre múltiples personas a menudo requiere un esfuerzo extra de comunicación.
- Al igual que con la comida, el resultado de algunas tareas se ve afectado si su proceso de desarrollo se apresura.
- Un equipo de dos personas con un líder es la mejor forma de usar dos elementos.
- Un equipo pequeño es muy lento para sistemas realmente grandes.
- En sistemas grandes, es necesario contar con un equipo que se dedique exclusivamente a revisar los tiempos de entrega.
- Siempre que haya un problema propio o ajeno que obstaculice el desarrollo del producto o prueba, es necesario comentarlo con el equipo y modificar las fechas de entrega en caso de que sea necesario.
- Llevar un registro de las fechas de entrega y lo que debe ser entregado, cada registro debe ser concreto, específico y medible.

4. DISEÑO Y DESARROLLO DE LA INFRAESTRUCTURA DE PRUEBAS AUTOMATIZADAS PARA EL SISTEMA DE NOTIFICACIÓN DE EVENTOS DEL CLUSTER

4.1. ANTECEDENTES

4.1.1. SOFTWARE TESTING

Existen varias definiciones respecto al *software testing*, debido a que no existe una que abarque todos los aspectos importantes, a continuación tres de las más relevantes. La primera de ellas nos dice que: [Myers79] "*es el proceso de ejecutar un programa o sistema con la finalidad de encontrar errores o fallos*". Otra menciona que: [Hetzel88] "*incluye la realización de cualquier actividad que esté enfocada a evaluar un atributo o una capacidad determinada de un programa o sistema y de determinar que cumple con los resultados esperados*", y finalmente otra más: [Jiantao99] "*es un proceso empírico que requiere de investigación profunda para mostrar los riesgos proporcionando información acerca del impacto sobre la calidad*".

El *software* no es muy diferente de cualquier otro sistema físico en el que se reciban entradas y se produzcan salidas, sin embargo, debido a la cantidad tan extensa de variables que intervienen durante el tiempo de ejecución de un programa o sistema, detectar la totalidad de los fallos es, por lo general, extremadamente complicado.

La mayoría de los defectos que verdaderamente requieren ser identificados debido a su alto grado de impacto negativo sobre el *software*, son de diseño y no de manufactura, ya que éstos últimos pueden ser corregidos a tiempo y por lo general es muy sencillo localizarlos. Los entornos de desarrollo modernos incluyen una gran cantidad de herramientas, entre las que destacan los editores de texto con resaltado de errores en tiempo real o el *software* de depuración que permite monitorear el estado de un elemento determinado del sistema. Sin embargo, aún no se cuenta con herramientas que permitan encontrar fallos de diseño genéricos.

Los defectos de *software* siempre van a existir en cualquier módulo de *software* de tamaño moderado, no tanto porque los programadores sean descuidados o irresponsables, sino por la complejidad inherente al sistema. De acuerdo a las teorías acerca las de etapas evolutivas del ser humano, éste tiene una habilidad limitada de comprender la complejidad dependiendo de factores como la edad y la interacción con el medio [Vigotsky79] [Piaget69].

En la mayoría de los textos de ingeniería de *software*, el *software testing* es considerado un “arte” [Jorgens08], ya que el desarrollador de pruebas, o llamado también *tester*, debe tener la capacidad de entender en su totalidad el sistema, sólo así podrá identificar todos aquellos posibles aspectos en los que se puede llegar a manifestar un fallo.

Debe encontrar, también, un mecanismo efectivo que le permita corroborar el correcto funcionamiento del *software*, al mismo tiempo que mantenga un balance ideal entre calidad y cantidad de casos de prueba, para lo cual debe analizar los requerimientos específicos de cada situación y establecer así los parámetros necesarios para que sus resultados sean efectivos.

Es de suma importancia reducir el número y gravedad de los efectos de un *bug*, dado que sus consecuencias pueden ir desde la pérdida de datos o dinero, hasta la pérdida de vidas humanas, como puede ser el caso de un fallo en un sistema de navegación aérea. La calidad es directamente proporcional a qué tan al pie de la letra se sigan los requerimientos de diseño. Para que sea aceptable, el mínimo grado de calidad es que la aplicación o sistema trabaje de manera requerida en las circunstancias especificadas.

Podemos visualizar el *software testing* como un gran mecanismo de depuración que es ejecutado intensamente para encontrar errores de diseño e implementación introducidos por el programador. La imperfección humana hace que sea prácticamente imposible que un programa de complejidad moderada sea realizado correctamente por primera vez [Jiantao99]. El propósito de la depuración en la fase de programación (véase fig. 6) es encontrar los problemas y brindar una pronta solución.

Resulta bastante útil para los desarrolladores de pruebas, contar con herramientas que le permitan determinar con resultados numéricos y veraces si un programa o sistema funciona correctamente bajo una o más circunstancias, además, de poder hacer comparaciones entre productos. Usando dichos datos, el desarrollador de pruebas puede decidir si el sistema funciona bien o no.

Dado que no podemos obtener un factor de calidad de manera directa, debemos usar factores que nos permitan tener una buena estimación. Dichos factores son: la funcionalidad, la ingeniería, y la adaptabilidad. La funcionalidad comprende los valores de exactitud, usabilidad e integridad; la ingeniería comprende la eficiencia, capacidad de prueba, documentación y estructura, y finalmente, la adaptabilidad, enmarca los factores de flexibilidad, reusabilidad y mantenibilidad [Hetzel88].

Además de ser una herramienta de depuración, el *software testing* también puede ser tomado como una herramienta de muestreo estadístico donde sean registrados los fallos y éxitos numéricamente, y así poder hacer una estimación de futuros resultados.

4.1.2. TERMINOLOGÍA BÁSICA USADA EN SOFTWARE TESTING

Paul Jorgensen menciona en su libro *Software Testing* [Jorgensen08] las siguientes definiciones:

Error: Los seres humanos cometen errores. Un buen sinónimo es "equivocación". Cuando la gente comete errores mientras programa, se suelen llamar *bugs*. Los errores tienden a propagarse. Un error en los requerimientos puede crecer durante la etapa de diseño y ser amplificado aún más durante la escritura del código.

Falta: Es el resultado de un error. Es su representación, o modo de expresión. Puede usarse defecto como un sinónimo.

Fallo: Un fallo sucede cuando una falta se ejecuta.

Incidente: Es el síntoma visible al usuario, se genera cuando se ejecuta una falta, o bien, cuando ocurre en fallo.

Probar el software: Es el acto de verificar el funcionamiento del software mediante casos de prueba. Probar el *software* tiene dos objetivos: encontrar fallos y demostrar el funcionamiento correcto del *software*.

Caso de prueba: Un caso de prueba tiene una identidad única y está asociado con una característica o funcionalidad específica de un programa cuando trabaja bajo ciertas condiciones. Mediante su ejecución se pretende determinar si el comportamiento resultante es correcto o erróneo. Tiene sus propias entradas y salidas. Las entradas deben ser de dos tipos: los parámetros que se van a pasar al sistema, y las condiciones ambientales que se deben cumplir, llamadas también pre-condiciones. De manera análoga con las salidas, deben existir dos tipos, las post-condiciones y los resultados que el programa de pruebas debe arrojar.

Calidad: Es el valor para una persona o empresa.

4.1.3. NIVELES DE TESTING

En el siguiente modelo se representan los diferentes niveles de *testing* y los objetivos de cada uno. El diagrama enfatiza la correspondencia entre *testing* y los niveles de diseño. En términos de *testing* estructural las tres fases de definición corresponden directamente a las tres etapas de *testing*. Es más conveniente realizar el *testing* estructural en la fase de prueba unitaria, mientras que *testing* funcional es más recomendable en *testing* a nivel sistema debido a la cantidad de información a ser verificada.

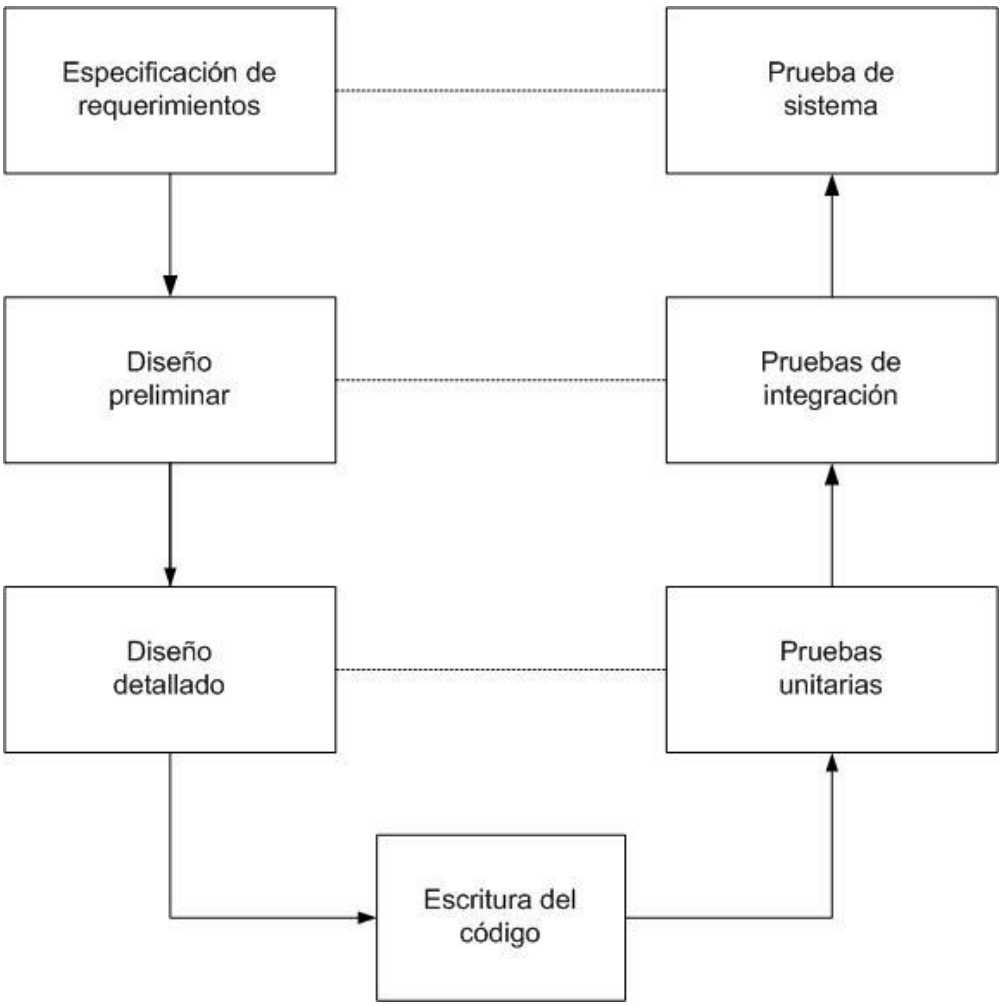

Fig. 9 : Niveles de *testing*.

4.1.4. SAD - SISTEMA DE ALTA DISPONIBILIDAD

La solución de cómputo distribuido de la empresa, SAD, satisface las necesidades de alta disponibilidad, desempeño, escalabilidad y de recuperación de fallos en un ambiente de bases de datos.

Su característica principal consiste en que se elimina a la base de datos como un punto único de fallo. Permite además la ejecución de múltiples instancias en una misma base de datos. Dichas instancias corren en diferentes nodos del *cluster*, y acceden el conjunto de información que constituye la base de datos.

La ventaja más importante respecto a un esquema de una única instancia consiste en que, a diferencia de ésta, presenta múltiples recursos de procesamiento para los usuarios de la base de datos.

Una instancia es un conjunto de estructuras de memoria en una máquina, asociadas con la base de datos. Por su lado, la base de datos es una gran colección de archivos físicos. Entre la base de datos y las instancias existe una relación de uno a muchos. Una base de datos puede estar montada concurrentemente por más de una instancia.

Dentro de un ambiente de alta disponibilidad, la base de datos puede ser visualizada como un recurso que puede ser fácilmente compartida en un *pool* de recursos entre los diferentes servidores que constituyen el *cluster*. Usando mecanismos de QoS, es posible mover recursos automáticamente de un *pool* a otro para garantizar que los objetivos de desempeño se mantengan.

El sistema de almacenamiento distribuido de alto rendimiento de la empresa hace uso de una infraestructura de archivos compartidos como base para su funcionamiento. Una pieza fundamental para su funcionamiento es el uso de una arquitectura *shared everything*, la cual elimina las limitaciones de *shared nothing* y *shared disk* garantizando altos niveles de calidad y confianza sin que se requieran cambios en las aplicaciones clientes.

Provee los siguientes beneficios fundamentales, esenciales para garantizar alta disponibilidad:

- Confiabilidad

  Al eliminar a la base de datos como el único punto de fallo, si una instancia determinada falla, las demás instancias en el pool de recursos se mantienen abiertas y activas. Su sistema de *clusterware* monitorea todos los procesos de la base de datos e inmediatamente reinicia cualquier componente que haya fallado.

- Detección de errores

  El sistema de *clusterware* automáticamente monitorea las bases de datos del ambiente de alta disponibilidad así como otros procesos, que pueden ser instancias, *listeners* o interfaces de red virtuales, y provee un mecanismo de detección rápida de errores en el ambiente

- Capacidad de recuperación

  Si una instancia falla en una base de datos dentro del ambiente de alta disponibilidad, es identificada por alguna otra instancia en el *pool* y un procedimiento de recuperación será ejecutado inmediatamente. NRE y RRC (*Rápida Recuperación de la Conexión*) enmascaran más fácilmente el fallo del componente al usuario.

4.1.5. CLUSTERWARE

El sistema de *clusterware* es la parte de infraestructura de *software* que permite proveer alta disponibilidad, pues proporciona todos los mecanismos para que SAD funcione en modo distribuido a nivel de sistema operativo. Se trata de un *software* portable que está constituido por varios procesos que corren en segundo plano, y llevan a cabo diferentes funciones que facilitan las operaciones del *cluster*. Puede correr sólo o con otros paquetes de soporte de *clustering* como *Sun Cluster* o *TruCluster*.

Los procesos de segundo plano y servicios de los que el *clusterware* de SAD se compone son: sacd, cssd, procd, evd y sn. El sistema operativo a través de su *demonio* de arranque *init*, ejecuta estos procesos usando los scripts de arranque del *cluster,* los cuales son instalados por el instalador universal durante la instalación del sistema SAD.

El sistema de *clusterware* permite a cada nodo comunicarse con los demás, unifica los nodos de tal manera que el *cluster* puede ser visualizado como un solo *host.* Es gestionado por SAC usando el registro del *cluster,* el cual almacena y mantiene la información de pertenencia de los nodos y sus recursos en el *cluster*.

4.1.6. SRN - SISTEMA DE SERVICIOS DE NOTIFICACIÓN

El sistema de servicios de notificación de eventos es configurado durante la instalación del sistema de SAD y se ejecuta en cada nodo del *cluster* cuando el demonio SAC inicia. Para cada cambio de estado de un recurso del *cluster*, SAC genera información de alta disponibilidad como *timestamps*, nombre del recurso, razón por la cual cambió de estado, el estado inicial y final, entre otros. Esta información de alta disponibilidad se conoce como evento.

Los eventos son creados por SAC y son transmitidos a los servicios de notificación, y son publicados a una capa superior en donde hay clientes que los consumen, por lo general con el objetivo de detectar fallos de manera muy rápida. Para poder consumir los eventos que publican los servicios de notificación, éste debe estar instalado en cada nodo. A todo el proceso de generar y publicar eventos se le conoce como NRE.

Los eventos NRE por sí solos no tienen utilidad, a menos de que la aplicación que los consume tenga la lógica requerida para procesarlos y llevar a cabo una acción en respuesta. La mejor manera de recibir eventos NRE es mediante el uso de clientes que estén integrados, como los recursos de la base de datos.

4.1.7. NRE - SISTEMA DE NOTIFICACIÓN RÁPIDA DE EVENTOS

NRE es un mecanismo de notificaciones de alta disponibilidad (*HA*), que se usan para comunicar a otros procesos acerca de cambios ocurridos en la configuración, incluyendo cambios en el estado de los servicios. NRE es capaz de terminar inmediatamente transacciones que se estén ejecutando en el momento en que una instancia o servidor falle. Los clientes integrados con NRE pueden recibir las notificaciones y actuar al respecto llevando a cabo acciones como propagar el error, o bien silenciarlo y volviendo a ejecutar la transacción.

Básicamente existen dos tipos de eventos: los *DOWN*, que ocurren cuando algún recurso del *cluster* fue desactivado o apagado. De manera análoga, un evento de tipo *UP* representa el momento en que un recurso fue iniciado. Cuando un evento *DOWN* ocurre, los clientes integrados pueden limpiar las conexiones con la base de datos inmediatamente. El segundo tipo son los *UP*, cuando éstos ocurren, el cliente crea nuevas conexiones a la nueva instancia de la base de datos.

Los eventos pueden ser publicados usando SN y las colas de almacenamiento avanzado. Las colas son configuradas automáticamente cuando se configura un servicio. SN por su parte, debe ser configurado manualmente usando una interfaz.

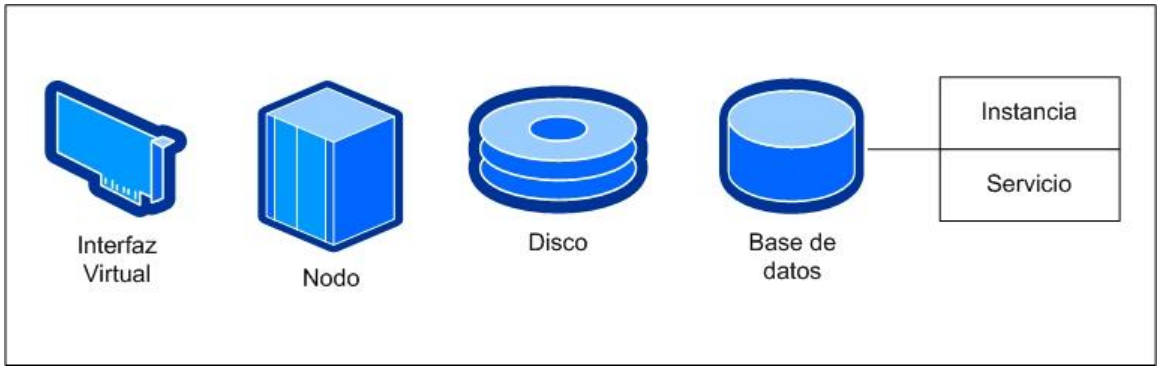

Fig. 10 : Algunos de los recursos disponibles en el *cluster* de SAD.

Los eventos son generados a partir de un cambio en el estado de alguno de los recursos registrados en el *cluster*, los cuales pueden ser bases de datos, instancias, servicios, entre muchos otros. Los cambios de estado pueden ser generados porque el recurso se inició o bien, se detuvo.

El evento es notificado por el proceso SACD y por el agente de la base de datos a otro agente llamado *sadagent*, cuya función, además de muchas otras, es invocar al demonio EVD quien a su vez distribuye los mensajes a los tres agentes de distribución de eventos, conocidos comúnmente como *publishers*, los cuales son: SN, *User Callouts* y CA (*Colas avanzadas*). De acuerdo a sus necesidades, los clientes deben elegir un método de conexión para obtener los eventos generados. Para el caso de SN, los métodos de conexión y obtención de eventos son muy variados dependiendo de la plataforma, se pueden obtener mensajes para clientes escritos en Java usando RUC, en C usando la tecnología apropiada, entre otros. Por su parte, *User Callouts*, es un mecanismo en el que habiendo recibido un evento, el agente ejecuta todos los programas en un directorio determinado, dichos programas son creación del usuario.

Por lo general, se trata de programas que envían correos electrónicos o bien, que ejecutan alguna tarea de administración del servidor. Finalmente, CA es una plataforma en la que los mensajes son obtenidos por clientes escritos en PL/SQL. A todos los programas cliente que se conectan a ONS, *User callouts*, o CA, se les conoce como *subscribers* o clientes subscriptores. Cabe mencionar que los mensajes emitidos por SN, *User Callouts* y CA tienen diferencias entre sí respecto al formato con el que son publicados. El conjunto de atributos de los eventos se conoce como "*payload*".

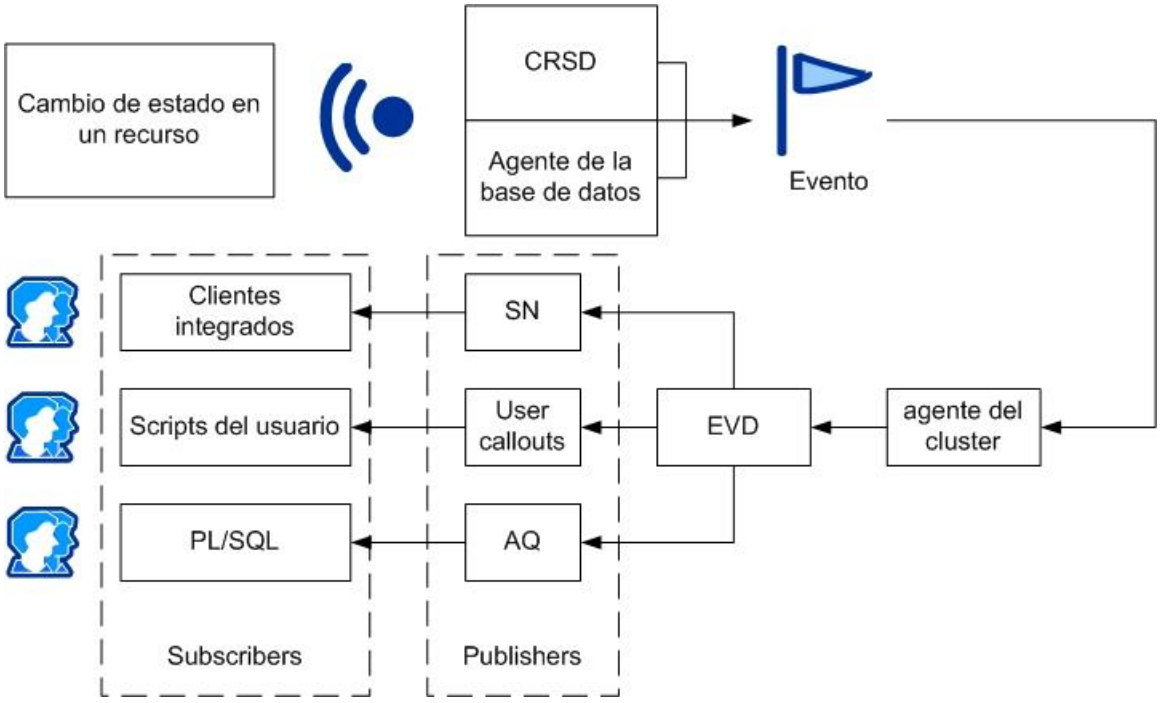

Fig. 11 : Diagrama general del funcionamiento de NRE.

4.2. LA INFRAESTRUCTURA DE PRUEBAS

4.2.1. CLIENTES SUBSCRIPTORES

Un cliente es básicamente un programa que se usa para realizar una conexión con el sistema de servicios de notificación a forma de subscripción, mediante la cual va a ser capaz de obtener notificaciones de alta disponibilidad cada vez que ocurra un cambio en el estado de algún recurso del *cluster*. El sistema de servicios de notificación tiene soporte para varias tecnologías y lenguajes de programación, tales como C, C++, Java y PL/SQL por mencionar algunos. Dicha diversidad radica en que se debe proporcionar al usuario un conjunto de herramientas integral mediante el cual se satisfagan sus necesidades particulares.

Un cliente del sistema de servicios de notificación permite que el usuario responda de manera eficaz ante eventos esperados e inesperados del *cluster* además de permitir un mejor control de recursos. La infraestructura de pruebas desarrollada brinda soporte a todos los tipos de cliente que pueden recibir eventos de alta disponibilidad.

En general, los clientes suscritos al sistema de servicios de notificación funcionan bajo un esquema de programación orientada a eventos, cada evento recibido debe ser procesado individualmente para poder brindar una respuesta adecuada. Su estructura se podría dividir en tres secciones básicas: la realización de la conexión, un bucle de procesamiento de eventos y la finalización de la conexión.

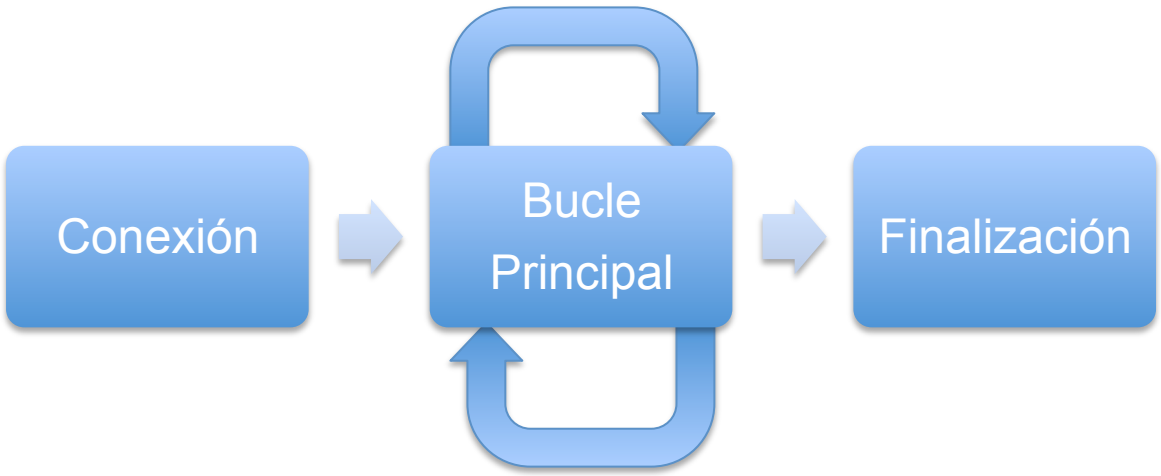

Fig. 12 : Diagrama general del funcionamiento de un cliente subscriptor.

El mecanismo de conexión al sistema de servicios de notificación de cada cliente, es muy específico, ya que difieren las tecnologías y librerías usadas para cada uno. Por lo general se proporciona una cadena en la que se especifica información acerca de los detalles de la conexión, tal como los puertos, nombre de la base de datos y servicios, nombre del servidor en el que se encuentra corriendo el "*listener*", entre otros. Dicha cadena puede ser pasada directamente al cliente a manera de parámetro de línea de comandos o bien, puede estar contenida en un archivo de configuración.

Durante la fase de procesamiento de eventos, el cliente debe analizar cada atributo del evento para poder determinar la respuesta a ejecutar.

La parte final es donde el cliente se detiene para dejar de procesar eventos. La condición para entrar a esta sección es que encuentre un archivo de sincronización o se reciba directamente un evento de finalización.

4.2.1.1. TIPOS DE CLIENTES SUBSCRIPTORES

Existen diferentes tipos de clientes con el objetivo de satisfacer todas las necesidades del usuario. Cada tipo de cliente subscriptor tiene su respectivo programa servidor corriendo como parte del sistema de servicios de notificación. Así mismo, cada uno posee una forma particular de representar y procesar el evento:

SN

- Recibe eventos directamente del sistema de servicios de notificación.
- Soporta los lenguajes de programación: C/C++ y Java
- Es el mecanismo de conexión más usado.

Payload de un evento de tipo SN:

```
2013-09-06 09:35:46.892: [ USRTHRD][17] SN
SnEventForwarder::postSnEvent posting event
"VERSION=1.0 event_type=NODE host=uimu1_mlb01iel_1
incarn=0 status=nodedown reason=public_nw_down
vip_ips=10.228.213.34 timestamp=2013-09-06 09:35:46
timezone=-07:00"
```

USRCO

- Recibe eventos producidos por el "*publisher*" de usrco.
- Todos los scripts o binarios que se encuentren alojados en la carpeta de usrco, son ejecutados cada vez que ocurre un evento.
- El usuario puede agregar cuantos scripts y binarios necesite.

Payload de un evento tipo User Callouts:

```
2013-09-06 09:35:46.584: [ USRTHRD][10] Usrco
UsrcoEventForwarder::postMyEvent posting event "NODE
VERSION=1.0 host=uimu1_mlb01iel_1 status=nodedown
reason=member_leave incarn=273920444 timestamp=2013-
09-06 09:35:46 timezone=-07:00 "
```

OCISN

- Recibe eventos desde el sistema de servicios de notificación.
- Es un mecanismo muy similar a sn, sin embargo, mediante el uso de un API particular (OCI), permite tener una mejor interacción con la base de datos.
- Soporta los lenguajes de programación: C/C++

Payload de un evento de tipo SN:

```
2013-09-06 09:35:46.892: [ USRTHRD][17] SN
SnEventForwarder::postSnEvent posting event
"VERSION=1.0 event_type=NODE host=uimu1_mlb01iel_1
incarn=0 status=nodedown reason=public_nw_down
vip_ips=10.228.213.34 timestamp=2013-09-06 09:35:46
timezone=-07:00"
```

CA

- Obtiene los eventos desde una base de datos.
- La particularidad de este mecanismo es que almacena los eventos mediante "encolamiento avanzado", el cual permite que los mensajes puedan ser almacenados persistentemente en la base de datos, y propagados entre las diferentes colas en múltiples servidores y bases de datos.
- La transmisión de la información se realiza mediante HTTP(S), SMTP y otros protocolos propietarios.
- Es un mecanismo que soporta: PL/SQL, Java y C.

Payload de un evento de tipo CA:

```
2013-09-06 09:35:46.586: [ USRTHRD][19] DBAGENT-uimu1
CaEventForwarder::postHaAlert posting event
":reason_name->NODE_DOWN:event_id->328:event_time-
>2013-09-06 09:35:46 -07:00:timestamp_fmt->YYYY-MM-DD
HH24:MI:SS TZH:TZM:event_reason-
>member_leave:host_name->aime1_mlb01iel_1:incarnation-
>273920444:cardinality->'':alert_timeout_seconds-
>300:immed_timeout->N:db_unique_name->:instance_name-
>:service_name->"
```

4.2.2. SCRIPTS

Software testing es un área en la que se hace un uso extensivo de la automatización por lo que es necesario contar con herramientas que permitan realizar tareas repetitivas de manera sencilla y eficaz. La manera más conveniente de lograrlo es mediante scripts que además también pueden resolver el problema de la portabilidad si es que se usa un esquema multiplataforma.

Los scripts realizados como parte de la infraestructura de pruebas pueden ser clasificados en dos divisiones principales de acuerdo a su impacto de uso.

4.2.2.1. DE SOPORTE GENERAL A LA EMPRESA

Se trata de scripts que proveen funciones básicas y de propósito general, los cuales reducen la complejidad de una gran cantidad de tareas. Éstos scripts pueden ser usados por cualquier equipo de desarrollo y pruebas dentro de la empresa ya que no son específicos para una tecnología en particular.

Ejemplos de programas desarrollados:

- Obtener líneas
- Ordenar archivos
- Obtener archivos
- Renombrar archivos

4.2.2.2. DE SOPORTE ESPECÍFICO AL PROYECTO

Están dedicados exclusivamente a un proyecto en particular, no se espera que puedan ser usados para cualquier equipo de desarrollo ya que resuelven problemas muy puntuales y por lo general están enfocados en incrementar el performance de alguna tarea.

Ejemplos de programas desarrollados:

- Encontrar eventos dado un conjunto de atributos en específico.
- Verificación y comparación de archivos.
- Obtener el nodo maestro en un momento determinado.

4.2.3. MACROS

Dentro del contexto de la empresa, una macro es desarrollada para ser usada como una función integrada dentro del lenguaje interno de pruebas con el objetivo de reducir el número de líneas de código de cada prueba. Mientras que el número de líneas de código disminuye, la legibilidad aumenta considerablemente, haciendo que se optimice el ciclo de desarrollo al crear código modular y reusable.

Para crear una macro es necesario determinar un nombre para el nuevo comando, y un archivo donde estará almacenado el código que la macro va a ejecutar cada vez que sea invocada.

Las macros son usadas ampliamente por los desarrolladores de pruebas y del producto ya que son bastante útiles.

Al igual que con los scripts mencionados en la sección anterior, se pueden subdividir en dos clasificaciones.

4.2.3.1. DE SOPORTE GENERAL A LA EMPRESA

Ejemplos de macros desarrolladas:

- Obtener archivos con una sintaxis particular.
- Buscar un *string* en un archivo.
- Hacer subdivisión de una cadena de acuerdo a un *token*.

4.2.3.2. DE SOPORTE ESPECÍFICO AL PROYECTO

Ejemplos de macros desarrolladas:

- Macros auxiliares a la rotación de *logs*.
- Macros auxiliares a la segmentación de *logs*.
- Macros de verificación de eventos de alta disponibilidad.

4.2.4. LIBRERÍA DE PROCESAMIENTO DE EVENTOS

Los clientes que están suscritos al sistema de servicios de notificación pueden ser de varios tipos, sin embargo, las tareas que realizan internamente son muy similares, de tal manera que para todos los que funcionan con un lenguaje de programación común, se puede elaborar una librería mediante la cual se evite la ambigüedad de código, y al igual que con las macros, aumente la legibilidad y se reduzca la complejidad de mantenimiento.

A grandes rasgos, la librería de procesamiento de eventos funciona de la siguiente manera:

- El contenido particular de un evento recibido "*payload*" es "pre" procesado y convertido a un evento genérico en el que están almacenados todos sus atributos.

- Todos los eventos genéricos son guardados en una estructura de datos particular, creada específicamente para ese fin.

- Genera un archivo de texto para cada evento recibido de manera que pueda ser usado posteriormente por las macros de verificación de eventos.

Un ejemplo de cómo un cliente subscriptor obtiene y procesa los eventos puede ser representado de la siguiente forma:

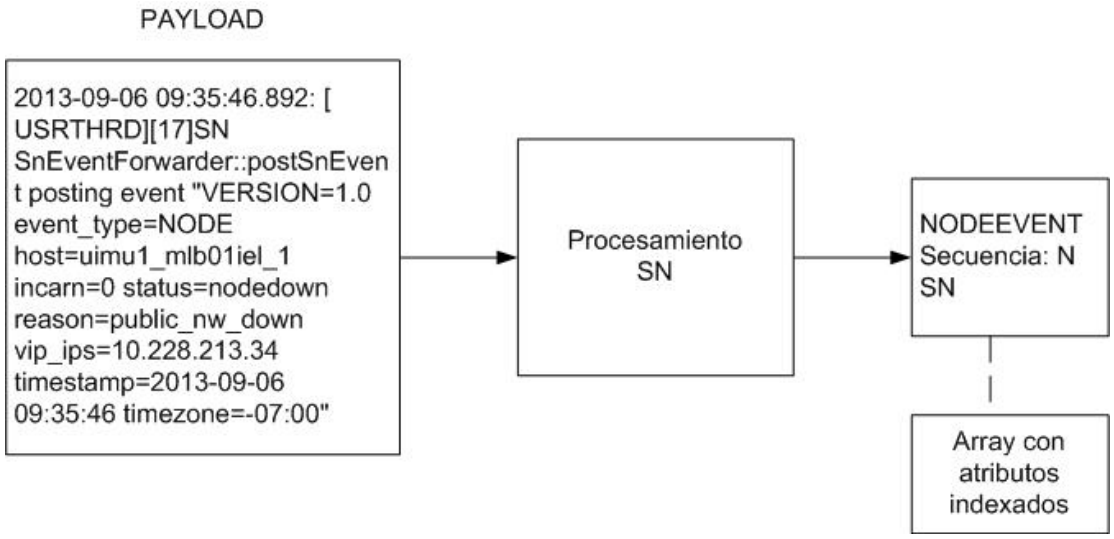

Fig. 13 : Procesamiento de *payload* para SN.

La librería de procesamiento de eventos hace posible que todos los subscriptores hagan uso del mismo mecanismo.

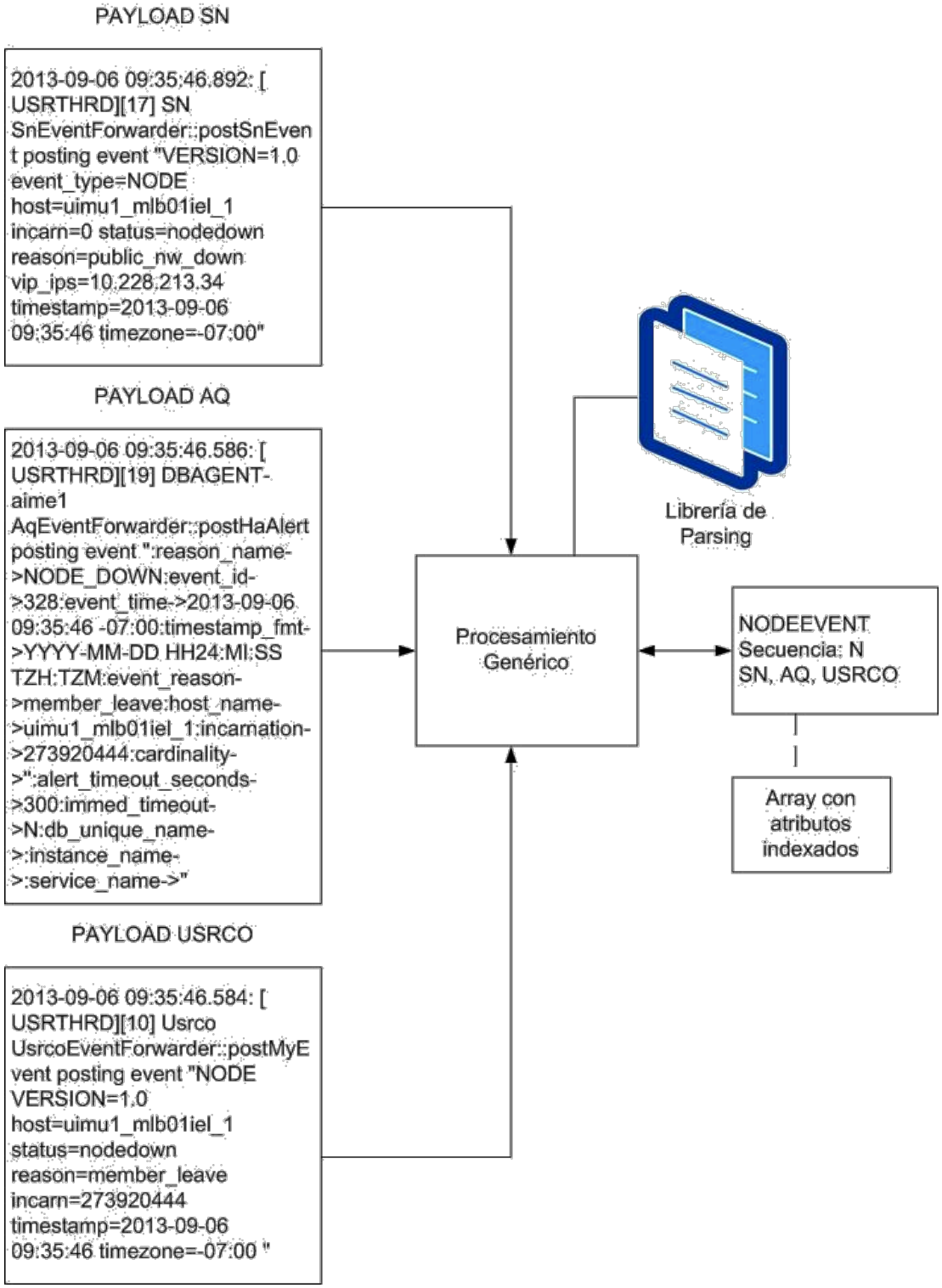

Fig. 14 : Procesamiento de *payload* usando la librería de *parsing.*

4.2.5. CASOS DE PRUEBA

Los casos de prueba son scripts creados en el lenguaje propietario de pruebas de la empresa, mediante los cuales se integran los clientes, macros, otros scripts, comandos externos, y todo aquello que se deba tomar en cuenta para la reproducción de una conducta en particular que vaya a ser ejercitada por la prueba. Las pruebas deben estar diseñadas de tal manera que al final de la ejecución de la misma, el desarrollador pueda saber con certeza si los resultados obtenidos son correctos o incorrectos. Los resultados deben arrojar información que permita el diagnóstico conciso en caso de que haya ocurrido algún fallo.

Estructura general de una prueba:

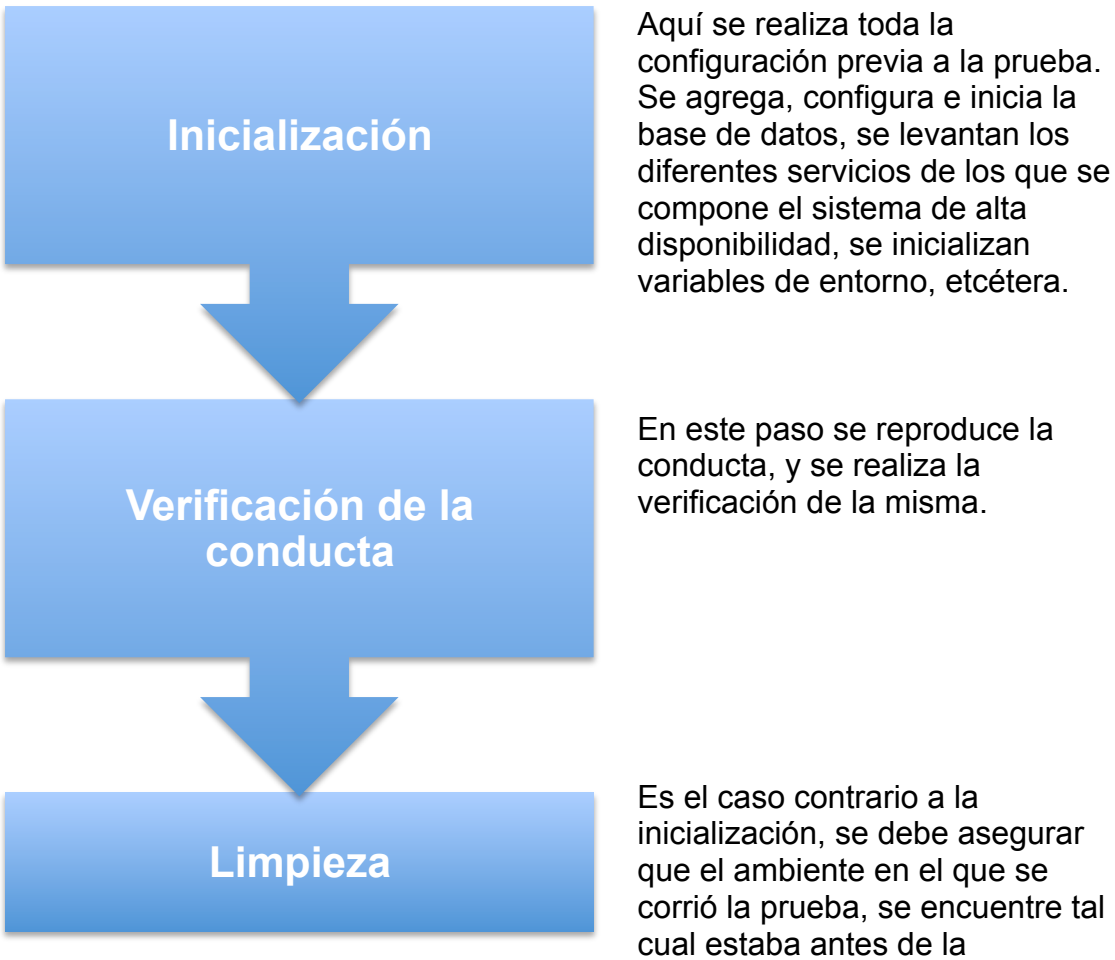

Fig. 15 : Estructura de un caso de prueba.

Todos los casos de prueba deben ser escritos con especial cuidado ya que mediante ellos es como se refleja la salud de los diferentes productos de la empresa. Las prácticas de la empresa hacen hincapié en que las pruebas se escriban teniendo en consideración las siguientes características:

Modularidad

Las pruebas deben ser escritas de manera que no tengan dependencia entre sí, y puedan ser usadas como conjunto para integrar diversos ambientes de verificación de conductas.

Reusabilidad

Cada prueba debe brindar la posibilidad de ser usada por más de un ambiente de verificación, de ésta forma se evita la ambigüedad y se reduce el costo de mantenibilidad.

Mantenibilidad

Las pruebas deben estar escritas de tal manera que sean legibles tanto para el desarrollador que las creó, como para todos los miembros del equipo. Escribir código modular y reusable es la mejor manera de impulsar ésta característica.

Portabilidad

Es importante mencionar que todos los escenarios de pruebas deben ser ejecutados en todas las plataformas que el sistema de alta disponibilidad soporta: Windows, Linux, HP-UX, zOS y Solaris, por lo tanto, la configuración de las pruebas a realizar debe ser extremadamente cuidadosa para que no ocurran problemas de portabilidad.

Buena documentación

Existen una gran cantidad de estándares de escritura de comentarios, todos ellos aplican para el caso de las pruebas, por mencionar algunos, se deben de elegir nombres de variables con un significado eficaz, se debe de evitar escribir un comentario en la misma línea que un archivo además de que cada que se agrega un nuevo archivo o una función, es necesario agregar su descripción. Ver sección 3.4.1.

4.2.5.1. TIPOS DE CASOS DE PRUEBA

La infraestructura de pruebas del sistema de notificación rápida de eventos contempla tres diferentes niveles a los que se realizan verificaciones de una conducta en particular:

Pruebas unitarias

Aquí se verifica el comportamiento de un segmento de código en particular, que puede ser una función, un ciclo, la entrada a una condicional o el cambio en una variable. Son pruebas que tratan de verificar una conducta muy específica del producto.

Pruebas de integración

Se trata de pruebas de mediano nivel, verifican el comportamiento de muchos componentes individuales ejecutándose al mismo tiempo e interactuando de manera directa o indirecta.

Pruebas de sistema

Los componentes cuya conducta fue verificada en las pruebas de integración, son ejercitados dentro de un ambiente en particular que puede ser un esquema de múltiples bases de datos o eventos.

En la empresa, cada día se ejecutan pruebas de sistema, integración y unitarias, para las últimas se tiene especial cuidado ya que representan las funcionalidades más importantes de los diferentes productos. En caso de que las pruebas unitarias fallen, la generación diaria del producto se detiene y los desarrolladores causantes de que la prueba haya fallado son notificados y obligados a corregir el error.

4.2.6. AMBIENTES DE PRUEBA

Al ser un componente vital del sistema de alta disponibilidad, el sistema de distribución de eventos requiere ser ejecutado en todos los escenarios posibles para ejercitar todas las conductas del software y así, encontrar la mayor cantidad de fallos.

4.2.6.1. MÚLTIPLES BASES DE DATOS

Se trata de un ambiente en el que hay múltiples bases de datos corriendo al mismo tiempo dentro de un sistema de alta disponibilidad. El reto de crear este ambiente es en primer lugar, hacer que las bases de datos puedan coexistir sin ninguna interferencia entre ellas, y posteriormente, diferenciar los eventos que vienen de una base de datos y otra. Los clientes subscriptores y los scripts de verificación de eventos que forman parte de la infraestructura de pruebas, son suficientemente robustos para satisfacer estas necesidades.

Este ambiente es muy necesario ya que conforme el tiempo transcurre, los requerimientos emergentes de almacenamiento distribuido de información van aumentando progresivamente, para lo cual se debe garantizar que el sistema de servicios de notificación funcione de manera apropiada ante la creación de más de una base de datos.

El esquema de múltiples bases de datos puede verse como sigue:

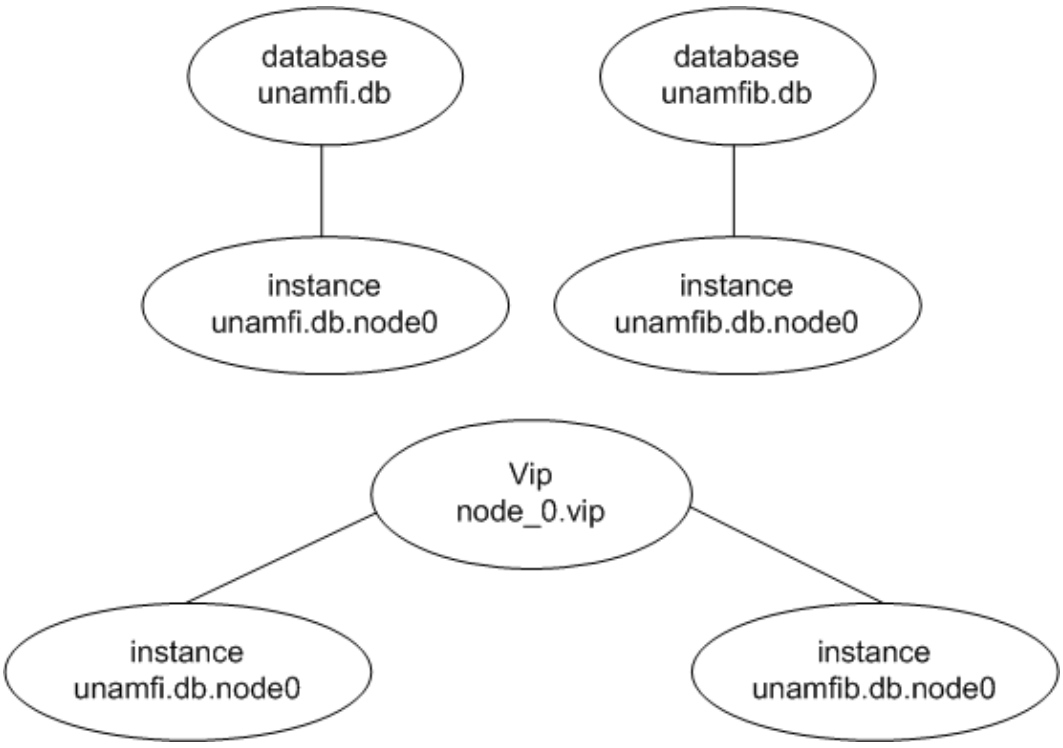

Fig. 16 : Esquema de múltiples bases de datos.

4.2.6.2. MÚLTIPLES SERVICIOS

Una base de datos precisa contar con diferentes servicios para permitir la conexión a ella, por lo tanto, es importante que cada que se presente un cambio de estado en un servicio determinado, los eventos correspondientes se produzcan adecuadamente y sean entregados al usuario para que éste pueda volver a conectarse en caso de que haya sido desconectada alguna de sus aplicaciones debido a un fallo.

Los eventos que puede llegar a producir el cambio en el estado de un servicio son diversos, tales como los producidos cuando el servicio se detiene o se inicia, o bien cuando se mueve entre un nodo y otro de acuerdo a las políticas de reubicación del sistema de alta disponibilidad.

Para este ambiente en particular, fue necesario realizar una macro que permitiera saber el momento preciso en que un recurso determinado cambiara a un estado que no fuera simplemente OFFLINE u ONLINE ya que para los casos de reubicación de servicios, los estados pueden ser de otros tipos.

El esquema de múltiples servicios se puede visualizar de la siguiente forma:

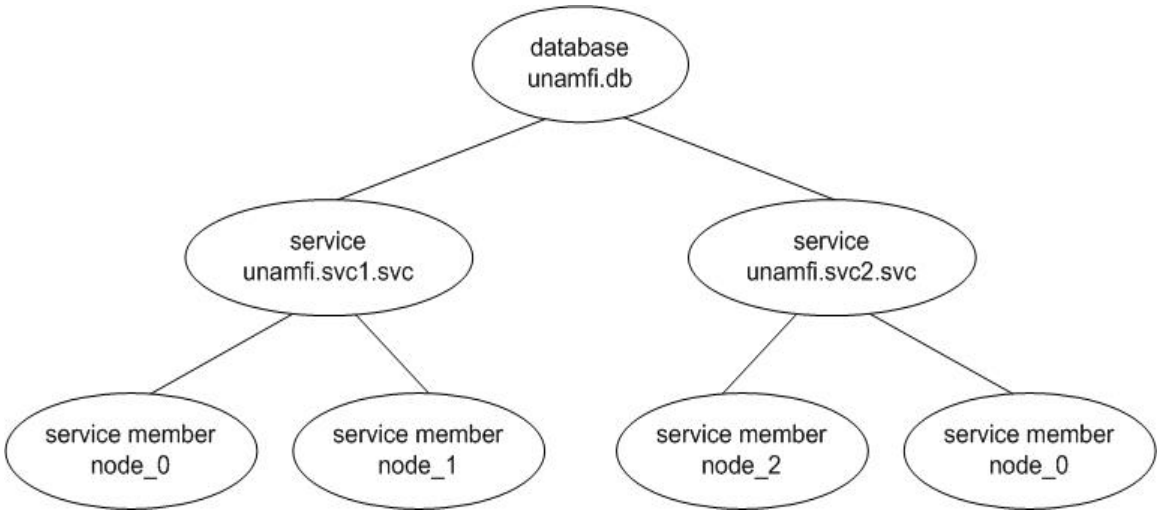

Fig. 17 : Esquema de múltiples servicios.

4.2.6.3. MÚLTIPLES REDES PÚBLICAS

A partir de la versión 11g, el sistema de alta disponibilidad permite al usuario contar con más de una red pública como mecanismo de aseguramiento de alta disponibilidad de recursos.

Esto se ve reflejado directamente en el número de recursos que el sistema de alta disponibilidad administra ya que para cada interfaz real, se debe contar con su respectiva representación lógica dentro del sistema de *cluster*. De esta manera, cada que se produce un cambio de estado, como bien puede ser una falla de la red pública, todos los recursos lógicos que dependan de esa red, pueden ser movidos a otro nodo.

Los eventos que se producen deben contener atributos que permitan identificar a qué red pública pertenece cada uno, asimismo deben satisfacer las políticas de jerarquía de orden.

En la figura 20 se muestra un esquema con múltiples redes públicas:

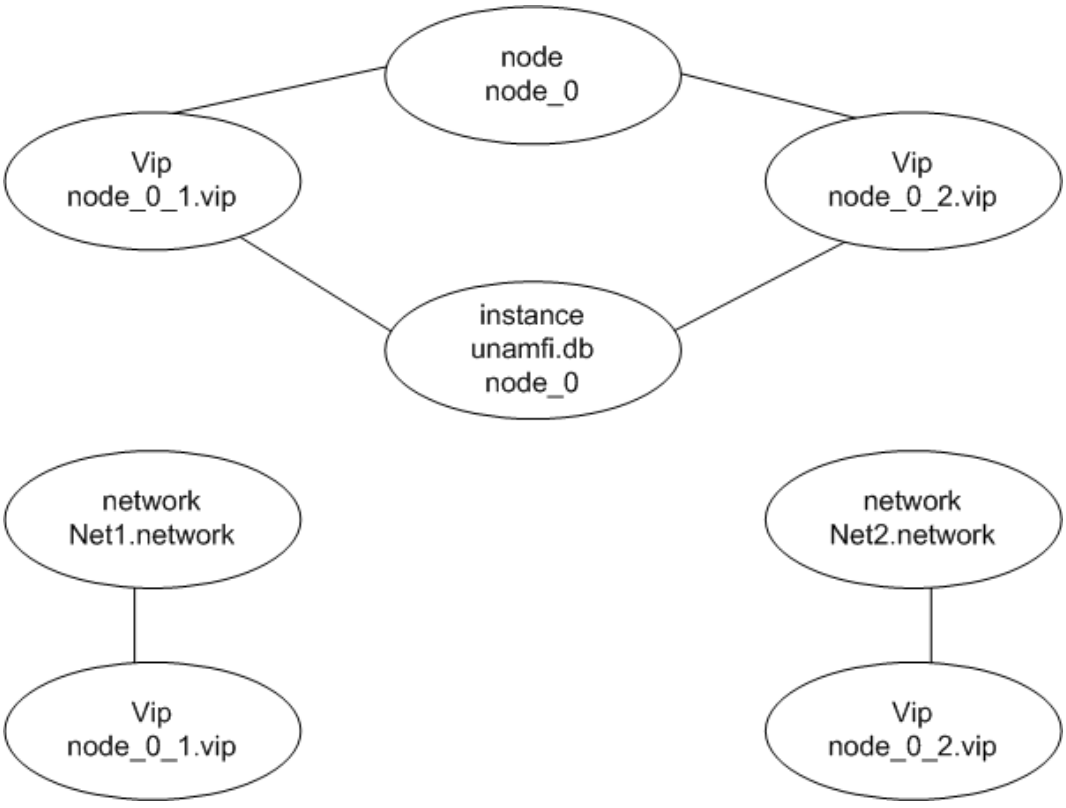

Fig. 18 : Esquema de múltiples redes públicas.

4.3. FUNCIONAMIENTO DE LA INFRAESTRUCTURA DE PRUEBAS

A continuación se presenta un ejemplo del procesamiento de eventos usando la infraestructura de verificación de eventos:

A. INICIALIZACIÓN
Dentro del script general de la prueba:

- Una macro levanta el sistema de alta disponibilidad en uno o más nodos. El esquema de alta disponibilidad depende del tipo de escenario y conducta que se quiera reproducir. Por lo general se usan escenarios en los que hay uno o más nodos y al menos una base de datos con tres servicios.

- Una macro agrega recursos para los que se recibirán eventos. Estos recursos pueden variar dependiendo de la conducta que se quiera reproducir.

B. REPRODUCCIÓN Y VERIFICACIÓN DE LA CONDUCTA
Dentro del script específico para la reproducción de la conducta:

- Una macro inicia el demonio que corresponde al sistema de servicios de notificación.

- Una macro inicia los diferentes clientes subscriptores bajo el esquema de alta disponibilidad establecido en la parte de inicialización.

- Antes de realizar una acción que represente un cambio en el estado de los recursos agregados, se toma el tiempo mediante el uso de una macro.

- Otra macro obtiene el estado de los recursos del cluster en ese momento y lo convierte a una estructura de datos de tipo árbol. Dicha acción se realiza ejecutando el comando sadctl, en las siguientes páginas se presenta la salida del comando, junto con su representación en una estructura de datos de tipo árbol.

```
$sadctl stat res -t
--------------------------------------------------------------------------
Name       Target  State    Server         State details
--------------------------------------------------------------------------
Local Resources
--------------------------------------------------------------------------
LISTENER.lsnr                                     STABL
             ONLINE ONLINE     node_0         E
                                               STABL
             ONLINE ONLINE     node 1         E
                                               STABL
             ONLINE ONLINE     node_2         E
net1.network                                      STABL
             ONLINE ONLINE     node_0         E
                                               STABL
             ONLINE ONLINE     node_1         E
                                               STABL
             ONLINE ONLINE     node_2         E
sn                                                STABL
             ONLINE ONLINE     node_0         E
                                               STABL
             ONLINE ONLINE     node 1         E
                                               STABL
             ONLINE ONLINE     node_2         E
--------------------------------------------------------------------------
Cluster Resources
--------------------------------------------------------------------------
SLSNR_SCAN1.lsnr
     1       ONLINE ONLINE     node 0         STABLE
SLSNR_SCAN2.lsnr
     1       ONLINE ONLINE     node_1         STABLE
node_0.vip
     1       ONLINE ONLINE     node_0         STABLE
node 1.vip
     1       ONLINE ONLINE     node_1         STABLE
node 2.vip
     1       ONLINE ONLINE     node_2         STABLE
unamfi.db                                         Open,STAB
     1       ONLINE ONLINE     node_0         LE
                                               Open,STAB
     2       ONLINE ONLINE     node_1         LE
                                               Open,STAB
     3       ONLINE ONLINE     node_2         LE
unamfi.svc1.svc
     1       ONLINE ONLINE     node_1         STABLE
     2       ONLINE ONLINE     node_0         STABLE
unamfi.svc2.svc
     1       ONLINE ONLINE     node_2         STABLE
     2       ONLINE ONLINE     node_0         STABLE
scan1.vip
     1       ONLINE ONLINE     node_0         STABLE
scan2.vip
     1       ONLINE ONLINE     node_1         STABLE
--------------------------------------------------------------------------
```

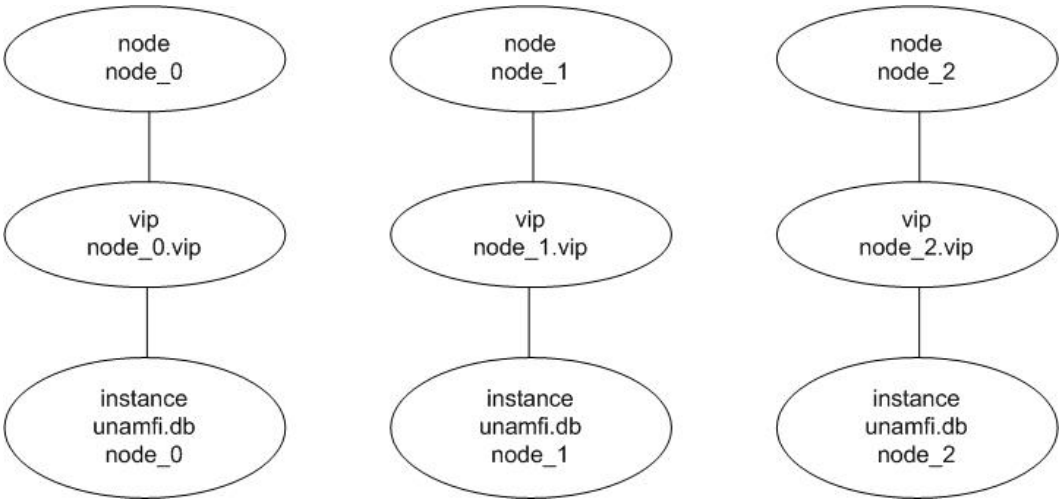

Fig. 19 : Árbol de recursos, nodos, vips e instancias.

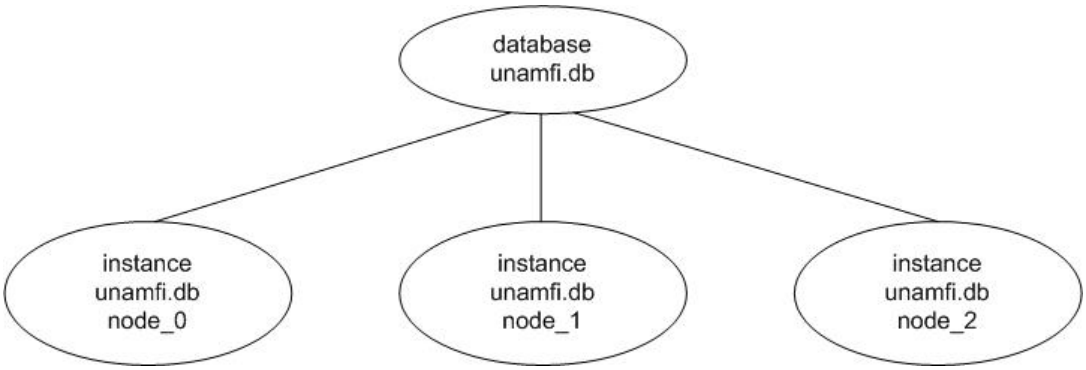

Fig. 20 : Árbol de recursos, base de datos e instancias.

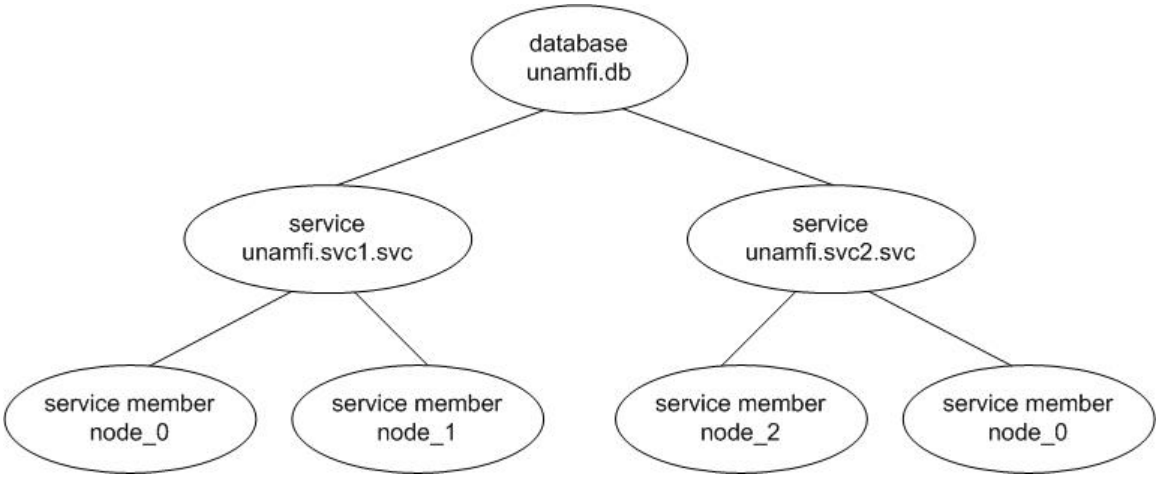

Fig. 21 : Árbol de recursos, base de datos y servicios.

- Se realizan acciones que cambian el estado de los recursos.

- Para cada cambio que se produce en un recurso determinado, los subscriptores crean un archivo cuyo nombre tiene una sintaxis particular que va a permitir el procesamiento posterior por parte de la macro de verificación de eventos. El archivo contiene toda la información que está relacionada con el cambio de estado del recurso, al conjunto de toda esa información se le conoce como *payload*.

- Se toma el tiempo y el estado de los recursos cuando ocurrió el cambio de estado. Para determinar el momento exacto en que un recurso cambió de estado se usa otra macro.

- Se usa la macro de verificación de eventos, la cual revisa que los eventos cumplan con las siguientes características:

    - Orden

        - Los eventos deben llegar en el orden apropiado de acuerdo a la jerarquía que el recurso tenga dentro del sistema de alta disponibilidad. Para realizar esta verificación se requiere que la estructura de los recursos del *cluster* hayan sido mapeados a una estructura de datos de tipo árbol.

        - La verificación de orden toma en cuenta el número de secuencia del evento, a medida que un nivel desciende, los eventos deben incrementar su número de secuencia.

    - Tiempo

        - Los eventos deben llegar dentro del rango establecido de tiempo obtenido con las macros en el punto 5 y 9.

    - Contenido
        - El contenido de los eventos se verifica con una comparación de archivos usando expresiones regulares.

Para el caso de la verificación de eventos del recurso de base de datos cuyo árbol está representado por la figura 22, cada servicio debe tener un número de secuencia N + 1, donde N representa el número de secuencia del evento que corresponde a la base de datos. De la misma manera, los números de secuencia de cada *service member* deben ser de N + 2. Sólo debe existir un solo evento de base de datos, dos eventos de servicios y cuatro de *service members*. Los eventos debieron haber llegado dentro de los *timestamps* establecidos, y los *payloads* deben ser congruentes con el archivo de comparación que contiene las expresiones regulares correspondientes al tipo de evento.

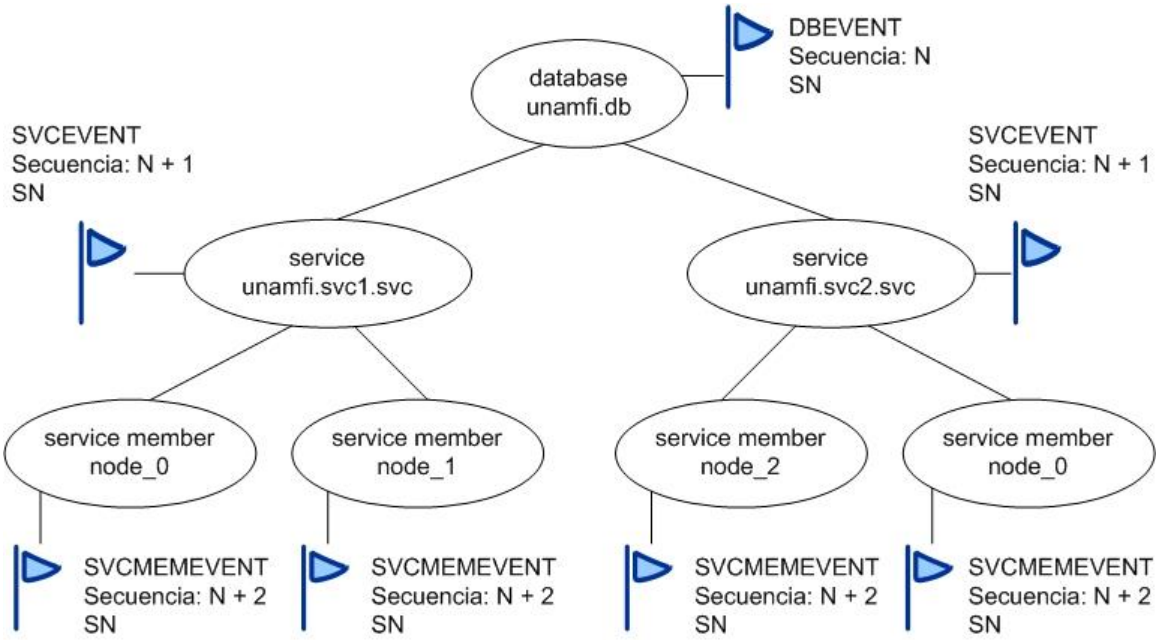

Fig. 22 : Verificación de los eventos de servicios.

- El resultado de la verificación de eventos produce información que permite realizar un diagnóstico en caso de que haya ocurrido un fallo en cualquiera de los aspectos verificados.
- El proceso se repite en caso de que haya más conductas a verificar.
- Se detienen los clientes subscriptores.
- Se detiene el demonio de servicios de notificación.

C. LIMPIEZA
Dentro del script general de la prueba:

- Se remueven los recursos de *cluster*.
- Se detiene el sistema de alta disponibilidad.

La estructura de las pruebas debe ser realizada de esta manera ya que cada ambiente se ejecuta diariamente como parte de las pruebas de regresión. Es posible que si una prueba no limpia correctamente los recursos que usó, la próxima a ser ejecutada tenga resultados negativos.

4.4. PARTICIPACIÓN PROFESIONAL EN EL PROYECTO

La labor del desarrollador de pruebas puede ser muy diversa y abarcar una gran cantidad de campos. De manera muy concreta, la clasificación de las labores que desarrollé en el proyecto es la siguiente:

- Reporté *bugs* de código de desarrollo y pruebas.
- Corregí *bugs* de código de desarrollo.
- Corregí *bugs* en código de pruebas.
- Diseñé y codifiqué una gran cantidad de nuevos casos de prueba.
- Realicé la documentación de los nuevos casos de prueba.
- Diseñé y desarrollé todos los clientes subscriptores mencionados en 4.2.1, entre otros.
- Diseñé y desarrollé la librería de procesamiento de eventos mencionada en 4.2.4.
- Diseñé, elaboré y revisé documentos de especificación de pruebas.
- Mejoré el sistema de verificación de eventos.
- Proporcioné mantenimiento constante del sistema de pruebas.
- Diseñé y desarrollé todos los ambientes de prueba mencionados en la sección 4.2.6.
- Diseñé y desarrollé una gran cantidad de macros y scripts tanto de propósito general como de propósito específico para el proyecto, dentro los que destacan los dedicados a la verificación de eventos. Todos los mencionados en la sección 4.2.2.
- Diseñé y desarrollé los archivos de referencia para la comparación del contenido de los eventos haciendo uso extensivo de expresiones regulares.
- Mantuve un seguimiento muy cercano al avance del proyecto mediante la asistencia a varias juntas por semana con los diferentes equipos de desarrollo.
- Impartí un curso de capacitación a los nuevos integrantes del equipo.

- Acudí a capacitación en las oficinas centrales de la empresa en Redwood, California.

4.5. ALCANCE DEL TRABAJO DESARROLLADO

Al ser NRE un elemento clave de SAD, resulta de suma importancia garantizar que su funcionamiento sea el apropiado. A grandes rasgos, NRE es el mecanismo del sistema de *clusterware* que notifica al usuario cuando un recurso del "*cluster*", como una base de datos, cambió de estado. No es muy difícil pensar en escenarios donde el mal funcionamiento de este componente resulta desastroso. Para cualquier compañía el hecho de no percatarse que la ejecución de su base de datos se detuvo representa grandes pérdidas. Por lo tanto, es imperativo asegurar que NRE funcione correctamente sobre cualquier plataforma y cualquier configuración, por más especial y rebuscada que ésta sea. Así, el conjunto de casos de prueba debe ser lo suficientemente extensivo para abarcar todos los aspectos que representen un riesgo para el cliente.

NRE es un componente incluido en varios productos de la empresa. Muchos de ellos son usados hoy en día por muchas empresas para agilizar y optimizar sus sistemas de almacenamiento y base de datos. Algunos de los clientes de éstos productos son empresas del tamaño de Hyundai, Claro, Allegis, Nintendo, entre muchos otros.

4.6. RESULTADOS Y APORTACIONES

Para la empresa, contar con personal que garantice la calidad de cada componente de sus productos es de suma importancia. Una solución de almacenamiento eficaz es uno de los factores que marcan la diferencia entre una aplicación que resuelve problemas, y una aplicación que los genera.

NRE representa una parte fundamental del sistema de *clusterware*. El conjunto de pruebas, herramientas y documentos de especificación que escribí, lo cual conforma el *framework* de testing, ha aumentado la calidad de este componente considerablemente, ya que se encontraron una gran cantidad de defectos en el código de producción del propio componente, y de aquellos con los que éste se relaciona, los cuales incluyen CA y JDBC (*Java Database Connectivity*). Las mejoras en el *framework* permiten que las pruebas sean ejecutadas de manera más rápida, pues al tener un mayor control sobre la estructura y funcionamiento de cada pequeña pieza, se pueden identificar aquellos factores que pueden ser corregidos y optimizados. Además del aumento en la velocidad, muchos otros factores se vieron beneficiados, tal es el caso del número de configuraciones sobre las cuales se ejecutaron las pruebas de verificación de eventos, las cuales incluyen, entre otras, escenarios de múltiples bases de datos y múltiples redes públicas.

El acercamiento a otros equipos de desarrollo ha jugado un papel fundamental en la creación de nuevos casos de prueba, solución de problemas e identificación de fallos. Un ejemplo muy importante es el del cliente consumidor de eventos OCISN (*Servicios de notificación de tipo OCI*), para éste caso en particular se trabajó en conjunto con el equipo de la base de datos para conocer a detalle todos los factores que debían ser tomados en cuenta y cubrir casi completamente todos los posibles fallos.

Cuando se encuentra un posible fallo, inmediatamente se establece un canal de comunicación con el equipo involucrado, las discusiones acerca de la posibilidad de que una conducta extraña en el producto sea un *bug*, es evaluada por los jefes de desarrollo en ambos lados. Usualmente este proceso no es muy tardado, sin embargo, el desarrollador de pruebas debe proporcionar toda la información necesaria para que la identificación del problema tarde el menor tiempo posible. Una vez identificada la causa, se deslinda la responsabilidad y se procede a llevar a cabo la solución.

Para el desarrollador que entra a una empresa de este tipo y que no ha tenido una experiencia previa similar, el cambio es muy drástico en una gran variedad de aspectos, que van desde la complejidad lógica de los problemas a resolver hasta el cambio de residencia e independencia económica. Los retos personales, académicos y profesionales se vuelven más ambiciosos, ya que cada vez se cuenta con más herramientas técnicas y sociales que son adquiridas al enfrentar problemas técnicos y colaborativos respectivamente.

Las ventajas que representa el poseer dichas habilidades son muy numerosas tanto para el individuo como para la empresa. Para la última es imperativo que el desarrollador resuelva los problemas de la manera más eficaz posible, pues así se acelera el proceso de desarrollo.

5. CONCLUSIONES

Los sistemas de *software* son quizá las creaciones humanas más complejas en términos del número y tipo de elementos que los componen. A medida que el avance tecnológico continúe, cada vez será más difícil poder predecir y evaluar con exactitud el comportamiento de un sistema artificial, tal como lo dice *Heisenberg* con su principio de incertidumbre. Cada desarrollador de *software* que contribuye con cambios en el código, está integrando sus ideas y razonamientos lógicos a manera de líneas de código en uno o más archivos de texto. Por ejemplo, el manejador de base de datos está compuesto por millones de líneas de código que, en otras palabras, son las ideas que representan la mejor solución para todos los desarrolladores involucrados.

Todas estas ideas formuladas en un lenguaje computacional, son procesadas y ejecutadas exactamente como fueron definidas para solucionar una necesidad, en el caso particular del manejador de base de datos: la de eficiencia de almacenamiento. La calidad de la solución está determinada exclusivamente por cómo el *software* fue concebido e implementado por los miembros del equipo de desarrollo de *software* durante todo el ciclo.

Debido a la naturaleza de los desarrolladores, es imposible pensar en una solución de *software* que sea completamente perfecta. Siempre van a existir minúsculos defectos que, aunque inicialmente son invisibles, poco a poco crecen hasta representar un problema severo. Las predicciones del buen funcionamiento del *software* son válidas sólo para un corto periodo de tiempo ya que irremediablemente existirán determinantes que escaparán de la capacidad de control y administración. De ésta manera, lo único que podemos asegurar, es que este caótico comportamiento va a continuar ocurriendo por un lapso indefinido.

Lo anterior representa la necesidad de un grupo de mentes especializadas únicamente en rastrear estos defectos, al igual que si se tratara de un equipo de peritos forenses buscando las causas de un delito y planeando su futura prevención. Las cualidades necesarias del desarrollador de pruebas son: una personalidad inquisitiva, una desconfianza e incredulidad fuera de lo ordinario; y en segundo lugar, la inteligencia para diseñar e implementar estrategias usando las herramientas adecuadas, siendo la automatización la más importante de ellas.

Así se creó la labor del desarrollador de pruebas, que constituye un pilar fundamental en el proceso de desarrollo de *software* de cualquier empresa cuyo propósito sea el de garantizar la buena calidad de su producto y así obtener mayores ganancias. El desarrollador de *software* debe crear un sistema que, de manera vigorosa, asegure que todas las funcionalidades establecidas en los documentos de especificación sean cumplidas a detalle y que lo hagan de manera óptima.

Es parte del desarrollo normal del ser humano probar todos los aspectos que componen el ambiente en el que vive, sería antinatural no tener un mecanismo para probar un sistema tan complejo como el que representa el *software*. El proceso de *testing* puede representar un mayor uso de recursos y costos, pero es necesario para reducir considerablemente el grado de entropía en el sistema, ya que se tiene un registro certero de los fallos ocurridos, de la identidad del fallo que está ocurriendo, y se puede realizar una mejor estimación de los que están por ocurrir. Empresas que han dejado de lado su proceso de *testing* porque supuestamente representa una innecesaria pérdida de recursos, han tenido muy malas experiencias incluso durante la presentación de su producto en las conferencias de prensa.

Hasta ahora, la teoría relacionada con el proceso de *software testing* es muy escasa, y en muchos casos, con diferencias entre sí, debido a que se trata de una disciplina altamente evolutiva, se encuentra en constante cambio porque las necesidades cambian radicalmente con el tiempo; las metodologías para realizar pruebas en un escenario son completamente ineficientes en otro. Los factores determinantes para construir un protocolo de pruebas efectivo para un problema en particular, son únicamente la experiencia y criterio del desarrollador quien también funge como arquitecto. Por lo anterior, hoy en día se considera que ésta disciplina en realidad es un arte.

6. GLOSARIO

AIX

Es un sistema operativo creado por IBM, basado en UNIX, inicialmente fue usado en mainframes IBM, pero con el tiempo la lista de plataformas soportadas se ha ido incrementando. Fue puesto a la venta por primera vez en el año 1986.

CA

Se trata de un sistema de almacenamiento de mensajes que usa una estructura de datos de tipo *cola*. Es usado principalmente por el lenguaje PL/SQL.

Alta disponibilidad

Es un esquema de diseño en el que se asegura un cierto grado de continuidad operacional en un sistema determinado. Bajo un esquema de alta disponibilidad, los usuarios deben poder acceder a los recursos del sistema en cualquier momento y situación.

Árbol de datos

Se trata de una estructura de datos usada comúnmente en computación. Tiene mucha similitud a un árbol orgánico, en el sentido de las relaciones de rama y hoja, existen varios tipos que responden a problemas distintos. Por lo general el árbol se construye a partir de un *nodo inicial* o *raíz*, el cual puede tener “hijos”, los cuales a su vez pueden tener más ramificaciones. A un nodo que no tiene hijos se le llama *nodo hoja*, de manera análoga, un nodo que tiene hijos se le conoce como *nodo rama* o *nodo padre.*

Arquitecto de *software*

Es un desarrollador de *software* que toma decisiones de alto nivel y dictamina estándares técnicos, los cuales incluyen esquemas de escritura de código, herramientas y plataformas.

Automatización

Se trata del uso de mecanismos de control para poder operar un sistema o llevar a cabo una tarea repetitiva sin asistencia humana.

Base de datos

Se trata de un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su uso posterior.

Bug

Se trata de un fallo en un programa o sistema computacional que fue originado generalmente por errores humanos en el diseño o implementación de un programa, un conjunto de herramientas o sistema operativo.

CEO

Máximo título de rango para un ejecutivo que esté a cargo de una organización. Sus siglas vienen del inglés *"Chief Executive Officer".*

Centro de desarrollo

Se trata de una de las sedes de una empresa de *software* que se especializa exclusivamente en el desarrollo del código de alguno o varios de los productos.

Cluster

Se trata de un sistema que está compuesto de dos o más computadoras y que debe contar con la habilidad de administrar los recursos de manera distribuida. Al conjunto de todas las computadoras o sistemas se le puede ver como uno solo.

Clusterware

Es un *software* de cluster que permite la creación de un esquema de clustering con servidores independientes que funcionan como un sistema único.

Coverage

Es un enfoque para identificar y realizar casos de prueba basados en el número de componentes lógicos de interés en un programa o sistema. Por lo general, se crea un caso de prueba para cada función. Una de sus ventajas es que se tiene un control estadístico del número de pruebas realizadas.

Coverage Metrics

Son las estadísticas que se generan al usar un enfoque de coverage para la identificación de los casos de prueba.

DLM (Shared everything)

Por sus siglas en inglés *Distributed Lock Manager*. Son usados como mecanismo fundamental para realizar aplicaciones distribuidas en esquemas de *clustering*. Las máquinas que componen el *cluster* pueden usar el almacenamiento de las demás usando un sistema de archivos unificado. Produce grandes ventajas respecto al desempeño y disponibilidad.

Debugger

Es un programa cuya función radica en probar y depurar otros programas. Permite que el desarrollador visualice en tiempo real el valor de las variables, las direcciones de memoria y las instrucciones que se están ejecutando. Cuando se presenta un problema muestra el lugar exacto del código fuente donde ocurrió.

Developer

Se trata del desarrollador de *software* que se encarga de diseñar programas, escribir código fuente y corregir *bugs*. Por lo general se trata de un Ingeniero especializado en *software*.

Diagrama de Venn

Es una forma de representar relaciones lógicas de conjuntos finitos. Cada círculo y las superposiciones entre ellos representan un conjunto y las relaciones entre ellos respectivamente. Representan un elemento fundamental en la teoría de conjuntos, son usados en probabilidad, lógica, estadística, lingüística y ciencias de la computación.

Documento de especificación de diseño

En él se especifica la funcionalidad del proyecto mediante diagramas de flujo, por lo general los elementos son representados por entidades de las cuales no se conoce su implementación. Muestra de una forma muy general el comportamiento global del sistema. En base a él se elabora el documento de implementación.

Documento de especificación de implementación

Se establecen de manera muy detallada todos los elementos que serán usados para construir la infraestructura detrás del proyecto. Se describen las clases, funciones con sus parámetros y valores de retorno, algoritmos a usar y variables. Se usan otro tipo de diagramas como el UML.

Documento de especificación de pruebas

Es el documento en el que están descritos todos los casos de prueba que serán implementados para el proyecto. Dichos casos de prueba son elegidos dependiendo del tipo de enfoque utilizado.

Entorno integrado de desarrollo

Se trata de un conjunto de herramientas que optimizan el proceso de escritura y depuración de código fuente. Ofrece al desarrollador un ambiente integrado con editores de texto, depuradores y acceso a las herramientas de compilación.

Expresión regular

Es una secuencia de caracteres que conforma un patrón de búsqueda de cadenas. Son usados en computación generalmente para buscar y reemplazar patrones así como para validar entradas del usuario. Su gramática está determinada por un conjunto de reglas muy estrictas.

Gerente de calidad

El gerente de calidad se encarga de verificar que las transacciones que se quieren integrar a las versiones diarias del código cumplan con valores numéricos de calidad, además de corroborar que los cambios no representan un peligro.

Grafo

Es la representación de un conjunto de objetos en la que cada objeto corresponde a un nodo, y están unidos entre ellos por enlaces llamados aristas o arcos. Son el objeto de estudio de la teoría de grafos. Existen principalmente dos tipos de grafos, los dirigidos y los no dirigidos.

HP-UX

Es la implementación de Unix realizada por Hewlett-Packard. Fue liberada por primera vez en 1984. Y fue el primero en ofrecer listas de control de acceso para permisos de acceso de archivos como una alternativa al esquema clásico de permisos de Unix.

Host

En su definición más básica, se trata de una computadora que está conectada a la red. Un *cluster* está integrado por uno o más *hosts*, cada uno puede realizar actividades distintas. Comúnmente se denomina "nodo".

IEEE

Por sus siglas en inglés *"Institute of Electrical and Electronic Engineers",* es una asociación profesional con oficinas centrales en Nueva York, dedicada a mejorar la excelencia en la innovación de avance tecnológico. Cuenta con una gran cantidad de publicaciones, entre las que destacan los estándares de uso y desarrollo de tecnologías, dentro de las cuales se encuentran las de ingeniería en computación.

Infraestructura de pruebas

Se llama así a todo el conjunto de herramientas y pruebas que verifican el buen funcionamiento de un componente o sistema de *software*.

Instancia de base de datos

Conjunto de estructuras de memoria que administran archivos de bases de datos.

Internship

Durante éste periodo, el interno está sometido a un proceso de aprendizaje de las tecnologías que una empresa usa. No está contratado como empleado regular, ya que por lo general, el proceso no es mayor a un año.

PL/SQL

Es un lenguaje usado para procesar comandos de SQL, que cuenta con una sintaxis específica para ese propósito. Cuenta con los mismos tipos de datos que SQL. Los scripts son almacenados, compilados por la base de datos y ejecutados en el binario de Oracle.

Payload

Se llama así al conjunto de atributos que componen un evento de tipo NRE.

Proceso

Se trata de la instancia de un programa de computadora que está ejecutándose en un momento determinado. Contiene el código del programa y su respectiva actividad. Dependiendo del sistema operativo puede incluso estar compuesto por múltiples hilos que ejecutan instrucciones concurrentes.

QoS

Conjunto de prácticas que satisfacen requerimientos de calidad y desempeño específicos y estrictos, como pueden ser el tiempo de respuesta de un servicio, pérdidas, reducción de ruido, eco, interrupciones, respuesta en frecuencia. Con ellas se puede garantizar la reservación y priorización de recursos a aplicaciones y usuarios.

SDET

Por sus siglas en inglés *"Software development engineer in test"* , es la clasificación de Microsoft para referirse a los desarrolladores de pruebas.

Script

Es un programa de computadora escrito en un lenguaje que sólo requiere ser interpretado y no compilado. Es ejecutado por algún ambiente especial.

Server pool

Se le llama así al lugar virtual de donde son tomados todos los recursos que conforman un *cluster*.

Servicio de base de datos

Son procesos que se encargan de dividir la carga de trabajo de la base de datos en diferentes grupos. Cada servicio representa una carga de trabajo con atributos comunes y prioridades. Las peticiones a la base de datos se realizan a través de los servicios.

Sistema distribuido

Se trata de un modelo de computación en el que se resuelven problemas utilizando más de un equipo de procesamiento o almacenamiento.

Solaris

Es un sistema operativo creado por Sun Microsystems en el año 1993. Esta basado en Unix. Es un sistema altamente escalable y a lo largo de los años ha incluido una gran cantidad de innovaciones como el sistema de archivos ZFS. Soporta procesadores SPARC y x86.

TODO

Tipo de comentario en el código fuente que se refiere a las actividades que aún no se realizan pero que son necesarias y deben ser implementadas en el futuro.

Tester

Se le llama así también al desarrollador de pruebas.

Timestamp

Cadena que registra de manera detallada un momento en el tiempo. Incluye la fecha, hora y zona horaria.

Transacción de *software*

Es un elemento atómico que está compuesto de todos los cambios que un desarrollador realiza para incluir una nueva funcionalidad, o bien, para hacer una corrección al sistema. Requiere ser aprobada para que pueda ser integrada a la versión diaria del *software*.

zOS

Es un sistema operativo de 64 bits creado por IBM para las computadoras que funcionan en un esquema de *mainframe*. Es el sucesor del OS/390. Es un sistema operativo basado en Unix.

7. REFERENCIAS

7.1. SOFTWARE TESTING

[Jorgensen08] Jorgensen, Paul C., *Software Testing, A craftsman's approach,* Tercera edición, Auerbach Publications, 2008

[Hetzel1988] Hetzel, William C., *The Complete Guide to Software Testing,* Segunda edición, Wellesley, Mass, QED Information Sciences, 1988.

[Myers79] Myers, Glenford J., *The art of software testing,* Wiley, New York, 1979.

[Goucher09] Goucher, Adam. *Beautiful Testing: Leading Professionals Reveal How They Improve Software (Theory in Practice),* Primera edición, O'Reilly, 2009

[Jiantao99] Jiantao Pan. *Software Testing,* Carnegie Mellon University, 1999. Recuperado de: http://goo.gl/0EYiV

[Kaner06] Kaner Cem. *Exploratory testing,* QAI, 2006. Recuperado de: http://goo.gl/uHcJJu

[Bach13] Bach, John., *Seven kind of testers, 2013.* Recuperado de: http://www.satisfice.com/blog/archives/893

[IEEE10] 1044-2009 IEEE Standard Classification for Software Anomalies, 2010

7.2. ALTA DISPONIBILIDAD

[Marcus03] Marcus, Evan; Stern, Hal. *Blueprints for High Availability,* Segunda edición, Wiley, 2003

[Gopala11] K Gopalakrishnan. *Oracle Database 11g, Oracle Real Application Cluster Handbook*, Segunda edición, Oracle Press, 2011.

7.3. TIEMPOS DE ENTREGA

[Brooks95] Brooks Jr, Frederick. *The Mythical Man-Month: Essays on Software Engineering,* Segunda edición, Wesley, 1995.

7.4. ENTREVISTAS DE SOFTWARE

[Mongan07] Mongan, John; *et al*, *Programming Interviews Exposed: Secrets to Landing Your Next Job*, Segunda edición, Wrox, 2007.

7.5. BUENAS PRÁCTICAS DE PROGRAMACIÓN

[Martin08] Martin , Robert C. *Clean Code*: *A Handbook of Agile Software Craftsmanship*, Primera edición, Prentice Hall, 2008.

[Kernighan07] Kernighan. Brian. *A regular expression matcher,* Princeton, 2007
Recuperado de: http://goo.gl/NJH0Zv

7.6. TEORÍAS EVOLUTIVAS

[Piaget69] Piaget, Jean. *Estudios de Psicología*. 1ra Parte, Seix Barral. *Barcelona,* 1969

[Vigotsky79] Vigotsky L.S. *El desarrollo de los procesos psicológicos superiores*. Caps. II y VI. *Barcelona*, 1979

8. ANEXOS

8.1. FALLOS DE SOFTWARE DE ACUERDO A LA IEEE

Los fallos del *software* se suelen clasificar de la siguiente manera [IEEE10]:

Tipo	**Instancias**
Entrada	La entrada correcta no fue aceptada La entrada incorrecta fue aceptada No hay descripción o está incompleta Parámetros incorrectos o inexistentes
Salida	Formato incorrecto Resultado incorrecto Resultado correcto en tiempo no indicado Resultado inexistente o incompleto Resultado espurio Error gramatical u ortográfico Detalle cosmético

Tabla 1: Fallos de operaciones de entrada y salida.

Casos faltantes Caso duplicados Condición de límite rechazada Mala interpretación Condición faltante Condición incorrecta Verificación de la variable equivocada Ciclo de iteración incorrecto Operador incorrecto (por ejemplo < en vez de <=)

Tabla 2: Fallos lógicos.

Algoritmo incorrecto
Procesamiento faltante
Operando incorrecto
Operación incorrecta
Error de paréntesis
Precisión insuficiente

Tabla 3: Fallos de procesamiento.

Error en el manejo de interrupciones
Mala sincronización de entrada y salida (I/O)
Llamada al procedimiento equivocado
Llamada a un procedimiento inexistente
Parámetros incongruentes (tipo o número)
Tipos de dato incompatibles
Inclusión superflua

Tabla 4: Fallos de interfaz.

Inicialización incorrecta
Almacenamiento y acceso incorrectos Índice incorrecto
Compresión y extracción incorrecta
Uso de una variable equivocada
Uso de una referencia incorrecta Error de escalamiento de datos Tamaño de datos errónea Subíndice incorrecto
Tipo de dato incorrecto
Error de contexto
Acceso fuera de límites
Datos inconsistentes

Tabla 5: Fallos de datos.

8.2. TESTING FUNCIONAL

Verifica el funcionamiento del *software* considerándolo una caja negra, una máquina que para cierto rango de entradas tiene un dominio de salidas específico. Se trata de un enfoque muy usado en el análisis de sistemas físicos en muchos otros campos de la ingeniería. Usando este enfoque, sólo se necesita el documento de especificación de diseño para poder construir el conjunto de casos de prueba.

Fig. A : Modelo de caja negra.

Las ventajas principales son, a) que los casos de prueba son completamente independientes de la implementación del *software*, de manera que si éste llega a sufrir cualquier tipo de cambio en su estructura, la prueba debería seguir funcionando sin tener que realizar modificaciones. Otra ventaja es, b) que dado que sólo se requiere el documento de especificación de diseño, el desarrollo de las pruebas puede ser realizado al mismo tiempo que el código del *software* a verificar. Las desventajas principales son que se genera mucha redundancia entre casos de prueba y que existe una gran posibilidad de que haya múltiples aspectos sin probar.

Usando el enfoque de *testing* funcional se tienen los siguientes mecanismos para identificar casos de prueba:

- Análisis de valores de frontera
- Pruebas de robustez
- Análisis del peor caso
- Prueba de valores especiales
- Equivalencia de rango
- Equivalencia de dominio
- Análisis basado en tablas de decisión

8.3. TESTING ESTRUCTURAL

Se considera al *software* como una caja a través de la cual se puede visualizar el funcionamiento interno, es llamado por varios autores, “de caja gris”, o bien, “de caja transparente”. En este enfoque es necesario conocer teoría de grafos para poder hacer una representación eficaz de cómo el *software* está implementado, para posteriormente realizar los casos de prueba de una manera procedimental. A esto se le conoce como *coverage metrics*.

Los *coverage metrics* proveen una descripción detallada de qué componentes del *software* están siendo probados, esto brinda una mejor gestión de los casos de prueba.

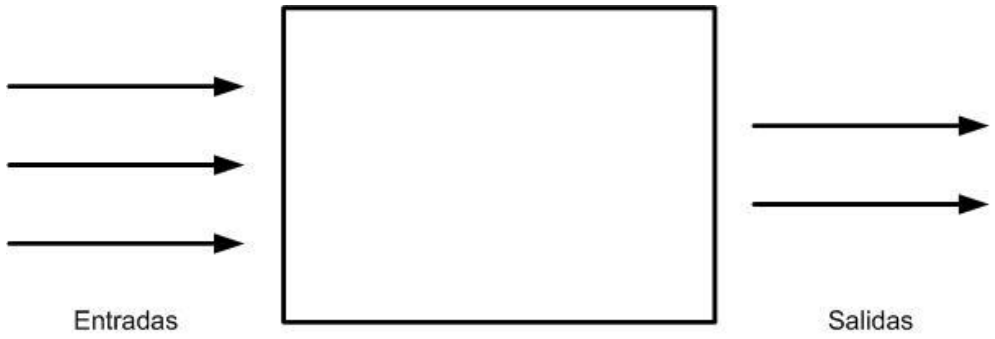

Fig. B : Modelo de caja transparente.

La principal desventaja del *testing* estructural es que dado que la identificación de casos de prueba está basada exclusivamente en la implementación, cualquier comportamiento que no esté programado en el código no será verificado.

Las metodologías de identificación de casos de prueba en éste enfoque son principalmente dos:

- Análisis basado en una ruta
- Análisis basado en el flujo de datos

8.4. ALTA DISPONIBILIDAD

Que un sistema o aplicación provea de alta disponibilidad (*High Availability*) significa que estará disponible todo el tiempo sin importar el clima, el lugar, el momento del día, y todos aquellos factores que pudieran representar una interrupción del servicio.

Las tecnologías más usadas para garantizar la alta disponibilidad están basadas principalmente en la redundancia de recursos, entre los que se pueden encontrar fuentes de alimentación, dispositivos de red como interfaces y *routers*. Inclusive se pueden llegar a tener diferentes centros de datos en una misma o diferente área para proveer un alto nivel de disponibilidad y de balanceo de cargas.

8.5. TOLERANCIA A FALLOS

Un sistema se cataloga como tolerante a fallos si está diseñado de manera que cuando se presenta un fallo, haya un componente o procedimiento que pueda ejecutarse inmediatamente para proporcionar el respaldo necesario que evite la pérdida del servicio. La tolerancia a fallos puede ser de *software*, hardware o la combinación de ambos. Las configuraciones más usadas de tolerancia a fallos son las siguientes:

Activo - Pasivo

Se le llama también asimétrico. Es un esquema que está conformado por dos nodos principalmente. Consiste en que mientras uno de los nodos está activo y realiza todo el procesamiento crítico del *cluster*, como el manejo de memoria y almacenamiento, además de ejecutar las aplicaciones, el otro simplemente se encuentra en estado de reposo, listo para ser usado en caso de que el primer nodo falle.

La desventaja de esta configuración es que se deben adquirir al menos dos equipos del mismo tipo; de los cuales sólo uno estará activo en un momento determinado. Ambos consumen la misma cantidad de electricidad, esfuerzo de administración y espacio físico. Sin embargo este esquema es la base para muchos sistemas de *clustering* actuales. La gran mayoría de ellos se componen de un servidor maestro y uno o más esclavos.

Activo - Activo

Se le conoce como asimétrico y al igual que en la configuración activo - pasivo se conforma principalmente de dos nodos. La diferencia es que ambos nodos están realizando procesamiento crítico al mismo tiempo. En caso de que uno falle, el que queda activo debe de satisfacer todas las peticiones que se le hacían al que falló, hasta que éste regrese a su servicio normal.

Aunque se trata de un esquema muy eficiente, tiene dos grandes desventajas. La primera es el riesgo de que el servidor que falló no pueda ser iniciado de nuevo, para lo cual todas las aplicaciones que corran en el *cluster* deben estar minuciosamente configuradas para poder trabajar en un sólo nodo. La segunda gran desventaja es la cantidad de trabajo extra que debe realizar el nodo que está respaldando al nodo fallido. El sistema debe estar preparado con los recursos de procesamiento (CPU) y memoria para poder satisfacer la totalidad de la demanda emergente.

8.6. RECUPERACIÓN DE DAÑOS DESPUÉS DE UN DESASTRE

Es la habilidad de continuar con la operación normal después de un incidente de gran impacto negativo, como puede ser la destrucción de un centro de datos y todo lo que haya dentro de él. En un escenario típico de recuperación de daños, transcurre una ventana de tiempo significativa antes de que el centro de datos pueda restablecer las operaciones, y se requiere reintroducir alguna cantidad de información para acceder a los respaldos.

8.7. TOLERANTE A DESASTRES

Algunos autores lo consideran un arte y ciencia debido a la gran complejidad que representa lograr que un negocio continúe con sus operaciones normales al mismo tiempo que un incidente grave está ocurriendo. Es mucho más difícil garantizar tolerancia a desastres que recuperación de los mismos, debido a que involucra un diseño muy detallado de la infraestructura de la empresa, además de que el cliente no debe darse cuenta de ningún efecto adverso, todos los incidentes deben ser invisibles para él. Por lo general, este tipo de soluciones son muy costosas.

8.8. DISTRIBUCIÓN DE RECURSOS DE ALMACENAMIENTO

Shared nothing

Está compuesta por un grupo de servidores independientes. Cada uno de ellos debe realizar una parte de la carga de trabajo en específico. Si por ejemplo, un número N de servidores componen al *cluster*, la carga de trabajo total es dividida entre el número de nodos y cada uno de ellos realizará una tarea en específico. La desventaja más grande, es que requiere que se seleccione detalladamente la labor que cada servidor va a realizar, además de que no es posible añadir nuevos servidores dinámicamente sin hacer cambios en el esquema de carga de trabajo, lo cual no es una opción escalable.

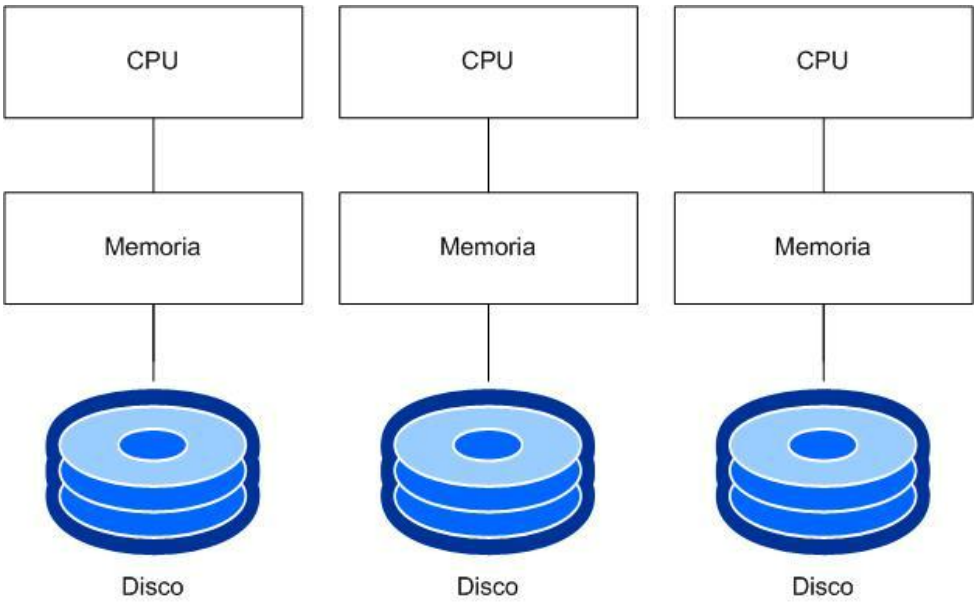

Fig. C : Arquitectura *Shared Nothing.*

Shared disk

Para lograr alta disponibilidad, es necesario que se comparta el acceso a los datos. En una aplicación pequeña en la que se use la arquitectura *shared disk*, el *cluster* puede hacer que un nodo tome el control del almacenamiento si otro nodo falla. Sin embargo, para una aplicación muy grande esta arquitectura se puede volver un poco problemática, dado que muchas aplicaciones corriendo en múltiples nodos pueden intentar acceder a los datos al mismo tiempo, por lo que el nodo que se encuentra despachando los datos puede volverse un cuello de botella. Para determinar el rol de los nodos se usa un mecanismo de *heartbeat,* mediante el cual, los nodos se comunican entre sí y en caso de que uno de los nodos no responda, el proceso de recuperación se inicia.

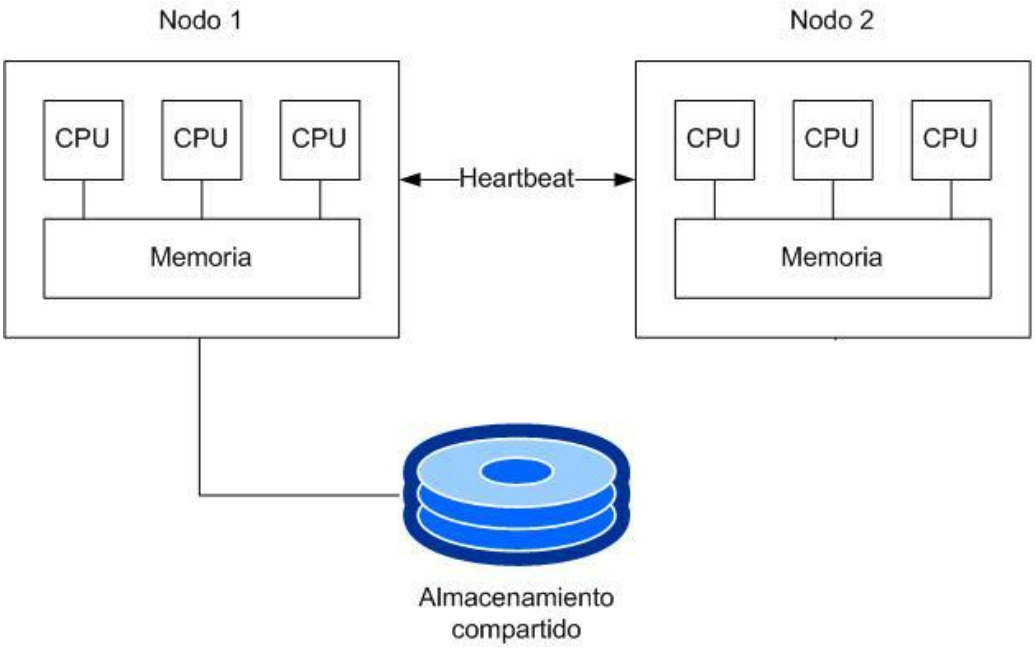

Fig. D : Arquitectura *shared disk.*

Shared everything

En la arquitectura *shared everything* el almacenamiento está disponible para todas las máquinas que componen el *cluster*. Usan un canal común para acceder de manera directa a los dispositivos de almacenamiento ya que a diferencia de la arquitectura *shared disk*, todos los nodos podrían intentar hacer escrituras o lecturas de manera simultánea sin necesitar de un mecanismo que controle el acceso por nodo. Dado que todos los nodos tienen la misma capacidad de acceder a la información, es necesario contar con un agente de sincronización que asegure la coherencia del sistema. Al mismo tiempo, es necesario contar con un sistema de archivos especializado, en tener la misma vista de la información contenida en los dispositivos de almacenamiento, independientemente del nodo en el que se ejecute una petición.

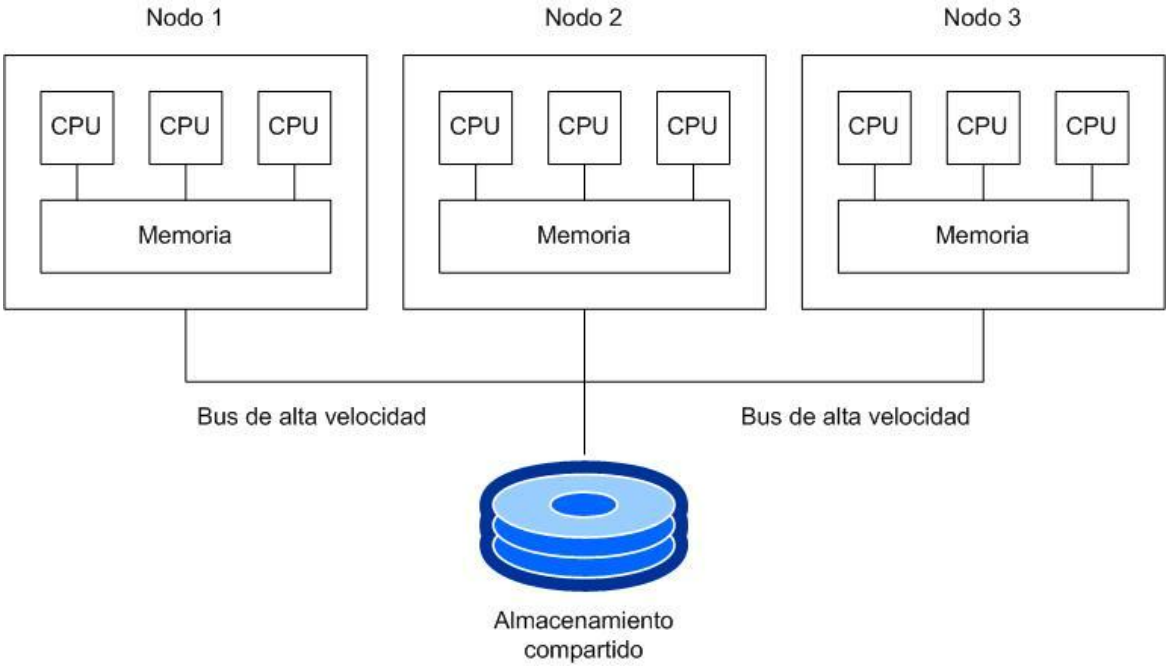

Fig. E : Arquitectura *Shared Everything*.